The Ciao System

A New Generation, Multi-Paradigm Programming Language and Environment
(Including a State-of-the-Art ISO-Prolog)
REFERENCE MANUAL
The Ciao Documentation Series
`https://ciao-lang.org/`
Generated/Printed on: 22 July 2023

Technical Report CLIP 3/97-1.22
Version 1.22 (2022/9/28, 17:41:23 CET)

Edited by:
Francisco Bueno
Manuel Carro
Manuel Hermenegildo
Pedro López
José F. Morales

The Computational logic, Languages,
Implementation, and Parallelism (CLIP) Lab
`https://www.cliplab.org/`
School of CS, T. U. of Madrid (UPM)
IMDEA Software Institute

Table of Contents

1 Summary

Ciao is a *general purpose*, *multi-paradigm* programming language in the Prolog family. Its modular design allows working with *fully declarative subsets* of Prolog and also to *extend* these subsets both syntactically and semantically. Most importantly, these restrictions and extensions can be activated separately on each program module so that several extensions can coexist in the same application for different modules, providing both clear program semantics and the precise assumptions required to perform effective global program analysis and static debugging and optimization.

The Ciao language is intended to support several kinds of uses:

- **Ciao** can be used as *a classical Prolog* and *logic programming* system. It includes standard **ISO-Prolog** as a sublanguage and a program development environment similar to that of traditional Prolog and *constraint logic programming* (CLP) implementations. The **Ciao** programming environment includes a classical top level and a rich Emacs interface with an embeddable source-level debugger, a number of execution visualization tools, and several useful extensions such as alternative control rules (breadth-first search, iterative deepening, ...), and several constraint domains.
- **Ciao** includes many *modern general-purpose programming language* features and idioms, such as functions, higher-order (with predicate abstractions), mutables, records, persistence, concurrency (threads/engines), parallel execution, or distributed execution. **Ciao** includes an **assertion language** for declaring (*optional*) program properties, automatic static inference and static/dynamic checking of such assertions, testing, documentation, etc. (some tasks performed by the **CiaoPP Program Processor** (`ciaopp`)).
- **Ciao** supports *programming in the large* with separate/incremental compiler, an integrated build system, an automatic documentation generator (`lpdoc`), and a set of standard and contributed libraries providing useful data structures, access to the operating system and networking, Web programming, as well as foreign interfaces, and several bindings to different languages and systems.
- **Ciao** supports *programming in the small* through its dynamic language features (dynamically typed, dynamic program modification). It supports producing small executables (including only those libraries actually used by the program) and writing scripts.
- **Ciao** is also a very good platform for experimenting with new language features, languages, analyses, and program transformations.

The **Ciao** distribution consists in a complete *public domain*, open-source implementation, very competitive with other commercial and academic languages (including other Prolog/CLP systems), and distributed under the GNU Library General Public License (LGPL).

This documentation corresponds to version 1.22 (2022/9/28, 17:41:23 CET).

2 Introduction

2.1 About this manual

This is the *Reference Manual* for the Ciao development system. It contains basic information on how to install Ciao and how to write, debug, and run Ciao programs from the command line, from inside GNU `emacs`, or from a windowing desktop. It also documents all the libraries available in the standard distribution.

This manual has been generated using the *LPdoc* semi-automatic documentation generator for LP/CLP programs [HC97, Her00]. `lpdoc` processes Ciao files (and files in Prolog and other CLP languages) adorned with assertions and machine-readable comments, which should be written in the Ciao assertion language [PBH97, PBH00]. From these, it generates manuals in many formats including `postscript`, `pdf`, `texinfo`, `info`, `HTML`, `man`, etc., as well as on-line help, ascii `README` files, entries for indices of manuals (`info`, WWW, ...), and maintains WWW distribution sites.

The big advantage of this approach is that it is easier to keep the on-line and printed documentation in sync with the source code [Knu84]. As a result, *this manual changes continually as the source code is modified*. Because of this, the manual has a version number. You should make sure the manual you are reading, whether it be printed or on-line, coincides with the version of the software that you are using.

The approach also implies that there is often a variability in the degree to which different libraries or system components are documented. Many libraries offer abundant documentation, but a few will offer little. The latter is due to the fact that we tend to include libraries in the manual if the code is found to be useful, even if they may still contain sparse documentation. This is because including a library in the manual will at the bare minimum provide formal information (such as the names of exported predicates and their arity, which other modules it loads, etc.), create index entries, pointers for on-line help in the electronic versions of the manuals, and command-line completion capabilities inside `emacs`. Again, the manual is being updated continuously as the different libraries (and machine-readable documentation in them) are improved.

2.2 About the Ciao development system

The Ciao system is a full programming environment for developing programs in the Prolog language and in several other languages which are extensions and modifications of Prolog and (Constraint) Logic Programming in several interesting and useful directions. The programming environment offers a number of tools such as the Ciao standalone compiler (`ciaoc`), a traditional-style top-level interactive shell (`ciaosh` or `ciao`), an interpreter of scripts written in Ciao (`ciao-shell`), a Ciao (and Prolog) `emacs` mode (which greatly helps the task of developing programs with support for editing, debugging, version/change tracking, etc.), numerous libraries, a powerful program preprocessor (`ciaopp` [BGH99, HPB99, HBPLG99, BLGPH04, HPBLG05], which supports static debugging and optimization from program analysis via source to source program transformation), and an automatic documentation generator (`lpdoc`) [HC97, Her00]. A number of execution visualization tools [CGH93, CH00d, CH00c] are also available.

This manual documents the first four of the tools mentioned above [see Chapter 8 [PART I - The program development environment], page 53], and the Ciao language and libraries. The `ciaopp` and `lpdoc` tools are documented in separate manuals.

The Ciao language [see Chapter 23 [PART II - The Ciao basic language], page 123] has been designed from the ground up to be small, but to also allow extensions and restrictions in a modular way. This allows modular program development with clear program semantics, as well as automatic program manipulation and optimization. Ciao includes a robust module system [CH00a], module-based automatic incremental compilation [CH99b], and modular global program analysis, debugging and optimization [PH99], based on a rich assertion language [see Chapter 36 [PART III - Assertions and auto-documentation], page 187] for declaring (optional) program properties (including types, modes, determinacy, etc.), which can be checked either statically or dynamically. The program analysis, static debugging and optimization tasks related to these assertions are performed by the `ciaopp` preprocessor, as mentioned above. These assertions (together with special comment-style declarations) are also the ones used by the `lpdoc` autodocumenter to generate documentation for programs (the comment-style declarations are documented in the `lpdoc` manual).

Moreover, the modular design allows powerful extensions such as constraint logic programming, multi-paradigm features (higher-order, functional notation, etc.), alternative search and evaluation strategies, concurrency (threads), distributed and parallel execution, foreign language interfaces, restrictions such as working with pure horn clauses [see Chapter 53 [PART IV - Language extensions], page 275], as well as supporting standard ISO-Prolog [see Chapter 108 [PART V - Compatibility], page 491].

Ciao includes a collection of standard and additional libraries and utilities see [see Chapter 120 [PART VI - Data structures and algorithms], page 535, Chapter 142 [PART VII - Standard libraries], page 611, and Chapter 164 [PART VIII - Additional libraries], page 743]. Apart from these, many other tools and libraries, such as language bindings) to several languages and systems are distributed as separate bundles.

2.3 ISO-Prolog compliance and extensibility

One of the innovative features of Ciao is that it has been designed to subsume transparently *ISO-Prolog* (International Standard ISO/IEC 13211-1, PROLOG: Part 1–General Core [DEDC96]), while at the same time extending it in many important ways. The intention is to ensure that all ISO-compliant Prolog programs run correctly under Ciao. At the same time, the Ciao module system (see Chapter 23 [PART II - The Ciao basic language], page 123 and [CH00a] for a discussion of the motivations behind the design) allows selectively avoiding the loading of most ISO-builtins (and changing some other ISO characteristics) when not needed, so that it is possible to work with purer subsets of Prolog and also to build small executables. Also, this module system makes it possible to develop extensions using these purer subsets (or even the full ISO-standard) as a starting point. Using these features, the Ciao distribution includes libraries which significantly extend the language both syntactically and semantically.

In order to aid programmers who wish to write standard-compliant programs, library predicates that correspond to those in the ISO-Prolog standard are marked specially in the manuals, and differences between the Ciao and the prescribed ISO-Prolog behaviours, if any, are commented appropriately. There is also functionality (see ⟨undefined⟩ [ISO-Prolog package], page ⟨undefined⟩) which makes the standard ISO-Prolog predicates available by default in the modules that use traditional module declarations or that load this package explicitly. Thus, ISO-compliance is active transparently for ported programs.

The intention of the Ciao developers is to maintain and continuously improve the compliance of Ciao with the published parts of the ISO standard, with other reasonable extensions of the standard that may be published in the future, and with those features that the Prolog developer community agrees widely on. At the same time, since one of the design objectives of Ciao is to address some shortcomings of previous implementations of Prolog and logic programming in general, we also encourage and work towards making some of the better ideas present in the system reach the standards and also be adopted by other systems.

2.4 About the name of the System

Ciao is short for “Ciao Prolog,” a name which has an interesting (and not unintended) interpretation. Ciao is an interesting word which means both *hello* and *goodbye*. 'Ciao Prolog' intends to be a really good, all-round, freely available ISO-Prolog system which can be used as a classical Prolog, in both academic and industrial environments (and, in particular, to introduce users to Prolog and to constraint and logic programming –the *hello* part). An indeed many programmers use it just that way. But Ciao is also a new-generation, multiparadigm programming language and program development system which goes well beyond Prolog and other classical logic programming languages. And it has the advantage (when compared to other new-generation LP systems) that it does so while keeping full Prolog compatibility when needed.

2.5 Referring to Ciao

If you find Ciao or any of its components useful, we would appreciate very much if you added a reference to this manual in your work. The following is an appropriate BiBTeX entry with the relevant data:

```
@techreport{ciao-reference-manual-1.16-short,
  author = {F.~Bueno and M.~Carro and M.~V.~Hermenegildo
            and P.~L\'{o}pez-Garc\'{\i}a and
            and J.F.~Morales (Eds.)},
  title = {{T}he {C}iao {S}ystem. {R}ef. {M}anual (V1.16)},
  month = {July},
  year = 2017,
  url = {http://ciao-lang.org},
  butype = {techreport},
  note = {Available at \texttt{http://ciao-lang.org}}
}
```

2.6 Acknowledgments

The Ciao system is a joint effort on one hand of some present (Francisco Bueno, Manuel Carro, Manuel Hermenegildo, Pedro López, and Jose F. Morales) and past (Daniel Cabeza, María José García de la Banda, Germán Puebla) main members of the *CLIP group* at the School of Computer Science, *Technical University of Madrid* and at the *IMDEA Software Institute*, and on the other hand of many colleagues and students that have collaborated with us over the years. These names are listed generally in the documentation of each library or part of the system and appear in the *Author index* chapters of the different manuals, as well as the references. In addition, the following is an (inevitably incomplete) list of other colleagues and sources that have contributed to the development of Ciao:

- The *Ciao engine*, compiler, preprocessor, libraries, and documentation, although completely rewritten at this point by the authors above, have their origins in the *&-Prolog parallel Prolog* engine and *parallelizing compiler*, developed by Manuel Hermenegildo, Kevin Greene, Kalyan Muthukumar, and Roger Nasr at *MCC* and later at *UPM*. The &-Prolog engine and low-level (*WAM*) compilers in turn were derived from early versions (0.5 to 0.7) of *SICStus Prolog* [Car88]. SICStus is an excellent, high performance Prolog system, developed by Mats Carlsson and colleagues at the *Swedish Institute of Computer Science* (*SICS*), that every user of Prolog should check out [Swe95, AAF91]. Very special thanks are due to Seif Haridi, Mats Carlsson, and colleagues at SICS for allowing the SICStus 0.5-0.7 components in *&-Prolog* and its successor, Ciao, to be distributed freely. Parts of the parallel abstract machine have been developed in collaboration with Gopal Gupta and Enrico Pontelli (*New Mexico State University*).
- Several aspects of the analyzers in the *Ciao preprocessor* (`ciaopp`) have been developed in collaboration with Peter Stuckey (*Melbourne U.*), Kim Marriott (*Monash U.*), Maurice Bruynooghe, Gerda Janssens, Anne Mulkers, and Veroniek Dumortier (*K.U. Leuven*), and

Saumya Debray (*U. of Arizona*). The assertion system has been developed in collaboration with Jan Maluzynski and Wlodek Drabent (*Linkoping U.*) and Pierre Deransart (*INRIA*). Some parts of the type inference system derive from systems developed by John Gallagher [GdW94] (*Bristol University*) and later adapted to CLP(FD) by Pawel Pietrzak (*Linkoping U.*).

- The constraint solvers for R and Q are derived from the code developed by Christian Holzbauer (*Austrian Research Institute for AI* in Vienna) [Hol94, Hol90, Hol92].
- The Ciao manuals include material from the *DECsystem-10 Prolog User's Manual* by D.L. Bowen (editor), L. Byrd, F.C.N. Pereira, L.M. Pereira, and D.H.D. Warren [BBP81]. They also contain material from the SICStus Prolog user manuals for SICStus versions 0.5-0.7 by Mats Carlsson and Johan Widen [Car88], as well as from the Prolog ISO standard documentation [DEDC96].

If you feel you have contributed to the development of Ciao and we have forgotten to add your name to this list or to the acknowledgements given in the different chapters and indices, please let us know and we will be glad to give proper credits.

2.7 Version/Change Log

Version 1.22 (2022/9/28, 17:41:23 CET)
TBD

Version 1.21 (2022/3/2, 20:34:30 CET)

- Build system:
  - IMPROVED: Partial rewrite of network-based installation (better selection of releases, allow prebuilt docs and binaries).
  - IMPROVED: Tighter integration of `ciao publish` into the builder.
  - IMPROVED: Third-party commands moved to `"advanced"` help.
- Core (compiler/engine, toplevel, libraries):
  - ADDED: Support PowerPC 64-bit in little-endian mode.
  - ADDED: Replaced using_tty/0 with system_extra:istty/1 (specify fd).
  - ADDED: cgoal/1 property (to distinguish from callable/1 ISO pred).
  - ADDED: `binexec` option for automatic spawning of active modules. This produces multi-purpose binaries that can start as either normal processes or active modules.
  - ADDED: Added `ivar/1` meta-property in assertions (expands to `var/1` plus independence from all other vars).
  - ADDED: New rtchecks (run-time assertion checking) code for `det/1`, `semidet/1`, `multidet/1`, `nondet/1` properties.
  - ADDED: Option to run unit tests in the same process.
  - ADDED: Custom headers in foreign interface gluecode (useful for custom type translations).
  - IMPROVED: Faster dependency checks in the compiler (see `itf_sections`). Improves x2 loading time of large executables.
  - IMPROVED: `iso` package renamed `iso_strict`, code adapted.
  - IMPROVED: merged `bf` and `af` search rule translation modules.
  - IMPROVED: Document need for `devenv` for running tests.
  - IMPROVED: Preserve timestamps when engine metadata does not change.
  - IMPROVED: More resilient unit test runner.

  - FIXED: Fixes to termux compilation (Android).
  - FIXED: Cleanup of itf when `opt_suff/1` used.
  - FIXED: Default hostname to `localhost` in active modules.
  - FIXED: Location of relative paths in `reexport`.
  - FIXED: Documentation of timeout library.
  - FIXED: Allow multiple doccomments before condcomp directives.
  - FIXED: Markdown parser issues.
  - FIXED: Fix `|` operator priority to 1105 (ISO DCG draft).
  - FIXED: Bug in `sub_atom/5`.
  - FIXED: Removed bashism in `config-sysdep.sh`.
  - FIXED: Use stderr consistently in unit test output.
- Ciao emacs mode:
  - ADDED: Distribute flycheck and company support for ciao-mode.
  - ADDED: New `ciao-emacs` command (start emacs with "batteries included"). It preinstalls markdown mode and contrib packages.
  - IMPROVED: library for navigating options menus (used in, e.g., CiaoPP): allow cursor navigation, eliminated need for OK/cancel buttons, allow integer values in responses.
  - CHANGED: Change check assertions binding to `C-c V`.
  - FIXED: `=>` is no longer a prompt recognised my ciao-emacs.
  - FIXED: Issue in passing of system-args to ciao process.

Version 1.20 (2021/3/18, 19:33:30 CET)

- Build system:
  - ADDED: Experimental `analyze` build grade for analyzing whole bundles (depends on CiaoPP).
  - FIXED: Fix activation links for `pkgconfig` files in 3rd-party installation.
  - DEPRECATED: Removed support for 3rd-party `bower` package system (all existing uses ported to `npm`).
- Language, compiler, and toplevel:
  - FIXED: Fix a bug that redirected `main/1` calls from the toplevel to the `ciaosh` `main/1` predicate (accidentally via the `user` module).
  - FIXED: Detect wrong arity in `add_goal_trans` directives.
  - IMPROVED: Towards a more modular `native_props.pl`, documentation improvements.
  - IMPROVED: Progress towards merging optim-comp branch for native compilation.
  - IMPROVED: Refactored default language packages (they can be used from modules and toplevels).
  - CHANGED: `note` messages go to user error (like other compiler messages).
- Engine:
  - ADDED: Support for new Apple M1. This port is based on existing support for the aarch64 (ARM64) architecture. NOTE: some executable formats depend on redoing 'codesign' after the executable is linked (this may cause issues when distributing binaries).
- Runtime checks, testing, and debugging:

  - IMPROVED: `mshare/1` internally represented as `mshare/2`, with explicit relevant arguments (for `rtchecks`).
  - IMPROVED: Refactor pieces of embedded debugger.
  - IMPROVED: Note message for modules compiled with the trace or debug packages.
  - FIXED: Do not ignore assertions with empty compats, calls, success, and comps fields (also for `texec`).
  - FIXED: Improve debugging of modules with rtcheck instrumentation.
  - FIXED: Expansion for runtime checks preserve `?- ...` directives.
- Libraries:
  - ADDED: Support for Unicode (UTF8) in source code:
    - Pre-generated character code classes for 0..127
    - Documented code class types and identifier syntax for Unicode source code (see `tokenize.pl`).
    - Very efficient and compact (8KB) code class table (see `unicode_gen.pl` for details).
  - ADDED: Unicode escape \uDDDD and \UDDDDDDDD in strings and atoms.
  - ADDED: Added byte-oriented predicates (see `stream_utils.pl`) and types (`basic_props:bytelist/1`), using them when needed.
  - ADDED: Added string_bytes/2 predicate. This predicate bidirectionally transforms between lists of character codes and lists of bytes (using UTF8 encoding/decoding). It is equivalent to =/2 when at least one of the lists is already a list of ASCII codes (0..127).
  - ADDED: Stronger redirection predicates (`open_std_redirect/3` and `close_std_redirect/1`), which allow the redirection of the standard output/error file descriptors together with the default output stream, user_output, and user_error. Added `system:fd_dup/2` and `system:fd_close/1` predicates to manipulate POSIX file descriptors.
  - ADDED: Added stream_utils:copy_stream/3 predicate (copies bytes from one stream to the other).
  - ADDED: Extended `io_once_port_reify/{3,4}` with better redirections (subset of process channels available for `process_call/3`).
  - ADDED: Parsing of version strings (`version_strings:version_parse/4`).
  - IMPROVED: Heuristics in `write_assertion/{6,7}`, more readable output.
  - IMPROVED: Allow JSON values (not only lists) as top argument for JSON write and parse.
  - IMPROVED: Faster and more reliable sockets predicates. New `socket_sendall/2`, `socket_send_stream/2`, changed `socket_send/3`. `socket_recv_code/3` is replaced by `socket_recv/3` (treating the returned length is encouraged, do not use `socket_recv/2`).
  - IMPROVED: Faster and more robust HTTP libraries (sockets improvements, faster IO). Several bug fixes.
  - FIXED: Fixed bug in markdown parser which confused some predicate heads with items.
  - FIXED: Allow numbers (as constants) in assertion head arguments.

  - FIXED: Fixed bug in `attrdump.pl` introduced when `assoc` replaced `dict`.
  - FIXED: Write blanks before -0.Nan if needed. Fixed some corner cases in parser for 0.Inf, 0.Nan .
  - FIXED: Cleanups in messages_basic:messages/1.
  - FIXED: Allow `use_package(tabling)` in a toplevel.
- ISO and Portability:
  - ADDED: Implemented `at_end_of_stream/0`, `at_end_of_stream/1`. Peek byte in `at_end_of_stream/{0,1}` for improved ISO compatibility.
  - ADDED: Implemented `peek_byte/1`, `peek_byte/2`.
  - ADDED: Added call_det/2 predicate (compatible with `gprolog`).
  - ADDED: Added forall/2 predicate (compatible with `gprolog`).
  - FIXED: callable/1 is an instantiation check.
  - IMPROVED: Added version_data flag as `ciao(Major,Minor,Patch,Extra)`.
  - IMPROVED: Added stream property `type(_)` in `open/4` predicate (for compatibility).
  - ADDED: `--iso-strict` flag in `ciaoc` and `ciaosh` to enable stricter compatibility ISO mode by default. Use with care, switching the flag will not enforce the recompilation of already compiled user files and modules (i.e., .po files).
  - IMPROVED: Additions to the `iso_strict.pl` package (only for code using this package):
    - More compatible version of `absolute_file_name/2` (do not repeat last path component when resolving library paths).
    - Allow stream aliases in most IO operations.
    - Import `keysort/2` and `format/?` predicates by default.
    - Enable `call/N` by default.
    - Enable `runtime_ops` package by default.
- Ciao emacs mode:
  - ADDED: Initial support for `flycheck`, integrating ciaoc, ciaopp, lpdoc, and testing.
  - ADDED: Support for company mode (text completion). Manuals are located dynamically. Completion list obtained using `librowser`.
  - ADDED: Extended Ciao mode (`ciao-emacs-plus.el`) using `flycheck` and `company` extensions.
  - ADDED: `M-x ciao-server-start`, `M-x ciao-server-stop` to start/stop the `ciao-serve` process (serving local HTML documentation and hub for HTTP based interface to active modules).
  - ADDED: Mark and color new `passed`, `failed`, `aborted` message types (for unit tests).
  - IMPROVED: Replace outdated `word-help` by `info-look`.
  - IMPROVED: Better binding for next error, better code highlight, narrow error location.
  - IMPROVED: Allow short location paths (resolved from the elisp side). See `bundle_paths:bundle_extend_path/2` for details on path extension.
  - IMPROVED: Faster font-lock in `ciao-inferior-mode`, only treat keywords.

  - FIXED: Fixed indentation and coloring of `=:=`.
  - FIXED: Fix build errors when compiling using emacs 27.1.
- Unit Tests:
  - ADDED: Allow `opt_suff/1` in unit tests (for alternative source files, e.g., for flycheck).
  - ADDED: New options for handling tests for stdout and stderr in unittest.
  - ADDED: Added support for test filters (see options in `run_tests/4`).
  - ADDED: Using 'passed', 'failed', and 'aborted' message types.
  - ADDED: Test timeout.
  - ADDED: Initial support for integrated regression testing (see `save`, `compare`, etc. actions in `run_tests/3`).
  - IMPROVED: More flexible quering of results and statistics (`get_statistical_summary/2`, `print_statistical_summary/1`, `status(S)` option).
  - IMPROVED: Simpler one-line output per test.
  - IMPROVED: Warnings when predicate fails/throws and there were no failure/exception properties in test assertion.
  - FIXED: Unittest regression does not depend on Ciao root path.

Version 1.19 (2020/3/20, 14:48:45 CET)

Highlights of this release:

- Build system: optional (weak) dependencies in bundle Manifest, out-of-tree builds by default in bundles (sources are no longer polluted with .po/.itf files).
- Language and libraries: a more natural argument order in partial applications, clarified semantics of shared variables in predicate abstractions (and faster implementation), improvements in tabling with constraints (TCLP).
- System: several fixes in small and large integer operations, new algorithm for float to string conversion (based on Ryu, fixing the round-trip property), and thread-safe exception handling mechanism.
- Runtime checks and unit tests: major cleanups, fixes, and improvements in functionality and efficiency.
- Top level: cyclic terms detected by default (`check_cycles` flag no longer needed), faster pretty printing of solutions.
- Ciao emacs mode: 4-space indentation by default. Improvements to syntax coloring. Coloring of info manuals generated by LPdoc. Connection with ciaoserve. Dropped support for xemacs.
- Installation: new instructions for Windows based on WSL and for Android based on Termux.

Detailed list of new features and changes:

- Builder and installation:
  - ADDED: Optional (weak) dependencies in bundle Manifests. Weak dependencies allow bundles with conditional code which depends on the availability of other bundles.
  - ADDED: Allow gitlab aliases in `ciao get`, i.e., `ciao get gitlab.x.y.z/some/path` will recognize (as a heuristic) that we are trying to install a bundle from a Gitlab repository. It should work for any instance of gitlab (as long as there is public access to the repository).

  - ADDED: Exposed `third-party-install` command (e.g., `ciao third-party-install ciao_ppl.ppl` to install `ppl` 3rd-party from `ciao_ppl` bundle).
  - ADDED: Execute `autoreconf -i` when GNU build system is selected and the `configure` file is not available (3rd-party installer).
  - ADDED: Support for `zsh` shell, unified config in `--core:update_shell=[yes|no]`
  - ENHANCED: Improved interactive `ciao-boot.sh` (reordered questions, check deps).
  - EXPERIMENTAL: Support for `--parallel=yes` build option.
  - FIXED: Throw exception if foreign config tool is not found.
  - FIXED: Absolute path in rpath for 3rd-party libs.
  - FIXED: Make sure that `cache/` dir exists before loading bundle manifest hooks.
  - FIXED: `ciao clean-tree` works with non-absolute paths.
  - FIXED: Do not assume `/bin/rm` is there, use `TMPDIR` if defined.
  - FIXED: Make `./ciao-boot.sh clean` work even if `core` is not built.
  - FIXED: Make sure that building `core` prepares the bin grade (`core.ciaobase`).
- Language, compiler, and toplevel:
  - ADDED: Out-of-tree builds, enabled by default. Use `CIAOCCACHE=0` to disable it. The compilation of modules under a bundle produces `.itf` and `.po` files located in the build/cache directory of their corresponding workspace.
  - ENHANCED: Using atomic file writes everywhere in the compiler. Now several processes may compile simultaneously the same code base without corrupting the `.po`/`.itf` compiler output files (there are a few documented bugs in the build scripts that should be fixed in the next commits to allow parallel systems builds).
  - ENHANCED: New internal representation for predicate abstraction, which fixes a potential performance problem due to unnecessary renaming of shared variables.
  - ENHANCED: Pretty printing of solutions from the toplevel is now orders of magnitude faster for some corner cases.
  - CHANGED: Using standard hiord argument order. This changes the argument ordering for predicate abstractions for `hiord` to make it compatible with other systems and languages. The old order was implemented to favor 1st argument indexing, but it can be confusing because of the difference with other languages with higher-order (specially for partial applications).

    Note that was overdue because it was a complicated backwards incompatible change that required many changes in the compiler and libraries (including the assertion language, parametric properties, runtime checks, and parts of the CiaoPP analysis framework).
  - CHANGED: Made the semantics of shared variables in predicate abstractions more strict. Now only the variables specified in `ShVs` for `call((ShVs -> ''(...) :- ...), ...)` will share with the caller's body variables. No other variables will be implicitly shared.
  - CHANGED: Conditional compilation is built-in in the compiler now. This improves the portability of some experimental libraries.

- EXPERIMENTAL: `string_type` package for native strings.
- EXPERIMENTAL: a default behavior was defined for `ciao-serve` so that it provides the available manuals in `index.html` (via the HTTP protocol).
- FIXED: Make partial application work as expected (e.g., `X=append, Y = X([1]), Y([2], Z)`)
- FIXED: Syntactic errors in assertion normalization are now treated correctly by `c_itf.pl`.
- FIXED: Program point assertions (`check/1`,`true/1`,etc.) are not removed by `mexpand.pl`
- FIXED: `load_compilation_module/1` was incorrectly ignoring modules that were already processed (e.g., compiled) but not loaded in the context of `c_itf.pl`.
- ENHANCED: Pretty printing of solutions detects cyclic terms automatically. The `check_cycles` flag is no longer needed and has been removed.
- REMOVED: `check_cycles` flag is no longer needed.

- Engine:
  - ADDED: Support for `aarch64` on Android (using Termux).
  - ADDED: Support for 64-bits in tabling libraries. Trie adapted to allow the use of big numbers.
  - FIXED: using C `-fno-stack-check` option as a workaround for Darwin19/Xcode-11 bug (macOS Catalina)
  - FIXED: Using C `MoveFileEx()` intead of `rename()` in Win32.
  - FIXED: Normalize `c_headers_directory()` in Win32.
  - FIXED: Fix C warnings due to casts of integers of different sizes.
  - FIXED: Using `__builtin_mul_overflow()` for better 32-bit/64-bit portability.
  - FIXED: Fix evaluation of right shift with large numbers.
  - FIXED: Fixed round-trip property in float to string and string to float conversions. The new code is based on the extremely fast Ryu algorithm (see 2018 paper).
  - FIXED: Added `lib(engine)` to `core/Manifest` (this ensures that `engine/` files are copied in global installations).
  - FIXED: Thread-safe and more efficient reimplementation of exceptions.
  - FIXED: Handler for signals that are not intercepted.
  - FIXED: Fix right shift of negative `smallint`. For `Shift` in 0..70 and `V=(-1<<57)` or `V=(-1)`, `X is V>>Shift` produced `X=0` instead of a negative number.
- Runtime checks, testing, and debugging:
  - ADDED: New `timeout/2` property for test assertions (10 min default).
  - ADDED: New `generate_from_calls_n/2` property for test assertions (generate multiple test states from the calls field, 1 by default).
  - ENHANCED: Major cleanups in `rtchecks` package.
  - CHANGED: Default value of `try_sols/2` is 2 (rather than infinite).
  - CHANGED: `unittest` aborts on compilation errors.
  - CHANGED:
    Replaced `num_solutions('>'(N))` by `num_solutions('<'(N))` (due to new hiord).

  - FIXED: rtchecks for exception properties rethrow rtcheck error exceptions.
  - FIXED: Added `@` option also in embedded debugger (see `debug` or `trace` package).
- Libraries:
  - ADDED: Improvements to TCLP: support for 64 bits, interface with the Mod TCLP modular framework, new solver interfaces (difference constraints, CLP(Q), CLP(R)), new constraint solver over lattices (abs_new_constraint), a new framework for incremental evaluation of lattice-based aggregates (`tclp_aggregates`).
  - ADDED: New `system:get_numcores/1`, obtains the number of logical CPU cores.
  - ENHANCED: Updated pretty printing-style formatting of clauses and assertions.
  - CHANGED: Inline foreign code feature moved to `foreign_inliner` package.
  - CHANGED: Update indentation rules in `write:portray_clause/{1,2}`.
  - CHANGED: All code ported to the new hiord argument order and predicate abstraction sharing rules.
  - FIXED: Bug in fastrw due to wrong integer casting (64-bit).
  - FIXED: `C-c` restarts the toplevel only if it was started (for embedded toplevels).
  - FIXED: `bfall` and `afall` search rules compatible with more language extensions.
  - FIXED: Add missing parenthesis for `(,)/2` in `assrt_write` predicates.
  - FIXED: Fix foreign interface C embedding example.
  - FIXED: Missing cuts, `==/2`, and meta_predicate declarations in `assoc.pl`.
  - FIXED: Avoid invalid cross-device link errors in `file_buffer.pl` predicates.
  - FIXED: Missing GC roots in `system:extract_paths/2`.
- Reference manual:
  - ADDED: Installation instructions for Windows based on WSL and for Android based on Termux.
  - ENHANCED: Improved documentation of several libraries (`regexp`, `runtime_control`, read).
  - ENHANCED: Separate language conventions from introduction.
  - FIXED: Document that `call_with_time_limit/{2,3}` behaves as `once/1`.
  - FIXED: Added `classic_predicates.pl` (for classic compatibility package)
- Ciao emacs mode:
  - ENHANCED: Indentation code rewritten (supporting block syntax, better indentation of if-then-else, argument-based indentation for columns).
  - ENHANCED: New syntax coloring code (better multiline coloring of strings and comments, quoted atoms, doccomments blocks, documentation commands, assertion syntax)
  - ENHANCED: Coloring of info manuals generated by LPdoc.
  - ENHANCED: Do not set tty colors for ciao faces (modern terminals look great with the default colors).

  - CHANGED: Using 4-space indentation by default.
  - CHANGED: `C-g` clears compilation error marks.
  - CHANGED: Blanks instead of tabs in ciao-mode.
  - CHANGED: Set default emacs init file to `~/.emacs.d/init.d` (`~/.emacs.el`, `~/.emacs` are still detected if present)
  - EXPERIMENTAL: `M-x ciao-serve` (starts a Ciao server), `M-x ciao-dist` (prepares data for a Ciao service).
  - FIXED: `word-help-extract-index` ignore missing indices.
  - FIXED: use the lpdoc toplevel to generate/view buffer documentation (instead of a shell).
  - FIXED: Modified error location colors for clarity when background is dark.
  - FIXED: Fixing many elisp compilation warnings.
  - REMOVED: Dropped support for xemacs.

Version 1.18 (2018/12/6, 11:25:8 CEST)

- Backward-incompatible changes in this version:
  - Changed the defaults for modules declared with `module/3`. The following predicates and features are no longer included by default in module/3. They should be enabled explicitly with the following packages or modules:
    - `call/N`: `hiord` package.
    - `data`, `concurrent` declarations, `assertz_fact/1`, etc.: `datafacts` package.
    - `dynamic` declarations, `assertz/1`, etc.: `dynamic` package.
    - `set_prolog_flag/2`, etc.: `engine(runtime_control)` (which merges deprecated `engine(prolog_flags)` and `engine(prolog_sys)`).
    - nl/0, nl/1, display/0, open/3, etc.: library(streams) (which reexports stream handling and operations, namely `engine(stream_basic)` and `engine(io_basic)`).
  - Added `noprelude` that prevents loading the prelude (default definitions).
  - The `pure` package now includes a minimum set of control constructs `(,)/2`, `true/0`, `fail/0`.
- Language, compiler, toplevel:
  - Major update of the Ciao manual (basic language, language extensions, Ciao standard library, additional libraries, abstract data types, ISO and compatibility, etc.).
  - Built-in build system and software packaging system (*bundles*) (see documentation for details).
  - Added (optional) `CIAOROOT` and `CIAOPATH` environment variables (replace `CIAOLIB`). `CIAOROOT` points to the root of the Ciao sources rather than the lib directory.
  - New `ciao-env` command to set up the environment for some specific Ciao installations.
  - Fix `MANPATH`,`INFOPATH` in `ciao-env` (trailing `:` was incorrectly removed, it is meaningful and represents default paths).
  - Fixed `ciaosh -e Goal` (accepts any goal), removed `-g` option.
  - DCG `phrase/3` available by default in classic mode (toplevel, user modules, and modules declared with `module/2`).
  - Fixes in runtime check versions of `mshare/1`, `indep/1`, `indep/2`, and `covered/2`.

- Fixed issues with cyclic terms in debugger (when `check_cycles` flag is activated).
- (experimental) `ciao-serve` command to start a Ciao server to serve both HTTP and active module requests.

- Ciao emacs mode:
  - Added
    `M-x ciao-set-ciao-root`, `M-x ciao-set-ciao-path` (see `CIAOROOT` and `CIAOPATH` changes).
  - Improved syntax highlighting.
- Engine:
  - Fixed a bug while freeing sources in `eng_call/{3,4}`.
  - Fixed bug in dynamic/data predicates (uninitialized registers may lead to memory corruption during garbage collection).
  - Added `ciao_root/1`, replaces `ciao_lib_dir/1`.
  - Improved documentation and examples for interfacing with C/C++ (including embedding engines in C/C++ applications).
- Libraries:
  - Fixed bug in tokenizer (dealing with `\^` escape sequences in strings).
  - Refurbished HTTP libraries (separated from pillow, see documentation).
  - Added `library(io_port_reify)` (like `port_reify` but allows IO redirection).
  - Added `filter/3`, `partition/4`, `maplist/N` to `library(hiordlib)`.
  - Added `library(opendoc)` (opens a document with the default OS viewer).
  - Revamped active modules model and implementation (see documentation for details).
  - Renamed `library(file_utils)` to `library(stream_utils)`.
  - Predicates `stream_to_string/{2,3}` replaced by `read_to_end/{2,3}` (which do not close the stream).
  - Added `library(terms_io)` (`terms_to_file/2`, `file_to_terms/2`).
  - (experimental) `library(timeout)` (`call_with_time_limit/{2,3}`).
  - (experimental) package for traits (interfaces).

Version 1.16 (2016/12/31, 11:36:37 CEST)

- Engine:
  - Generating the emulator loop with our own code expansion and emulator generator (emugen).
  - Refactor, clean up, rewrite some engine parts.
  - Reworking custom engine compilation (under `build/` directory). Ciao headers must be included now using `#include <ciao/...>` rather than double quotes.
  - Fix bug in arithmetic shifting operators by 0.
  - Fix `X is (1<<20)*(1<<10)` returned `X=0` (detect multiplication overflows using `__builtin_smul_overflow` to avoid C undefined behaviours, i.e., in clang).
  - Fixes in bignums and float to integer conversion in 64-bits mode.
  - `lib/engine/` merged into `engine/` (no need to separate Prolog and C files).
  - 64-bit port, enabled by default (this was a very large change which required rewriting some parts of the engine).

  - Adding `--trace-instr` engine option (traces instructions, for debugging).
  - Faster implementation of `unify_with_occurs_check/2`.
  - Fix potential overflow in string to number conversion.
  - Better support for `UTF8`.
  - Properly escaping all control characters in quoted atom print (ISO compliance).
  - Fix ending of quoted atom and strings (ISO compliance).
  - Fix `get_char/1` (ISO conformance, past end of file).
  - Fix in treatment of `EOF` in IO predicates (do not assume `EOF == -1`).
  - `CIAOARCH` replaced by `CIAOOS` (e.g., Linux) and `CIAOARCH` (e.g., `i686`). New `ciao_sysconf` command (replaces `ciao_get_arch` script), which accepts the arguments `--os`, `--arch`, and `--osarch`.
- Portability and OS support:
  - Using `clang` as default C compiler in MacOS.
  - Identify `MINGW64_NT` as Win32 (which is commonly accepted as a generic OS name which does not necessarily mean 32-bits).
  - Drop support for IRIX and SunOS4.
  - Improved support for NetBSD (NetBSD 7), FreeBSD.
  - Support for Raspberry Pi.
  - (experimental) support for MINGW32 and MINGW64 (and MSYS2) builds (for Windows).
  - (experimental) support for EMSCRIPTEN as compilation target.
- Language, compiler, toplevel:
  - Conditional compilation library `library(condcomp)` enabled by default.
  - Deprecated `alias(a(b(...)))` as a module specifier name (using the more compatible `alias(a/b/...)` instead).
  - Fix exit status (returns 1) for toplevel and executables on abort, e.g., due to uncaught exceptions or unexpected failure.
  - New `ciaoc_sdyn` tool to help in the distribution of standalone executables with foreign code (collects all required dynamic libraries).
  - Starting work on new build system.
  - (experimental) syntax extension for infix dot `(A.B)` (see `set_prolog_flag(read_infix_dot, on)`).
  - (experimental) syntax extension for string data type (see `set_prolog_flag(read_string_data_type, on)`).
- Libraries:
  - Fix `system:touch/1`, implemented through C `utime()`.
  - Fix buffer overflow in `absolute_file_name/?` with `nul/NUL` in Win32 (it is a reserved name).
  - Fix bug in check for cyclic terms, implemented faster C (low-level) version.
  - Fix `get_tmp_dir/1` so that it always produces a normalized path, with no trailing `/`, and considering `TMPDIR` on POSIX systems.
  - Better use of `current_executable/1` implementation (macOS: `_NSGetExecutablePath()`, Linux: `readlink` on `/proc/self/exe`, Windows: `GetModuleFileName()` with `hModule = NULL`).
  - Added support for `phrase/2` and `phrase/3` in DCGs (in `dcg_phrase` package).

   - Added `library(global_vars)`, backtrackable global variables.
   - Added `library(datetime)`, manipulate date and time in different formats.
   - Added `library(clpfd)`, new CLP(FD) implementation.
   - Added `library(glob)`, support *glob* patterns, filenames with wildcard characters.
   - Added `library(pathnames)`, predicates for file path name manipulation, compatible with common semantics in other languages.
   - Added `library(port_reify)`, metacalls which reify the `exit` port so that it can be delayed.
   - Added `library(process)`, portable high-level interface for child process creation, supporting stream redirection, background processes, signals, etc.
   - Added `library(text_template)`, text-based templates.
   - Added `library(http_get)`, retrieve files via HTTP/HTTPs/FTP protocol.
   - Added `system:get_home/1`, `system:find_executable/2`.
   - Added `system:extract_paths/2`, split atom containing a colon-separated path list as individual paths.
   - Deprecated `exec/?` from `library(system)`.
   - Deprecated `system:get_exec_dir/1`, can be replaced by `current_executable/1` and `path_dirname/2`.
   - (experimental) `library(indexer)`, a package that extends first-argument indexing.
   - (experimental) heap limits exceptions (`set_heap_limit/1`).
   - (experimental) `library(stream_wait)`, wait for input to be available, with timeouts.
- Ciao emacs mode:
   - Cleanups, refactoring into smaller individual components (highlighting, interaction with Ciao, etc.).
   - `M-x ciao-grep*` emacs command (search over all code).
- Foreign interface:
   - Fix exception throw from C builtins during shallow backtracking.
   - Allow exception throwing using arbitrary terms.
   - Foreign interface types corresponding to different fixed-width C types (`c_int`,`c_size`,`c_uint8`, etc.).

Version 1.15 (2011/7/8, 11:48:1 CEST)
New development version (Jose Morales)

Version 1.14#2 (2011/8/12, 18:14:31 CEST)
Merging r13606 (trunk) into 1.14. This backports an optimization for DARWIN platforms (Jose Morales)

Version 1.14#1 (2011/8/10, 18:17:10 CEST)
Merging r13583 through r13586 (trunk) into 1.14. This fixes problems in the Windows version of Ciao (Edison Mera, Jose Morales)

Version 1.14 (2011/7/8, 10:51:55 CEST)
It has been a long while since declaring the last major version (basically since moving to subversion after 1.10/1.12), so quite a bit is included in this release. Here is the (longish) summary:

- Extensions to functional notation:
  - Introduced `fsyntax` package (just functional syntax). (Daniel Cabeza)
  - Added support to define on the fly a return argument different from the default one (e.g. `~functor(~,f,2)`). (Daniel Cabeza)
  - Use of '`:- function(defined(true)).`' so that the defined function does not need to be preceded by `~` in the return expression of a functional clause. (Daniel Cabeza)
  - Functional notation: added to documentation to reflect more of the FLOPS paper text and explanations. Added new functional syntax examples: arrays, combination with constraints, using func notation for properties, lazy evaluation, etc. (Manuel Hermenegildo)
  - Added functional abstractions to `fsyntax` and correct handling of predicate abstractions (the functions in the body where expanded outside the abstraction). (Jose Morales)
  - Improved translation of functions. In particular, old translation could lose last call optimization for functions with body or with conditional expressions. Furthermore, the translation avoids now some superfluous intermediate unifications. To be studied more involved optimizations. (Daniel Cabeza, Jose Morales).
  - More superfluous unifications taken out from translated code, in cases where a goal `~f(X) = /Term/` appears in the body. (Daniel Cabeza)
  - Added `library/argnames_fsyntax.pl`: Package to be able to use `$~/2` as an operator. (Daniel Cabeza)
  - Added a new example for lazy evaluation, saving memory using lazy instead of eager evaluation. (Amadeo Casas)
- Improvements to signals and exceptions:
  - Distinguished between exceptions and signals. Exceptions are thrown and caught (using `throw/1` and `catch/3`). Signals are sent and intercepted (using `send_signal/1` and `intercept/3`). (Jose Morales, Remy Haemmerle)
  - Back-port of the (improved) low-level exception handling from `optim_comp` branch. (Jose Morales)
  - Fixed `intercept/3` bug, with caused the toplevel to not properly handle exceptions after one was handled and displayed (bug reported by Samir Genaim on 04 Dec 05, in ciao mailing list, subject "`ciao top-level : exception handling`"). Updated documentation. (Daniel Cabeza)
  - `intercept/3` does not leave pending choice points if the called goal is deterministic (the same optimization that was done for `catch/3`). (Jose Morales)
- New/improved libraries:
  - New `assoc` library to represent association tables. (Manuel Carro, Pablo Chico)
  - New `regexp` library to handle regular expressions. (Manuel Carro, Pablo Chico)
  - Fixed bug in string_to_number that affected ASCII to floating point number conversions (`number_codes/2` and bytecode read). (Jose Morales)
  - `system.pl`: Added predicates `copy_file/2` and `copy_file/3`. Added predicates `get_uid/1`, `get_gid/1`, `get_pwnam/1`, `get_grnam/1` implemented natively to get default user and groups of the current process. (Edison Mera)

- Added library for mutable variables. (Remy Haemmerle)
- Added package for block declarations (experimental). (Remy Haemmerle)
- Ported CHR as a Ciao package (experimental). (Tom Schrijvers)
- Debugged and improved performance of the CHR library port. (Remy Haemmerle)
- `contrib/math`: A library with several math functions that depends on the GNU Scientific Library (GSL). (Edison Mera)
- `io_aux.pl`: Added `messages/1` predicate. Required to facilitate printing of compact messages (compatible with emacs). (Edison Mera)
- Added library `hrtimer.pl` that allow us to measure the time using the higest resolution timer available in the current system. (Edison Mera)
- Global logical (backtrackable) variables (experimental). (Jose Morales)
- New dynamic handling (`dynamic_clauses` package). Not yet documented. (Daniel Cabeza)
- Moved `=` from `iso_misc` to `term_basic`. (Daniel Cabeza)
- `lib/lists.pl`: Added predicate `sequence_to_list/2`. (Daniel Cabeza)
- `lib/lists.pl`: Codification of `subordlist/2` improved. Solutions are given in other order. (Daniel Cabeza)
- `lib/filenames.pl`: Added `file_directory_base_name/3`. (Daniel Cabeza)
- `library/symlink_locks.pl`: preliminary library to make locks a la emacs. (Daniel Cabeza)
- `lib/between.pl`: Bug in `between/3` fixed: when the low bound was a float, an smaller integer was generated. (Daniel Cabeza)
- Fixed bug related to implication operator `->` in Fuzzy Prolog (Claudio Vaucheret)
- `contrib/gendot`: Generator of dot files, for drawing graphs using the dot tool. (Claudio Ochoa)
- Addded `zeromq` library (bindings for the Zero Message Queue (ZeroMQ, 0MQ) cross-platform messaging middleware) (Dragan Ivanovic)
- Minor documentation changes in `javall` library (Jesus Correas)
- Fix a bug in calculator `pl2java` example (Jesus Correas)
- `lib/aggregates.pl`: Deleted duplicated clauses of `findnsols/4`, detected by Pawel. (Daniel Cabeza)
- Added library to transform between color spaces (HSL and HVS) (experimental). (Jose Morales)
- Added module qualification in DCGs. (Remy Haemmerle, Jose Morales)
- `prolog_sys:predicate_property/2` behaves similar to other Prolog systems (thanks to Paulo Moura for reporting this bug). (Jose Morales)
- Added DHT library (implementation of distributed hash table) (Arsen Kostenko)
- Adding property `intervals/2` in `native_props.pl` (for intervals information) (Luthfi Darmawan)
- Added code to call polynomial root finding of GSL (Luthfi Darmawan)
- Some improvements (not total, but easy to complete) to error messages given by errhandle.pl . Also, some of the errors in `sockets_c.c` are now proper exceptions instead of faults. (Manuel Carro)

  - `sockets` library: added a library (`nsl`) needed for Solaris (Manuel Carro)
  - Driver, utilities, and benchmarking programs from the ECRC suite. These are aimed at testing some well-defined characteristics of a Prolog system. (Manuel Carro)
  - `library/getopts.pl`: A module to get command-line options and values. Intended to be used by Ciao executables. (Manuel Carro)
- Improved ISO compliance:
  - Ported the Prolog ISO conformance testing.
  - Fixed read of files containing single "`%`" char (reported by Ulrich Neumerkel). (Jose Morales)
  - Added exceptions in `=../2`. (Remy Haemmerle)
  - Added exceptions in arithmetic predicates. (Remy Haemmerle)
  - Arithmetics integer functions throw exceptions when used with floats. (Remy Haemmerle)
  - Added exceptions for resource errors. (Remy Haemmerle)
- Improvements to constraint solvers:
  - Improved CLPQ documentation. (Manuel Hermenegildo)
  - Added clp_meta/1 and clp_entailed/1 to the clpq and clpr packages (Samir Genaim):
    - `clp_meta/1`: meta-programming with clp constraints, e.g, `clp_meta([A.>.B,B.>.1])`.
    - `clp_entailed/1`: checks if the store entails specific cnstraints, e.g, `clp_entailed([A.>.B])` succeeds if the current store entailes `A.>.B`, otherwise fails.
  - Exported the simplex predicates from CLP(Q,R). (Samir Genaim)
- Other language extensions:
  - Added new `bf/bfall` package. It allows running all predicates in a given module in breadth-first mode without changing the syntax of the clauses (i.e., no `<-` needed). Meant basically for experimentation and, specially, teaching pure logic programming. (Manuel Hermenegildo)
  - Added `afall` package in the same line as `bf/bfall` (very useful!). (Manuel Hermenegildo)
  - Improved documentation of `bf` and `af` packages. (Manuel Hermenegildo)
  - Added partial commons-style dialect support, including dialect flag. (Manuel Hermenegildo)
  - `yap_compat` and `commons_compat` compatibility packages (for Yap and Prolog Commons dialects). (Jose Morales)
  - `argnames` package: enhanced to allow argument name resolution at run-time. (Jose Morales)
  - A package for conditional compilation of code (`:- use_package(condcomp)`). (Jose Morales)
- Extensions for parallelism (And-Prolog):
  - Low-level support for andprolog library has been taken out of the engine and moved to `library/apll` in a similar way as the sockets library. We are planning to reduce the size of the actual engine further, by taking some components out of engine, such as locks, in future releases. (Amadeo Casas)

    - Improved support for deterministic parallel goals, including some bug fixes. (Amadeo Casas)
    - Goal stack definition added to the engine. (Amadeo Casas)
    - And-parallel code and the definition of goal stacks in the engine are now wrapped with conditionals (via `AND_PARALLEL_EXECUTION` variable), to avoid the machinery necessary to run programs in parallel affects in any case the sequential execution. (Amadeo Casas)
    - Stack expansion supported when more than one agent is present in the execution of parallel deterministic programs. This feature is still in experimental. Support for stack expansion in nondeterministic benchmarks will be added in a future release. (Amadeo Casas)
    - Support for stack unwinding in deterministic parallel programs, via `metachoice/metacut`. However, garbage collection in parallel programs is still unsupported. We are planning to include support for it in a future release. (Amadeo Casas)
    - Backward execution of nondeterministic parallel goals made via events, without speculation and continuation join. (Amadeo Casas)
    - Improved agents support. New primitives included that aim at increasing the flexibility of creation and management of agents. (Amadeo Casas)
    - Agents synchronization is done now by using locks, instead of using `assertz/retract`, to improve efficiency in the execution of parallel programs. (Amadeo Casas)
    - Optimized version of `call/1` to invoke deterministic goals in parallel has been added (`call_handler_det/1`). (Amadeo Casas)
    - Optimization: locks/`new_atom` only created when the goal is stolen by other process, and not when this is pushed on to the `goal_stack`. (Amadeo Casas)
    - Integration with the new annotation algorithms supported by CiaoPP, both with and without preservation of the order of the solutions. (Amadeo Casas)
    - New set of examples added to the `andprolog` library. (Amadeo Casas)
    - Several bug fixes to remove some cases in execution of parallel code in which races could appear. (Amadeo Casas)
    - `andprolog_rt:&` by `par_rt:&` have been moved to `native_builtin` (Amadeo Casas)
    - `indep/1` and `indep/2` have been moved to `native_props`, as `ground/1`, `var/1`, etc. (Amadeo Casas)
    - Added assertions to the `library/apll` and `library/andprolog` libraries. (Amadeo Casas)
    - Removed clauses in `pretty_print` for the `&>/2` and `<&/1` operators. (Amadeo Casas)
    - Shorter code for `<& / 1` and `<&! / 1` (Manuel Carro)
    - Trying to solve some problems when resetting WAM pointers (Manuel Carro)
    - Better code to clean the stacks (Manuel Carro)
- Improvements to foreign (C language) interface:
    - Better support for cygwin and handling of dll libraries in Windows. Now usage of external dll libraries are supported in Windows under cygwin. (Edison Mera)

    - Improvements to documentation of foreign interface (examples). (Manuel Hermenegildo)
    - Allow reentrant calls from Prolog to C and then from C to Prolog. (Jose Morales)
    - Fix bug that prevented `ciaoc -c MODULE` from generating dynamic `.so` libraries files. (Jose Morales)
    - Fix bug that prevented `ciaoc MODULE && rm MODULE && ciaoc MODULE` from emitting correct executables (previously, dynamic `.so` libraries files where ignored in executable recompilations when only the main file was missing). (Jose Morales)
- Run-Time Checking and Unit Tests:
    - Added support to perfom run-time checking of assertions and predicates outside `ciaopp` (see the documentation for more details). In addition to those already available, the new properties that can be run-time checked are: `exception/1`, `exception/2`, `no_exception/1`, `no_exception/2`, `user_output/2`, `solutions/2`, `num_solutions/2`, `no_signal/1`, `no_signal/2`, `signal/1`, `signal/2`, `signals/2`, `throws/2`. See library `assertions/native_props.pl` (Edison Mera)
    - Added support for testing via the `unittest` library. Documentation available at `library(unittest/unittest)`. (Edison Mera)
- Profiling:
    - Improved profiler, now it is cost center-based and works together with the run-time checking machinery in order to also validate execution time-related properties. (Edison Mera)
    - A tool for automatic bottleneck detection has been developed, which is able to point at the predicates responsible of lack of performance in a program. (Edison Mera)
    - Improved profiler documentation. (Manuel Hermenegildo)
- Debugger enhancements:
    - Added the flag `check_cycles` to control whether the debugger takes care of cyclic terms while displaying goals. The rationale is that to check for cyclic terms may lead to very high response times when having big terms. By default the flag is in off, which implies that a cyclic term in the execution could cause infinite loops (but otherwise the debugger is much more speedy). (Daniel Cabeza)
    - Show the variable names instead of underscores with numbers. Added option `v` to show the variables list. Added `v <N>` option, where `N` is the `Name` of the variable you like to watch (experimental). (Edison Mera)
    - Distinguish between program variables and compiler-introduced variables. Show variables modified in the current goal. (Edison Mera)
    - `debug_mode` does not leave useless choicepoints (Jose Morales)
- Emacs mode:
    - Made ciao mode NOT ask by default if one wants to set up version control when first saving a file. This makes more sense if using other version control systems and probably in any case (several users had asked for this). There is a global customizable variable (which appears in the LPdoc area) which can be set to revert to the old behaviour. Updated the manual accordingly. (Manuel Hermenegildo)
    - Added possibility of chosing which emacs Ciao should use during compilation, by LPdoc, etc. Previously only a default emacs was used which is not

always the right thing, specially, e.g., in Mac OS X, where the latest/right emacs may not even be in the paths. Other minor typos etc. (Manuel Hermenegildo)

- Moved the version control menu entries to the LPdoc menu. (Manuel Hermenegildo)
- Updated highlighting for new functional syntax, unit tests, and all other new features. (Manuel Hermenegildo)
- Completed CiaoPP-java environment (menus, buttons, etc.) and automated loading when visiting Java files (still through hand modification of .emacs). CiaoPP help (e.g., for properties) now also available in Java mode. (Manuel Hermenegildo)
- Changes to graphical interface to adapt better to current functionality of CiaoPP option browser. Also some minor aesthetic changes. (Manuel Hermenegildo)
- Various changes and fixes to adapt to emacs-22/23 lisp. In particular, fixed cursor error in emacs 23 in Ciao shell (from Emilio Gallego). Also fixed prompt in ciaopp and LPdoc buffers for emacs 23. (Manuel Hermenegildo)
- Unified several versions of the Ciao emacs mode (including the one with the experimental toolbar in xemacs) that had diverged. Sorely needed to be able to make progress without duplication. (Manuel Hermenegildo)
- New version of ciao.el supporting tool bar in xemacs and also, and perhaps more importantly, in newer emacsen (>= 22), where it previously did not work either. New icons with opaque background for xemacs tool bar. (Manuel Hermenegildo)
- Using `key-description` instead of a combination of `text-char-description` and `string-to-char`. This fixes a bug in the Ciao Emacs Mode when running in emacs 23, that shows wrong descriptions for `M-...` key bindings. The new code runs correctly in emacs 21 and 22. (Jose Morales)
- Coloring strings before functional calls and `0'` characters (strings like `"~w"` were colored incorrectly) (Jose Morales)
- `@begin{verbatim}` and `@include` colored as LPdoc commands only inside LPdoc comments. (Jose Morales)
- Fixed colors for dark backgrounds (workaround to avoid a bug in emacs) (Jose Morales)
- Added an automatic indenter (contrib/plindent) and formatting tool, under emacs you can invoque it using the keyword `C-c I` in the current buffer containing your prolog source. (Edison Mera)

- Packaging and distribution:
  - User-friendly, binary installers for several systems are now generated regularly and automatically: Ubuntu/Debian, Fedora/RedHat, Windows (XP, Vista, 7) and MacOSX. (Edison Mera, Remy Haemmerle)
- Improvements in Ciao toplevel:
  - Introduced `check_cycles prolog_flag` which controls whether the toplevel handles or not cyclic terms. Flag is set to false by default (cycles not detected and handled) in order to speed up responses. (Daniel Cabeza)
  - Modified `valid_solution/2` so that it asks no question when there are no pending choice points and the `prompt_alternatives_no_bindings` prolog flag is on. (Jose Morales)

- Now 'Y' can be used as well as 'y' to accept a solution of a query. (Daniel Cabeza)
- Added newline before `true` when displaying empty solutions. (Jose Morales)
- Multifile declarations of packages used by the toplevel were not properly handled. Fixed. (Daniel Cabeza)
- Fixed bug in output of bindings when current output changed.
- Changes so that including files in the toplevel (or loading packages) does not invoke an expansion of the ending end_of_file. This makes sense because the toplevel code is never completed, and thus no cleanup code of translations is invoked. (Daniel Cabeza)

- Compiler enhancements and bug fixes:
  - Added a command line option to `ciaoc` for generating code with runtime checks. (Daniel Cabeza)
  - Now the compiler reads assertions by default (when using the assertion package), and verifies their syntax. (Edison Mera)
  - Added option `-w` to `ciaoc` compiler to generate the WAM code of the specified prolog files. (Edison Mera)
  - Fixed bug in exemaker: now when `main/0` and `main/1` exists, `main/0` is always the program entry (before in modules either could be). (Daniel Cabeza)
  - Fixed bug: when compiling a file, if an imported file had no itf and it used the redefining declaration, the declaration was forgotten between the reading of the imported file (to get its interface) and its later compilation. By now those declarations are never forgotten, but perhaps it could be done better. (Daniel Cabeza)
  - The unloading of files kept some data related to them, which caused in some cases errors or warnings regarding module redefinitions. Now this is fixed. (Daniel Cabeza)
  - Undefined predicate warnings also for predicate calls qualified with current module (bug detected by Pawel Pietrzak). (Daniel Cabeza)
  - Fixed bug `debugger_include` (that is, now a change in a file included from a module which is debugged is detected when the module is reloaded). (Daniel Cabeza)
  - Fixed `a(B) :- _=B, b, c(B)` bug in compilation of unification. (Jose Morales)

- Improving general support for language extensions:
  - Every package starts with '`:- package(...)`' declaration now. This allows a clear distinction between packages, modules, and files that are just included; all of them using the same `.pl` extension. (Jose Morales)
  - Added priority in syntax translations. Users are not required to know the details of translations in order to use them (experimental: the the correct order for all the Ciao packages is still not fixed) (Jose Morales)
  - Now the initialization of sentence translations is done in the translation package, when they are added. In this way, previous active translations cannot affect the initialization of new translations, and initializations are not started each time a new sentence translation is added. Additionally, now the initialization of sentence translations in the toplevel is done (there was a bug). (Daniel Cabeza)

  - Added `addterm(Meta)` meta-data specification for the implementation of the changes to provide a correct `clause/2` predicate. (Daniel Cabeza)
  - Generalized `addmodule` meta-data specification to `addmodule(Meta)`, `addmodule` is now an alias for `addmodule(?)`. Needed for the implementation of the changes to provide a correct `clause/2` predicate. (Daniel Cabeza)
- Improvements to system assertions:
  - Added regtype `basic_props:num_code/1` and more assertions to `basic_props.pl` (German Puebla)
  - Added trust assertion for `atomic_basic:number_codes/2` in order to have more accurate analysis info (first argument a number and second argument is a list of num_codes) (German Puebla)
  - Added some more binding insensitivity assertions in `basic_props.pl` (German Puebla)
  - Added the `basic_props:filter/2` property which is used at the global control level in order to guarantee termination. (German Puebla)
  - Added `equiv` assertion for `basiccontrol:fail/0` (German Puebla)
  - Modified eval assertion so that partial evaluation does not loop with ill-typed, semi-instantiated calls to `is/2` (this problem was reported some time ago) (German Puebla)
  - Replaced `true` assertions for arithmetic predicates with `trust` assertions (`arithmetic.pl`). (German Puebla)
  - Added assertions for `term_basic:'='/2` (the *not unification*) (German Puebla)
  - Added assertions for `lists:nth/3` predicate and `lists:reverse/3`. (German Puebla)
  - Changed calls to `atom/1` to `atm/1` in `c_itf_props:moddesc/1` (it is a regular type) (Jesus Correas)
  - `formulae:assert_body_type/1` switched to `prop`, it is not a `regtype`. (Jesus Correas)
  - Added assertions to `atom_concat/2`. (Jesus Correas)
  - Added some assertions to `dec10_io`, `lists`, `strings` libraries. (Jesus Correas)
  - Removed `check` from pred and success froom many library assertions. (Jesus Correas)
  - Fixed a problem when reading multiple disjunction in assertions (`library/formulae.pl` and `lib/assertions/assrt_write.pl`). (Pawel Pietrzak)
  - Added/improved assertions in several modules under `lib/` (Pawel Pietrzak)
- Engine enhancements:
  - Added support for Ciao compilation in `ppc64` architecture. (Manuel Carro)
  - `sun4v` added in `ciao_get_arch`. (Amadeo Casas)
  - Solved compilation issue in Sparc. (Manuel Carro, Amadeo Casas)
  - Support for 64 bits Intel processor (in 32-bit compatibility mode). (Manuel Carro)
  - Switched the default memory manager from linear to the binary tree version (which improves management of small memory blocks). (Remy Haemmerle)

  - Using `mmap` in Linux/i86, Linux/Sparc and Mac OS X (Manuel Carro)
  - A rename of the macro `REGISTER` to `CIAO_REGISTER`. There have been reports of the macro name clashing with an equally-named one in third-party packages (namely, the PPL library). (Manuel Carro)
  - A set of macros `CIAO_REG_n` (`n` currently goes from `1` to `4`, but it can be enlarged) to force the GCC compiler to store a variable in a register. This includes assignments of hardware registers for `n = 1` to `3`, in seemingly ascending order of effectiveness. See coments in registers.h (Manuel Carro)
  - An assignement of (local) variables to be definitely stored in registers for some (not all) functions in the engine – notably `wam.c`. These were decided making profiling of C code to find out bottlenecks and many test runs with different assignments of C variables to registers. (Manuel Carro)
  - Changed symbol name to avoid clashes with other third-party packages (such as minisat). (Manuel Carro)
  - Fixed a memory alignment problem (for RISC architectures where words must be word-aligned, like Sparc). (Jose Morales)
  - Unifying some internal names (towards merge with optim_comp experimental branch). (Jose Morales)
- Attributed variables:
  - Attributes of variables are correctly displayed in the toplevel even if they contain cyclic terms. Equations added in order to define cyclic terms in attributes are output after the attributes, and do use always new variable names (doing otherwise was very involved). (Daniel Cabeza)
  - `lib/attrdump.pl`: The library now works for infinite (cyclic) terms. (Daniel Cabeza)
  - Changed multifile predicate `dump/3` to `dump_constraints/3`. (Daniel Cabeza)
  - Added `copy_extract_attr_nc/3` which is a faster version of `copy_extract_attr/3` but does not handle cyclic terms properly. (Daniel Cabeza)
  - Added `term_basic:copy_term_nat/2` to copy a term taking out attributes. (Daniel Cabeza)
- Documentation:
  - Added `deprecated/1`. (Manuel Hermenegildo)
  - Improvements to documentation of `rtchecks` and tests. (Manuel Hermenegildo)
  - Many updates to manuals: dates, copyrights, etc. Some text updates also. (Manuel Hermenegildo)
  - Fixed all manual generation errors reported by LPdoc (still a number of warnings and notes left). (Manuel Hermenegildo)
  - Adding some structure (minor) to all manuals (Ciao, LPdoc, CiaoPP) using new LPdoc `doc_structure/1`. (Jose Morales)
- Ciao Website:
  - Redesigned the Ciao website. It is generated again through LPdoc, but with new approach. (Jose Morales)

Version 1.13 (2005/7/3, 19:5:53 CEST)
New development version after 1.12. (Jose Morales)

Version 1.12 (2005/7/3, 18:50:50 CEST)
Temporary version before transition to SVN. (Jose Morales)

Version 1.11#1 (2003/4/4, 18:30:31 CEST)
New development version to begin the builtin modularization (Jose Morales)

Version 1.10#8 (2007/1/28, 18:1:27 CEST)
Backports and bug fixes to stable 1.10:

- Changes to make Ciao 1.10 compile with the latest GCC releases.
- Imported from `CiaoDE/branches/CiaoDE-memory_management-20051016`, changes from revisions 4909 to 4910: Changes to make Ciao issue a better message at startup if the allocated memory does not fall within the limits precomputed at compile time (plus some code tidying).
- Port of revisions 5415, 5426, 5431, 5438, 5546, 5547 applied to Ciao 1.13 to Ciao 1.10 in order to make it use `mmap()` when possible and to make it compile on newer Linux kernels. Tested in Ubuntu, Fedora (with older kernel) and MacOSX.
- Configuration files for DARWIN (ppc) and 64-bit platforms (Intel and Sparc, both in 32-bit compatibility mode).
- Force the creation of the module containing the foreign interface compilation options before they are needed.

Version 1.10 (2004/7/29, 16:12:3 CEST)

- Classical prolog mode as default behavior.
- Emacs-based environment improved.
  - Improved emacs inferior (interaction) mode for Ciao and CiaoPP.
  - Xemacs compatibility improved (thanks to A. Rigo).
  - New icons and modifications in the environment for the preprocessor.
  - Icons now installed in a separate dir.
  - Compatibility with newer versions of `Cygwin`.
  - Changes to programming environment:
    - Double-click startup of programming environment.
    - Reorganized menus: help and customization grouped in separate menus.
    - Error location extended.
    - Automatic/Manual location of errors produced when running Ciao tools now customizable.
    - Presentation of CiaoPP preprocessor output improved.
  - Faces and coloring improved:
    - Faces for syntax-based highlighting more customizable.
    - Syntax-based coloring greatly improved. Literal-level assertions also correctly colored now.
    - Syntax-based coloring now also working on ASCII terminals (for newer versions of emacs).
    - Listing user-defined directives allowed to be colored in special face.
    - Syntax errors now colored also in inferior buffers.
    - Customizable faces now appear in the documentation.
    - Added new tool bar button (and binding) to refontify block/buffer.
    - Error marks now cleared automatically also when generating docs.
    - Added some fixes to hooks in lpdoc buffer.

- Bug fixes in compiler.
  - Replication of clauses in some cases (thanks to S. Craig).
- Improvements related to supported platforms
  - Compilation and installation in different palatforms have been improved.
  - New Mac OS X kernels supported.
- Improvement and bugs fixes in the engine:
  - Got rid of several segmentation violation problems.
  - Number of significant decimal digits to be printed now computed accurately.
  - Added support to test conversion of a Ciao integer into a machine int.
  - Unbound length atoms now always working.
  - C interface .h files reachable through a more standard location (thanks to R. Bagnara).
  - Compatibility with newer versions of gcc.
- New libraries and utilities added to the system:
  - Factsdb: facts defined in external files can now be automatically cached on-demand.
  - Symfnames: File aliasing to internal streams added.
- New libraries added (in beta state):
  - fd: clp(FD)
  - xml_path: XML querying and transformation to Prolog.
  - xdr_handle: XDR schema to HTML forms utility.
  - ddlist: Two-way traversal list library.
  - gnuplot: Interface to GnuPlot.
  - time_analyzer: Execution time profiling.
- Some libraries greatly improved:
  - Interface to Tcl/Tk very improved.
    - Corrected many bugs in both interaction Prolog to Tcl/Tk and viceversa.
    - Execution of Prolog goals from TclTk revamped.
    - Treatment of Tcl events corrected.
    - Predicate `tcl_eval/3` now allows the execution of Tcl procedures running multiple Prolog goals.
    - Documentation heavily reworked.
    - Fixed unification of prolog goals run from the Tcl side.
  - Pillow library improved in many senses.
    - HTTP media type parameter values returned are always strings now, not atoms.
    - Changed verbatim() pillow term so that newlines are translated to
.
    - Changed management of cookies so that special characters in values are correctly handled.
    - Added predicate `url_query_values/2`, reversible. Predicate `url_query/2` now obsolete.
    - Now attribute values in tags are escaped to handle values which have double quotes.

- Improved `get_form_input/1` and `url_query/2` so that names of parameters having unusual characters are always correctly handled.
- Fixed bug in tokenizer regarding non-terminated single or multiple-line comments. When the last line of a file has a single-line comment and does not end in a newline, it is accepted as correct. When an open-comment /* sequence is not terminated in a file, a syntax error exception is thrown.
- Other libraries improved:
  - Added native_props to assertions package and included `nonground/1`.
  - In atom2terms, changed interpretation of double quoted strings so that they are not parsed to terms.
  - Control on exceptions improved.
  - Added `native/1,2` to basic_props.
  - Davinci error processing improved.
  - Foreign predicates are now automatically declared as implementation-defined.
  - In lists, added `cross_product/2` to compute the cartesian product of a list of lists. Also added `delete_non_ground/3`, enabling deletion of nonground terms from a list.
  - In llists added `transpose/2` and changed `append/2` implementation with a much more efficient code.
  - The make library has been improved.
  - In persdb, added `pretractall_fact/1` and `retractall_fact/1` as persdb native capabilities.
  - Improved behavior with user environment from persdb.
  - In persdb, added support for `persistent_dir/4`, which includes arguments to specify permission modes for persistent directory and files.
  - Some minor updates in persdb_sql.
  - Added treatment of operators and module:pred calls to pretty-printer.
  - Updated report of read of syntax errors.
  - File locking capabilities included in `open/3`.
  - Several improvements in library system.
  - New input/output facilities added to sockets.
  - Added `most_specific_generalization/3` and `most_general_instance/3` to terms_check.
  - Added `sort_dict/2` to library vndict.
  - The xref library now treats also empty references.
- Miscellaneous updates:
  - Extended documentation in libraries actmods, arrays, foreign_interface, javall, persdb_mysql, prolog_sys, old_database, and terms_vars.

Version 1.9 (2002/5/16, 23:17:34 CEST)

New development version after stable 1.8p0 (MCL, DCG)

Version 1.8 (2002/5/16, 21:20:27 CEST)

- Improvements related to supported platforms:
  - Support for Mac OS X 10.1, based on the Darwin kernel.
  - Initial support for compilation on Linux for Power PC (contributed by Paulo Moura).

    - Workaround for incorrect C compilation while using newer (> 2.95) gcc compilers.
    - .bat files generated in Windows.
- Changes in compiler behavior and user interface:
    - Corrected a bug which caused wrong code generation in some cases.
    - Changed execution of initialization directives. Now the initialization of a module/file never runs before the initializations of the modules from which the module/file imports (excluding circular dependences).
    - The engine is more intelligent when looking for an engine to execute bytecode; this caters for a variety of situations when setting explicitly the CIAOLIB environment variable.
    - Fixed bugs in the toplevel: behaviour of `module:main` calls and initialization of a module (now happens after related modules are loaded).
    - Layout char not needed any more to end Prolog files.
    - Syntax errors now disable .itf creation, so that they show next time the code is used without change.
    - Redefinition warnings now issued only when an unqualified call is seen.
    - Context menu in Windows can now load a file into the toplevel.
    - Updated Windows installation in order to run CGI executables under Windows: a new information item is added to the registry.
    - Added new directories found in recent Linux distributions to INFOPATH.
    - Emacs-based environment and debugger improved:
        - Errors located immediataly after code loading.
        - Improved ciao-check-types-modes (preprocessor progress now visible).
        - Fixed loading regions repeatedly (no more predicate redefinition warnings).
        - Added entries in `ciaopp` menu to set verbosity of output.
        - Fixed some additional xemacs compatibility issues (related to searches).
        - Errors reported by inferior processes are now explored in forward order (i.e., the first error rewported is the first one highlighted). Improved tracking of errors.
        - Specific tool bar now available, with icons for main fuctions (works from emacs 21.1 on). Also, other minor adaptations for working with emacs 21.1 and later.
        - Debugger faces are now locally defined (and better customization). This also improves comtability with xemacs (which has different faces).
        - Direct access to a common use of the preprocessor (checking modes/types and locating errors) from toolbar.
        - Inferior modes for Ciao and CiaoPP improved: contextual help turned on by default.
        - Fixes to set-query. Also, previous query now appears in prompt.
        - Improved behaviour of stored query.
        - Improved behaviour of recentering, finding errors, etc.
        - Wait for prompt has better termination characteristics.
        - Added new interactive entry points (M-x): ciao, prolog, ciaopp.
        - Better tracking of last inferior buffer used.

    - Miscellanous bugs removed; some colors changed to adapt to different Emacs versions.
    - Fixed some remaining incompatibilities with xemacs.
    - `:- doc` now also supported and highlighted.
    - Eliminated need for calendar.el
    - Added some missing library directives to fontlock list, organized this better.
- New libraries added to the system:
  - hiord: new library which needs to be loaded in order to use higher-order call/N and P(X) syntax. Improved model for predicate abstractions.
  - fuzzy: allows representing fuzzy information in the form or Prolog rules.
  - use_url: allows loading a module remotely by using a WWW address of the module source code
  - andorra: alternative search method where goals which become deterministic at run time are executed before others.
  - iterative deepening (id): alternative search method which makes a depth-first search until a predetermined depth is reached. Complete but in general cheaper than breadth first.
  - det_hook: allows making actions when a deterministic situation is reached.
  - ProVRML: read VRML code and translate it into Prolog terms, and the other way around.
  - io_alias_redirection: change where stdin/stdout/stderr point to from within Ciao programs.
  - tcl_tk: an interface to Tcl/Tk programs.
  - tcl_tk_obj: object-based interface to Tcl/Tk graphical objects.
  - CiaoPP: options to interface with the CiaoPP Prolog preprocessor.
- Some libraries greatly improved:
  - WebDB: utilities to create WWW-based database interfaces.
  - Improved java interface implementation (this forced renaming some interface primitives).
  - User-transparent persistent predicate database revamped:
    - Implemented passerta_fact/1 (asserta_fact/1).
    - Now it is never necessary to explicitly call init_persdb, a call to initialize_db is only needed after dynamically defining facts of persistent_dir/2. Thus, pcurrent_fact/1 predicate eliminated.
    - Facts of persistent predicates included in the program code are now included in the persistent database when it is created. They are ignored in successive executions.
    - Files where persistent predicates reside are now created inside a directory named as the module where the persistent predicates are defined, and are named as F_A* for predicate F/A.
    - Now there are two packages: persdb and 'persdb/ll' (for low level). In the first, the standard builtins asserta_fact/1, assertz_fact/1, and retract_fact/1 are replaced by new versions which handle persistent data predicates, behaving as usual for normal data predicates. In the second package, predicates with names starting with 'p' are defined, so that there is not overhead in calling the standard builtins.

        - Needed declarations for persistent_dir/2 are now included in the packages.
    - SQL now works with mysql.
    - system: expanded to contain more predicates which act as interface to the underlying system / operating system.
- Other libraries improved:
    - xref: creates cross-references among Prolog files.
    - concurrency: new predicates to create new concurrent predicates on-the-fly.
    - sockets: bugs corrected.
    - objects: concurrent facts now properly recognized.
    - fast read/write: bugs corrected.
    - Added 'webbased' protocol for active modules: publication of active module address can now be made through WWW.
    - Predicates in library(dynmods) moved to library(compiler).
    - Expansion and meta predicates improved.
    - Pretty printing.
    - Assertion processing.
    - Module-qualified function calls expansion improved.
    - Module expansion calls goal expansion even at runtime.
- Updates to builtins (there are a few more; these are the most relevant):
    - Added a prolog_flag to retrieve the version and patch.
    - current_predicate/1 in library(dynamic) now enumerates non-engine modules, prolog_sys:current_predicate/2 no longer exists.
    - exec/* bug fixed.
    - srandom/1 bug fixed.
- Updates for C interface:
    - Fixed bugs in already existing code.
    - Added support for creation and traversing of Prolog data structures from C predicates.
    - Added support for raising Prolog exceptions from C predicates.
    - Preliminary support for calling Prolog from C.
- Miscellaneous updates:
    - Installation made more robust.
    - Some pending documentation added.
    - 'ciao' script now adds (locally) to path the place where it has been installed, so that other programs can be located without being explicitly in the $PATH.
    - Loading programs is somewhat faster now.
    - Some improvement in printing path names in Windows.

Version 1.7 (2000/7/12, 19:1:20 CEST)

Development version following even 1.6 distribution.

Version 1.6 (2000/7/12, 18:55:50 CEST)

- Source-level debugger in emacs, breakpts.
- Emacs environment improved, added menus for Ciaopp and LPDoc.
- Debugger embeddable in executables.

- Standalone executables available for Unix-like operating systems.
- Many improvements to emacs interface.
- Menu-based interface to autodocumenter.
- Threads now available in Win32.
- Many improvements to threads.
- Modular clp(R) / clp(Q).
- Libraries implementing And-fair breadth-first and iterative deepening included.
- Improved syntax for predicate abstractions.
- Library of higher-order list predicates.
- Better code expansion facilities (macros).
- New delay predicates (when/2).
- Compressed object code/executables on demand.
- The size of atoms is now unbound.
- Fast creation of new unique atoms.
- Number of clauses/predicates essentially unbound.
- Delayed goals with freeze restored.
- Faster compilation and startup.
- Much faster fast write/read.
- Improved documentation.
- Other new libraries.
- Improved installation/deinstallation on all platforms.
- Many improvements to autodocumenter.
- Many bug fixes in libraries and engine.

Version 1.5 (1999/11/29, 16:16:23 MEST)
Development version following even 1.4 distribution.

Version 1.4 (1999/11/27, 19:0:0 MEST)
- Documentation greatly improved.
- Automatic (re)compilation of foreign files.
- Concurrency primitives revamped; restored &Prolog-like multiengine capability.
- Windows installation and overall operation greatly improved.
- New version of O'Ciao class/object library, with improved performance.
- Added support for "predicate abstractions" in call/N.
- Implemented reexportation through reexport declarations.
- Changed precedence of importations, last one is now higher.
- Modules can now implicitly export all predicates.
- Many minor bugs fixed.

Version 1.3 (1999/6/16, 17:5:58 MEST)
Development version following even 1.2 distribution.

Version 1.2 (1999/6/14, 16:54:55 MEST)
Temporary version distributed locally for extensive testing of reexportation and other 1.3 features.

Version 1.1 (1999/6/4, 13:30:37 MEST)
Development version following even 1.0 distribution.

Version 1.0 (1999/6/4, 13:27:42 MEST)

- Added Tcl/Tk interface library to distribution.
- Added push_prolog_flag/2 and pop_prolog_flag/1 declarations/builtins.
- Filename processing in Windows improved.
- Added redefining/1 declaration to avoid redefining warnings.
- Changed syntax/1 declaration to use_package/1.
- Added add_clause_trans/1 declaration.
- Changed format of .itf files such that a '+' stands for all the standard imports from engine, which are included in c_itf source internally (from engine(builtin_exports)). Further changes in itf data handling, so that once an .itf file is read in a session, the file is cached and next time it is needed no access to the file system is required.
- Many bugs fixed.

Version 0.9 (1999/3/10, 17:3:49 CET)
- Test version before 1.0 release. Many bugs fixed.

Version 0.8 (1998/10/27, 13:12:36 MET)
- Changed compiler so that only one pass is done, eliminated `.dep` files.
- New concurrency primitives.
- Changed assertion comment operator to #.
- Implemented higher-order with call/N.
- Integrated SQL-interface to external databases with persistent predicate concept.
- First implementation of object oriented programming package.
- Some bugs fixed.

Version 0.7 (1998/9/15, 12:12:33 MEST)
- Improved debugger capabilities and made easier to use.
- Simplified assertion format.
- New arithmetic functions added, which complete all ISO functions.
- Some bugs fixed.

Version 0.6 (1998/7/16, 21:12:7 MET DST)
- Defining other path aliases (in addition to 'library') which can be loaded dynamically in executables is now possible.
- Added the posibility to define multifile predicates in the shell.
- Added the posibility to define dynamic predicates dynamically.
- Added addmodule meta-argument type.
- Implemented persistent data predicates.
- New version of PiLLoW WWW library (XML, templates, etc.).
- Ported active modules from “distributed Ciao” (independent development version of Ciao).
- Implemented lazy loading in executables.
- Modularized engine(builtin).
- Some bugs fixed.

Version 0.5 (1998/3/23)
- First Windows version.
- Integrated debugger in toplevel.
- Implemented DCG's as (Ciao-style) expansions.

- Builtins renamed to match ISO-Prolog.
- Made ISO the default syntax/package.

Version 0.4 (1998/2/24)
- First version with the new Ciao emacs mode.
- Full integration of concurrent engine and compiler/library.
- Added new_declaration/1 directive.
- Added modular syntax enhancements.
- Shell script interpreter separated from toplevel shell.
- Added new compilation warnings.

Version 0.3 (1997/8/20)
- Ciao builtins modularized.
- New prolog flags can be defined by libraries.
- Standalone comand-line compiler available, with automatic "make".
- Added assertions and regular types.
- First version using the automatic documentation generator.

Version 0.2 (1997/4/16)
- First module system implemented.
- Implemented exceptions using catch/3 and throw/1.
- Added functional & record syntax.
- Added modular sentence, term, and goal translations.
- Implemented attributed variables.
- First CLPQ/CLPR implementation.
- Added the posibility of linking external .so files.
- Changes in syntax to allow `P(X)` and `"string"||L`.
- Changed to be closer to ISO-Prolog.
- Implemented Prolog shell scripts.
- Implemented data predicates.

Version 0.1 (1997/2/13)

First fully integrated, standalone Ciao distribution. Based on integrating into an evolution of the &-Prolog engine/libraries/preprocessor [Her86, HG91] many functionalities from several previous independent development versions of Ciao [HC93, HC94, HCC95, Bue95, ciao-comp-dist-tr-deliv ?, HBdlBP95, HBC96, CHV96, ciao-novascience ?].

3 Getting started

Author(s): The Ciao Development Team.

This part guides you through some very basic first steps with Ciao.

We begin with the installation instructions in Chapter 4 [Installing Ciao from the source distribution], page 39. The following sections assume that Ciao is already installed correctly on your system. It is a good idea to perform the tests in ⟨undefined⟩ [Checking for correct installation], page ⟨undefined⟩ to check that Ciao is installed correctly (these are the same tests that you are instructed to do during installation, so you can obviously skip them if you have done them already at that time). If any of these tests do not succeed please review the installation instructions.

We continue with a description of the basics of Chapter 5 [Using Ciao from the command line], page 47. We strongly recommend reading also Chapter 6 [An introduction to the Ciao emacs environment], page 49 for the basics on using Ciao under `emacs`, which is a much simpler and much more powerful way of developing Ciao programs, and has the advantage of offering an almost identical environment across different operating systems.

For any problem during the installation or use of Ciao, please refer to Chapter 7 [Troubleshooting], page 51 for common issues and reporting bugs.

4 Installing Ciao from the source distribution

Author(s): Manuel Carro, Daniel Cabeza, Jose F. Morales, Manuel Hermenegildo.

This describes the installation procedure for the Ciao system, including libraries and manuals, from a *source* distribution. For more specific details, please read also the `INSTALLATION` file that comes with each component of Ciao. However, in most cases following this summary suffices.

4.1 Requirements

A minimal build requires:

- A computing device running a supported operating system (Linux, BSD, macOS, Windows, Android).
- A supported architecture (`amd64`, `x86`, `arm`).
- A relatively modern C compiler (GCC or clang).
- A POSIX-like environment.

Although Ciao implements its own build system, the GNU implementation of the make command is (still) used internally. If any of the installation steps stop right away with `make` error messages, you probably need to install `gmake`.

Installing **Emacs** (`https://www.gnu.org/software/emacs/`) is highly recommended: the Ciao distribution includes a very powerful *application development environment* which is based on `emacs` and which enables, e.g., syntax coloring, source code formatting, embedded top-level, source-level debugging, context-sensitive on-line help, etc.

In any case, it is easy to use Ciao from a terminal and edit code with any editor of your choice. The top level includes some minimal editing facilities when rlwrap (`https://github.com/hanslub42/rlwrap`) is available.

Finally, documentation generation in PDF format requires TeX and ImageMagick.

4.1.1 Installing dependencies

Depending on the operating system and package management tool, you may need to install the following dependencies (using `sudo` or as administrator or `root` user):

- Debian/Ubuntu:

```
$ apt-get install build-essential
# (optional) for 32 bits compatibility mode (x86)
$ apt-get install gcc-multilib libc6-i386 libc6-dev-i386 g++-multilib
# (optional) for emacs-based IDE
$ apt-get install emacs
# (optional) for line edition from the terminal
$ apt-get install rlwrap
# (optional) for generating documentation in PDF format
$ apt-get install texlive texinfo imagemagick
```

  (valid for both `x86` and `arm` architectures)

- Fedora:

```
$ dnf install gcc make which kernel-headers kernel-devel emacs
# (optional) for emacs-based IDE
$ dnf install emacs
# (optional) for 32 bits compatibility mode (x86)
$ dnf install glibc-devel.i686 glibc-devel libstdc++-devel.i686
# (optional) for line edition from the terminal
```

```
$ dnf install rlwrap
# (optional) for generating documentation in PDF format
$ dnf install texlive texinfo texinfo-tex ImageMagick
```

  (use `yum` instead of `dnf` above in older versions of Fedora)

- Arch Linux:

  ```
  # Optional, upgrade if needed
  $ pacman -Syu
  # Dependencies for build and development environment
  # (base-devel: includes gcc, make, which)
  $ pacman -S base-devel linux-headers
  # (optional) for 32 bits compatibility mode (x86)
  # NOTE: Remember to enable multilib (https://wiki.archlinux.org/index.php/Multilib)
  $ pacman -S lib32-glibc lib32-libstdc++5
  # (optional) for emacs-based IDE
  $ pacman -S emacs
  # (optional) for line edition from the terminal
  $ pacman -S rlwrap
  # (optional) for generating documentation in PDF format
  $ pacman -S texlive-core texinfo imagemagick
  ```

- macOS:
  - Install command line tools for Xcode (from the App store)
  - Install emacs and ImageMagick (and texinfo if needed, but recent versions of macOS include it). A software management tool like homebrew or macports is recommended, e.g. (as root/sudo):

    ```
    $ brew install emacs-mac imagemagick
    ```

  - Install TexLive. If using homebrew we recommend installing the MacTeX distribution, available from: `https://www.tug.org/mactex`.
  - Install the following packages:

    ```
    # (optional) for line edition from the terminal
    $ brew install rlwrap
    ```

- FreeBSD:
  - Install GCC or clang and the following packages:

    ```
    $ pkg install gmake
    # (optional) for emacs-based IDE
    $ pkg install emacs ImageMagick expat
    # NOTE: use emacs-nox and ImageMagick-nox if X11 support is not needed
    # (optional) for line edition from the terminal
    $ pkg install rlwrap
    # (optional) for generating documentation in PDF format
    $ pkg install texinfo texlive-full
    ```

- NetBSD:
  - Install GCC or clang and the following packages:

    ```
    $ pkgin install gmake
    # (optional) for emacs-based IDE
    $ pkgin install emacs ImageMagick expat
    # (optional) for line edition from the terminal
    $ pkgin install rlwrap
    # (optional) for generating documentation in PDF format
    $ pkgin install tex-texinfo texlive-collection-latex
    ```

- Windows (using Windows Subsystem or Linux):
  - Install WSL (`https://docs.microsoft.com/en-us/windows/wsl/install-win10`)
  - Install some Linux distribution (e.g., Ubuntu) from the Windows Store.
  - Open a `bash` terminal and install the dependencies for the selected distribution (see points above), e.g., Debian/Ubuntu.
- Android (using the Termux Linux environment):
  - Install Termux (`https://termux.com/`)
  - Install the following packages:

```
$ pkg install clang make
# (optional) for emacs-based IDE
$ pkg install emacs
# (optional) for line edition from the terminal
$ pkg install rlwrap
```

Additional dependencies:

Some advanced libraries and components require an additional set of software packages:

```
# Debian/Ubuntu:
$ apt-get install g++ libgsl0-dev libgsl0ldbl
$ apt-get install default-jdk ant ant-optional

# Fedora:
$ yum install gsl gsl-devel ant gcc-c++
```

To install the Java JDK on Fedora, please visit Sun Java website (`http://java.sun.com/javase/downloads/index.jsp`) and follow the installation instructions there.

4.2 Network-based installation

Ciao can be installed from the network using a `curl` pipe from `https://ciao-lang.org/boot` to `sh` in a terminal. This is probably the fastest/easiest way to install Ciao from sources.

The network installer is delivered as an HTTPs redirection to `https://raw.githubusercontent.com/ciao-lang/ciao/master/ciao-boot.sh` (which contains the latest version of the `ciao-boot.sh` script). The bootstrap script recognizes that it is being called without sources automatically. Examples:

```
# Interactive installation
curl https://ciao-lang.org/boot -sSfL | sh
# Minimal local installation with prebuilt binaries
curl https://ciao-lang.org/boot -sSfL | sh -s -- --prebuilt-bin local-install
# Full environment
curl https://ciao-lang.org/boot -sSfL | sh -s -- get devenv
# Show network installer help
curl https://ciao-lang.org/boot -sSfL | sh -s -- --help
```

4.3 Quick installation from source

The source code for the latest stable and development versions of Ciao is available from `https://ciao-lang.org`.

Installation of Ciao from source is driven by the `ciao-boot.sh` script (`ciao-boot.bat` in Windows), which bootstraps and invokes the Ciao build system.

Note: It is recommended that you read the full installation instructions. However, in most cases following this summary suffices:

1. Type `./ciao-boot.sh configure`. This command will attempt to detect the configuration options for your system automatically and select reasonable defaults.

   Options can be configured interactively using `./ciao-boot.sh configure --interactive` or passed explicitly as arguments from the command-line (see `./ciao-boot.sh help` for more information). In particular, make sure the location of the `emacs` executable is set correctly.
2. Type `./ciao-boot.sh build`. This will build executables, compile libraries, and generate the documentation.
3. Type `./ciao-boot.sh install`. This will install everything in the specified directories.

Alternatively, `ciao-boot.sh` provides shorthands to perform configure, build, and install in a single command:

1. Type `./ciao-boot.sh local-install`. This will do a user-local installation (that will be accessible just for your user).
2. Type `./ciao-boot.sh global-install`. This will do a system-wide installation (e.g., as administrator or `root` user).

The system will include appropriate code at the end of your startup scripts. This will make the documentation and executables accessible, make sure the correct mode is set when opening Ciao source files in `emacs`, etc.

If you need more advanced control of the configuration please read the following sections.

4.4 Full installation instructions

1. **Select installation options:** From the directory where the sources are stored, run:

   ```
   ./ciao-boot.sh configure
   ```

   It will perform a default configuration, where the system will be configured to run from the sources directly, and configured in the user's home directory (recommended for Ciao developers or users without admin rights).

   The option `--instype=global` will prepare Ciao to be installed as the system administrator (`root`) in a standard directory available for all users in the machine (e.g., `/usr/local`).

   For 32-bit builds in 64-bit architectures use the `--core:m32=yes` configuration flag.

   In case you want to install elsewhere, or change any of the installation options, you can use a customized configuration procedure (see `./ciao-boot.sh help` for more information). The meaning of some important options is as follows:

   - `--prefix`: prefix for the default values of installation directories (`<prefix>/bin`, `<prefix>/lib`, etc).
   - `--bindir`: directory where the Ciao commands will be installed. For example, if `--bindir` is set to `/usr/local/bin`, then the Ciao compiler (`ciaoc`) will be stored at `/usr/local/bin/ciaoc`. Actually, it will be a link to `ciaoc-`*VersionNumber*. This applies also to other executables below and is done so that several versions of Ciao can coexist on the same machine. Note that the *version installed latest* will be the one started by default when typing `ciao`, `ciaoc`, etc.
   - The Ciao installation procedure will create a new subdirectory `ciao/<vers>` denoted `INSTALL_CIAOROOT` below `--prefix`. The `<vers>` directory indicate the Ciao system version. It allows having several Ciao versions installed simultaneously.
   - `--mandir`: directory where the manuals in `man` format will be installed.
   - `--infodir`: directory where the manuals in `info` format will be installed.

2. **Build Ciao:** At the ciao top level directory type `./ciao-boot.sh build`.
3. **Install Ciao:** To install Ciao in the directories selected in the configuration script during step 2 above, type `./ciao-boot.sh install`. This will:
   - Install the executables of the Ciao program development tools (i.e., the general driver/top-level `ciao`, the standalone compiler `ciaoc`, the script interpreter `ciao-shell`, miscellaneous utilities, etc.) in the selected binary directory. In order to use these tools, make sure that the `PATH` environment variable contains such path.
   - Install the Ciao libraries under `INSTALL_CIAOROOT` (these will be automatically found).
   - Install the Ciao manuals in several formats (such as GNU `info`, `html`, `pdf`, etc.) under `INSTALL_CIAOROOT` and other documentation-specific paths. In order for these manuals to be found when typing `M-x info` within `emacs`, or by the standalone `info` and `man` commands, the `MANPATH` and `INFOPATH` environment variables of users both need to contain the path specified at `--mandir` and `--infodir`. Documentation in other formats can be accesed with the `ciao doc` command.
   - Install under `INSTALL_CIAOROOT` the Ciao interface with GNU `emacs` (which provides an interactive interface to the Ciao program development tools, as well as some other auxiliary files).
4. **Set up user environments:** The installation process will modify the startup files automatically by default. If you disable this option in the configuration or you wish to switch between different installations, it would be necessary to modify the startup files manually.

   The installation process generates a `ciao-env` command to update the environments for `sh`-like and `csh`like shells. It also generates a `INSTALL_CIAOROOT/ciao-mode-init` (for emacs) with appropriate definitions which will take care of all needed environment variable definitions and emacs mode setup.
   - For users a *csh-compatible shell* (`csh`, `tcsh`, ...), add to `~/.cshrc`:

     ```
     if ( -x <prefix>/bin/ciao-env ) then
        eval `<prefix>/bin/ciao-env --csh`
     endif
     ```
   - For users of an *sh-compatible shell* (`sh`, `bash`, ...), the installer will add to `~/.bashrc` the next lines:

     ```
     if [ -x <prefix>/bin/ciao-env ]; then
        eval "$(<prefix>/bin/ciao-env --sh)"
     fi
     ```
   - For users of an `zsh` shell, the same lines will be added to `~/.zshrc`.

     This will set up things so that the Ciao executables are found and you can access the Ciao system manuals using the `info` command. Note that, depending on your shell, *you may have to log out and back in* for the changes to take effect.
   - Also, if you use `emacs` (highly recommended) the install will add the next line to your `~/.emacs` file:

     ```
     (load-file "INSTALL_CIAOROOT/ciao-site-file.el")
     (if (file-exists-p "INSTALL_CIAOROOT/ciao-site-file.el")
       (load-file "INSTALL_CIAOROOT/ciao-site-file.el")
     )
     ```

   If you are installing Ciao globally in a multi-user machine, make sure that you instruct all users that to do the same. If you are the system administrator, the previous steps can be done once and for all, and globally for all users by including the lines above in the central startup scripts (e.g., in Linux `/etc/bashrc`, `/etc/csh.login`, `/etc/csh.cshrc`, `/etc/skel`, `/usr/share/emacs/.../lisp/site-init.pl`, etc.).

5. **Check installation / read documentation:** You may now want to check your installation (see ⟨undefined⟩ [Checking for correct installation], page ⟨undefined⟩) and read the documentation. There are special "getting started" sections at the beginning of the manual.

If you have any problems you may want to check Chapter 7 [Troubleshooting], page 51.

4.5 Uninstalling

The system can be *uninstalled* by typing `./ciao-boot.sh uninstall` in the top directory. To ensure that this process works, configuration should have not changed since installation, so that the same directories are cleaned.

4.6 Checking for correct installation

If everything has gone well, several applications and tools should be available to a normal user. Try the following while logged in as a *normal user* (important in order to check that permissions are set up correctly):

- Typing `ciao` (or `ciaosh`) should start the typical Prolog-style top-level shell.
- In the top-level shell, Ciao library modules should load correctly. Type for example `use_module(library(dec10_io))` –you should get back a prompt with no errors reported.
- To exit the top level shell, type `halt.` as usual, or ⟨^D⟩.
- Typing `ciaoc` should produce the help message from the Ciao standalone compiler.
- Typing `ciao-shell` should produce a message saying that no code was found. This is a Ciao application which can be used to write scripts written in Ciao, i.e., files which do not need any explicit compilation to be run.

Also, the following documentation-related actions should work:

- If the `info` program is installed, typing `info` should produce a list of manuals which *should include Ciao manual(s) in a separate area* (you may need to log out and back in so that your shell variables are reinitialized for this to work).
- Typing `ciao doc alldocs` should open a series of Ciao-related manuals in a WWW browser.
- Typing `man ciao` should produce a man page with some very basic general information on Ciao (and pointing to the on-line manuals).
- The documentation directory should contain the manual also in the other formats such as `pdf`.

Finally, if `emacs` is installed, after starting it (typing `emacs`) the following should work:

- Typing ⟨^H⟩ ⟨i⟩ (or in the menus `Help->Manuals->Browse Manuals with Info`) should open a list of manuals in info format in which the Ciao manual(s) should appear.
- When opening a Ciao file, i.e., a file with `.pl` or `.pls` ending, using ⟨^X⟩⟨^F⟩`filename` (or using the menus) the code should appear highlighted according to syntax (e.g., comments in red), and `Ciao/Prolog` menus should appear in the menu bar on top of the `emacs` window.
- Loading the file using the `Ciao/Prolog` menu (or typing ⟨^C⟩ ⟨l⟩) should start in another emacs buffer the Ciao toplevel shell and load the file. You should now be able to switch the the toplevel shell and make queries from within `emacs`.

Note: when using `emacs` it is *very convenient* to swap the locations of the (normally not very useful) ⟨Caps Lock⟩ key and the (very useful in `emacs`) ⟨Ctrl⟩ key on the keyboard. How to do this is explained in the `emacs` frequently asked questions FAQs (see the `emacs` download instructions for their location).

4.7 Cleaning up the source directory

After (global) installation, the source directory can be cleaned up using `./ciao-boot.sh realclean`, leaves the distribution is its original form, throwing away any intermediate files (as well as any unneeded files left behind by the Ciao developers), while still allowing recompilation.

4.8 Environment variables used by Ciao executables

The executables generated by the Ciao compiler (including the ciao development tools themselves) locate automatically where the Ciao engine and libraries have been installed, since those paths are stored as defaults in the engine and compiler at installation time. Thus, there is no need for setting any environment variables in order to *run* Ciao executables (on a single architecture – see ⟨undefined⟩ [Multiarchitecture installation], page ⟨undefined⟩ for running on multiple architectures).

However, the default paths can be overridden by using the environment variable `CIAOROOT`, which tell the Ciao executables where to look for engines and system libraries.

4.9 Multiarchitecture installation

In order to perform a multi-architecture installation, it is possible to repeat successively the build and installation process for several architectures from different machines sharing part of their filesystem.

The Ciao build and installation process maintains separate name spaces for platform- and architecture-dependant binaries (like the engine binary, or `.so` or `.dll` gluecode files for the foreign interface). Portable Ciao applications (i.e., except self-contained standalone executables) can run on several machines with different architectures without any need for recompiling, provided the Ciao engine and libraries are correctly installed.

5 Using Ciao from the command line

Author(s): Manuel Hermenegildo.

5.1 Starting/exiting the top-level shell

The basic methods for starting/exiting the top-level shell have been discussed before. If upon typing `ciao` you get a "command not found" error or you get a longer message from Ciao before starting, it means that either Ciao was not installed correctly or you environment variables are not set up properly. Follow the instructions on the message printed by Ciao or refer to the installation instructions regarding user-setup for details.

5.2 Getting help

The basic methods for accessing the manual on-line have also been discussed before. Use the table of contents and the indices of *predicates*, *libraries*, *concepts*, etc. to find what you are looking for. Context-sensitive help is available within the `emacs` environment (see below).

5.3 Compiling and running programs

Once the shell is started, you can compile and execute modules inside the interactive top-level shell in the standard way. E.g., type `use_module(`*file*`).`, `use_module(library(`*file*`)).` for library modules, `ensure_loaded(`*file*`).` for files which are not modules, and `use_package(`*file*`).` for library packages (these are syntactic/semantic packages that extend the Ciao language in many different ways). Note that the use of `compile/1` and `consult/1` is discouraged in Ciao.

For example, you may want to type `use_package(iso_strict)` to ensure Ciao has loaded all the ISO builtins (whether this is done by default or not depends on your `.ciaorc` file). Do not worry about any "module already in executable" messages –these are normal and simply mean that a certain module is already pre-loaded in the top-level shell. At this point, typing `write(hello).` should work.

Note that some predicates that may be built-ins in typical Prolog implementations are available through libraries in Ciao. This facilitates making small executables.

To change the working directory to, say, the `examples` directory from the Ciao root directory, first do:

```
?- use_module(library(system)).
```

(loading the `system` library makes a number of system-related predicates such as `cd/1` accessible) and then:

```
?- cd('examples').
```

For more information see Chapter 9 [The interactive top-level shell], page 55.

5.4 Generating executables

Executables can be generated from the top-level shell (using `make_exec/2`) or using the standalone compiler (`ciaoc`). To be able to make an executable, the file should define the predicate `main/1` (or `main/0`), which will be called upon startup (see the corresponding manual section for details). In its simplest use, given a top-level *foo*`.pl` file for an application, the compilation process produces an executable `foo`, automatically detecting which other files used by `foo.pl` need recompilation.

For example, within the `examples` directory, you can type:

```
?- make_exec(hw,_).
```

which should produce an executable. Typing `hw` in a shell (or double-clicking on the icon from a graphical window) should execute it.

For more information see Chapter 9 [The interactive top-level shell], page 55 and Chapter 13 [The standalone command-line compiler], page 81.

5.5 Running Ciao scripts

Ciao allows writing scripts. These are files containing Ciao source but which get executed without having to explicitly compile them (in the same way as, e.g., `.bat` files or programs in scripting languages). As an example, you can run the file `hw` in the `examples` directory of the Ciao distribution and look at the source with an editor. You can try changing the `Hello world` message and running the program again (no need to recompile!).

As you can see, the file should define the predicate `main/1` (not `main/0`), which will be called upon startup. The two header lines are necessary in Unix in. In Windows you can leave them in or you can take them out, but you need to rename the script to `hw.pls`. Leaving the lines in has the advantage that the script will also work in Unix without any change.

For more information see Chapter 14 [The script interpreter], page 87.

5.6 The Ciao initialization file

The Ciao toplevel can be made to execute upon startup a number of commands (such as, e.g., loading certain files or setting certain Ciao flags) contained in an initialization file. This file should be called `.ciaorc` and placed in your *home* directory (e.g., `~`, the same in which the `.emacs` file is put). You may need to set the environment variable `HOME` to the path of this directory for the Ciao toplevel shell to be able to locate this file on startup.

6 An introduction to the Ciao emacs environment

Author(s): Manuel Hermenegildo.

While it is easy to use Ciao with any editor of your choice, using it within the `emacs` editor/program development system is highly recommended: Ciao includes an `emacs` *mode* which provides a very complete *application development environment* which greatly simplifies many program development tasks. See Chapter 19 [Using Ciao inside GNU emacs], page 97 for details on the capabilities of `ciao/ emacs` combination.

If the (freely available) `emacs` editor/environment is not installed in your system, we highly recommend that you also install it at this point (there are instructions for where to find `emacs` and how to install it in the Ciao installation instructions). After having done this you can try for example the following things:

- A few basic things:
  - Typing ⟨^H⟩ ⟨i⟩ (or in the menus `Help->Manuals->Browse Manuals with Info`) should open a list of manuals in info format in which the Ciao manual(s) should appear.
  - When opening a Ciao file, i.e., a file with `.pl` or `.pls` ending, using ⟨^X⟩⟨^F⟩`filename` (or using the menus) the code should appear highlighted according to syntax (e.g., comments in red), and `Ciao/Prolog` menus should appear in the menu bar on top of the `emacs` window.
  - Loading the file using the `Ciao/Prolog` menu (or typing ⟨^C⟩ ⟨l⟩) should start in another emacs buffer the Ciao toplevel shell and load the file. You should now be able to switch the the toplevel shell and make queries from within `emacs`.

  Note: when using `emacs` it is *very convenient* to swap the locations of the (normally not very useful) ⟨Caps Lock⟩ key and the (very useful in `emacs`) ⟨Ctrl⟩ key on the keyboard. How to do this is explained in the `emacs` frequently asked questions FAQs (see the `emacs` download instructions for their location).

  (if these things do not work the system or emacs may not be installed properly).
- You can go to the location of most of the errors that may be reported during compilation by typing ⟨^C⟩ ⟨`⟩.
- You can also, e.g., create executables from the `Ciao/Prolog` menu, as well as compile individual files, or generate active modules.
- Loading a file for source-level debugging using the `Ciao/Prolog` menu (or typing ⟨^C⟩ ⟨d⟩) and then issuing a query should start the source-level debugger and move a marker on the code in a window while execution is stepped through in the window running the Ciao top level.
- You can add the lines needed in Unix for turning any file defining `main/1` into a script from the Ciao/Prolog menu or by typing ⟨^C⟩ ⟨I⟩ ⟨S⟩.
- You can also work with the preprocessor and auto-documenter directly from emacs: see their manuals or browse through the corresponding menus that appear when editing `.pl` files.

We encourage you once more to read Chapter 19 [Using Ciao inside GNU emacs], page 97 to discover the many other functionalities of this environment.

7 Troubleshooting

Author(s): The Ciao Development Team.

If you experience any problem downloading, installing, or using Ciao you can send a message to `ciao-bug@cliplab.org`. Please be as informative as possible in your messages, so that we can reproduce the bug.

- For *installation problems* we typically need to have the installation method, the version and patch number of the Ciao package (e.g., the name of the file downloaded), the output produced by the installation process (you can capture it by redirecting the output into a file or cutting and pasting with the mouse), and the exact version of the Operating System you are using (as well as the C compiler, if you took a source distribution).
- For *problems during use* we also need the Ciao and OS versions and, if possible, a small example of code which we can run to reproduce the bug.

This is a list of common issues and their solutions:

- **Problem:** Compilation errors appear when trying a new installation/compilation after the previous one was aborted (e.g., because of errors) or after a source upgrade from the development version.

  Possible reason and solution: It is a good idea to clean up any leftovers from the previous compilation. It the worst case, if none of the clean commands work, try `./ciao-boot.sh emergency-clean`.

8 PART I - The program development environment

Author(s): The Ciao Development Team.

This part documents the components of the basic Ciao program development environment. They include:

ciao: the Ciao *super-command*, which invokes the toplevel, compiler, or the build system for bundle management.

ciaosh: (also invoked simply as `ciao`) is an interactive top-level shell, similar to the one found on most Prolog systems (with some enhancements). It is *embeddable*, in the sense that it can be included as a library in executables.

debugger: a Byrd box-type debugger, similar to the one found on most Prolog systems (also with some enhancements, such as source-level debugging). This is not a standalone application, but is rather available from `ciaosh`, as is done in other systems supporting the Prolog language. However, it is also *embeddable*, and it can be activated dynamically and conditionally while running executables.

ciaoc: the standalone compiler, which creates executables without having to enter the interactive top-level.

ciao-shell: an interpreter/compiler for *Ciao scripts* (i.e., files containing Ciao code which run without needing explicit compilation).

Ciao emacs interface:
a *complete program development enviroment*, based on GNU emacs, with syntax coloring, direct access to all the tools described above (as well as the preprocessor and the documenter), atomatic location of errors, source-level debugging, context-sensitive access to on-line help/manuals, etc. The use of this environment is *very highly recommended*!

The Ciao program development environment also includes `ciaopp`, the preprocessor, and `lpdoc`, the documentation generator, which are described in separate manuals.

9 The interactive top-level shell

Author(s): Daniel Cabeza, The Ciao Development Team.

`ciaosh` is the Ciao interactive top-level shell. It provides the user with an interactive programming environment with tools for incrementally building programs, debugging programs by following their executions, and modifying parts of programs without having to start again from scratch. If available, it is strongly recommended to use it with the emacs interface provided, as it greatly simplifies the operation. This chapter documents general operation in the shell itself.

9.1 Shell invocation and startup

When invoked, the shell responds with a message of identification and the prompt `?-` as soon as it is ready to accept input.

When the shell is initialized it looks for a file `.ciaorc` in the HOME directory and makes an `include` of it, if it exists. This file is useful for including `use_module/1` declarations for the modules one wants to be loaded by default, changing prolog flags, etc. (Note that the `.ciaorc` file can only contain directives, not actual code; to load some code at startup put it in a separate file and load it using e.g. a `use_module/1` declaration.) If the initialization file does not exist, the default package `default` is included, to provide more or less what other prologs define by default. Thus, if you want to have available all builtins you had before adding the initialization file, you have to include `:- use_package(default)` in it. Two command-line options control the loading of the initialization file:

`-f` Fast start, do not load any initialization file.

`-q` Quiet, do not show banner.

`-i` Force interactive even if `stdin` is not a pseudo-terminal.

`-l File` Look for initialization file `File` instead of `~/.ciaorc`. If it does not exist, include the default package.

`-u File` Use module `File` on startup.

`-p Prompt` Set `Prompt` as prompt.

`-e Query` Call `Query` on startup.

9.2 Shell interaction

After the shell outputs the prompt, it is expecting either an internal command (see the following sections) or a *query* (a goal or sequence of goals). When typing in the input, which must be a valid prolog term, if the term does not end in the first line, subsequent lines are indented. For example:

```
?- X =
   f(a,
   b).

X = f(a,b) ?

yes
?-
```

The queries are executed by the shell as if they appeared in the user module. Thus, in addition to builtin predicates, predicates available to be executed directly are all predicates defined by loaded user files (files with no module declaration), and imported predicates from modules by the use of `use_module`.

The possible answers of the shell, after executing an internal command or query, are:

- If the execution failed (or produced an error), the answer is `no`.
- If the execution was successful and bindings where made (or constraints where imposed) on answer variables, then the shell outputs the values of answer variables, as a sequence of bindings (or constraints), and then prints a `?` as a prompt. At this point it is expecting an input line from the user. By entering a carriage-return (⟨RET⟩) or any line starting with `y`, the query terminates and the shell answer `yes`. Entering a '`,`' the shell enters a recursive level (see below). Finally, any other answer forces the system to backtrack and look for the next solution (answering as with the first solution).
- If the execution was successful, but no answer variable was bound or constrained, the answer is simply `yes`. This behavior can be changed by setting the prolog flag `prompt_alternatives_no_bindings` to `on`, so that if there are more solutions the user will be consulted as explained in the previous point (useful if the solutions produce side effects).

To allow using connection variables in queries without having to report their results, variables whose name starts with `_` are not considered in answers, the rest being the *answer variable*s. This example illustrates the previous points:

```
?- member(a, [b, c]).

no
?- member(a, [a, b]).

yes
?- member(X, [a|L]).

X = a ? ;

L = [X|_] ?

yes
?- atom_codes(ciao, _C), member(L, _C).

L = 99 ? ;

L = 105 ? ;

L = 97 ? ;

L = 111 ? ;

no
?-
```

9.3 Entering recursive (conjunctive) shell levels

As stated before, when the user answers with '`,`' after a solution is presented, the shell enters a *recursive level*, changing its prompt to *N* `?-` (where *N* is the recursion level) and keeping the bindings or constraints of the solution (this is inspired by the *LogIn* language developed by H. Ait-Kaci, P. Lincoln and Roger Nasr [AKNL86]). Thus, the following queries will be executed within that context, and all variables in the lower level solutions will be reported in subsequent solutions at this level. To exit a recursive level, input an ⟨EOF⟩ character or the command `up`. The last solution after entering the level is repeated, to allow asking for more solutions. Use command `top` to exit all recursive levels and return to the top level. Example interaction:

```
?- directory_files('.',_Fs), member(F,_Fs).

F = 'stream_utils.po' ? ,

1 ?- file_property(F, mod_time(T)).

F = 'stream_utils.po',
T = 923497679 ?

yes
1 ?- up.

F = 'stream_utils.po' ? ;

F = 'stream_utils.pl' ? ;

F = 'stream_utils.itf' ? ,

1 ?- file_property(F, mod_time(T)).

F = 'stream_utils.itf',
T = 923497679 ?

yes
1 ?- ^D
F = 'stream_utils.itf' ?

yes
?-
```

9.4 Usage and interface

- **Library usage:**
  The following predicates can be used at the top-level shell natively (but see also the commands available in Chapter 10 [The interactive debugger], page 63 which are also available within the top-level shell).
- **Exports:**
  - *Predicates:*
    `use_module/1`, `use_module/2`, `ensure_loaded/1`, `make_exec/2`, `include/1`, `use_package/1`, `consult/1`, `compile/1`, `./2`, `make_po/1`, `unload/1`, `set_debug_mode/1`, `set_nodebug_mode/1`, `force_lazy/1`, `undo_force_lazy/1`, `dynamic_search_path/1`, `multifile/1`.

9.5 Documentation on exports

use_module/1: PREDICATE

Usage: `use_module(Module)`

Load into the top-level the module defined in `Module`, importing all the predicates it exports.

– *The following properties should hold at call time:*
  `Module` is a source name. (`stream_basic:sourcename/1`)

use_module/2: PREDICATE

Usage: `use_module(Module,Imports)`

Load into the top-level the module defined in `Module`, importing the predicates in `Imports`.

– *The following properties should hold at call time:*
  `Module` is a source name. (`stream_basic:sourcename/1`)
  `Imports` is a list of `prednames`. (`basic_props:list/2`)

ensure_loaded/1: PREDICATE

Usage: `ensure_loaded(File)`

Load into the top-level the code residing in file (or files) `File`, which is user (i.e. non-module) code.

– *The following properties should hold at call time:*
  `File` is a source name or a list of source names. (`toplevel_doc:sourcenames/1`)

make_exec/2: PREDICATE

Usage: `make_exec(Files,ExecName)`

Make a Ciao executable from file (or files) `Files`, giving it name `ExecName`. If `ExecName` is a variable, the compiler will choose a default name for the executable and will bind the variable `ExecName` to that name. The name is chosen as follows: if the main prolog file has no `.pl` extension or we are in Windows, the executable will have extension `.cpx`; else the executable will be named as the main prolog file without extension.

– *The following properties should hold at call time:*
  `Files` is a source name or a list of source names. (`toplevel_doc:sourcenames/1`)
– *The following properties should hold upon exit:*
  `ExecName` is an atom. (`basic_props:atm/1`)

include/1: PREDICATE

Usage: `include(File)`

The contents of the file `File` are included in the top-level shell. For the moment, it only works with some directives, which are interpreted by the shell, or with normal clauses (which are asserted).

– *The following properties should hold at call time:*
  `File` is a source name. (`stream_basic:sourcename/1`)

use_package/1: PREDICATE

Usage: `use_package(Package)`

Include the package or packages specified in `Package`. Most package contents can be handled in the top level, but there are currently still some limitations.

– *The following properties should hold at call time:*

`Package` is a source name or a list of source names. (`toplevel_doc:sourcenames/1`)

consult/1: PREDICATE

Usage: `consult(File)`

Provided for backward compatibility. Similar to `ensure_loaded/1`, but ensuring each listed file is loaded in consult mode (see Chapter 10 [The interactive debugger], page 63).

– *The following properties should hold at call time:*

`File` is a source name or a list of source names. (`toplevel_doc:sourcenames/1`)

compile/1: PREDICATE

Usage: `compile(File)`

Provided for backward compatibility. Similar to `ensure_loaded/1`, but ensuring each listed file is loaded in compile mode (see Chapter 10 [The interactive debugger], page 63).

– *The following properties should hold at call time:*

`File` is a source name or a list of source names. (`toplevel_doc:sourcenames/1`)

./2: PREDICATE

Usage: `[File|Files]`

Provided for backward compatibility, obsoleted by `ensure_loaded/1`.

– *The following properties should hold at call time:*

`File` is a source name. (`stream_basic:sourcename/1`)

`Files` is a list of `sourcenames`. (`basic_props:list/2`)

make_po/1: PREDICATE

Usage: `make_po(Files)`

Make object (`.po`) files from `Files`. Equivalent to executing `"ciaoc -c"` on the files.

– *The following properties should hold at call time:*

`Files` is a source name or a list of source names. (`toplevel_doc:sourcenames/1`)

unload/1: PREDICATE

Usage: `unload(File)`

Unloads dynamically loaded file `File`.

– *The following properties should hold at call time:*

`File` is a source name. (`stream_basic:sourcename/1`)

set_debug_mode/1: PREDICATE

Usage: `set_debug_mode(File)`

Set the loading mode of `File` to *consult*. See Chapter 10 [The interactive debugger], page 63.

- *The following properties should hold at call time:*

  `File` is a source name. (`stream_basic:sourcename/1`)

set_nodebug_mode/1: PREDICATE

Usage: `set_nodebug_mode(File)`

Set the loading mode of `File` to *compile*. See Chapter 10 [The interactive debugger], page 63.

- *The following properties should hold at call time:*

  `File` is a source name. (`stream_basic:sourcename/1`)

force_lazy/1: PREDICATE

Usage: `force_lazy(Module)`

Force module of name `Module` to be loaded lazily in the subsequent created executables.

- *The following properties should hold at call time:*

  `Module` is an atom. (`basic_props:atm/1`)

undo_force_lazy/1: PREDICATE

Usage: `undo_force_lazy(Module)`

Disable a previous `force_lazy/1` on module `Module` (or, if it is uninstantiated, all previous `force_lazy/1`).

- *Call and exit should be compatible with:*

  `Module` is an atom. (`basic_props:atm/1`)

dynamic_search_path/1: PREDICATE

Usage: `dynamic_search_path(Name)`

Asserting a fact to this data predicate, files using path alias `Name` will be treated as dynamic in the subsequent created executables.

- *The following properties should hold at call time:*

  `Name` is an atom. (`basic_props:atm/1`)

multifile/1: PREDICATE

Usage: `multifile Pred`

Dynamically declare predicate `Pred` as multifile. This is useful at the top-level shell to be able to call multifile predicates of loaded files.

- *The following properties should hold at call time:*

  `Pred` is a predicate name. (`basic_props:predname/1`)

9.6 Documentation on internals

sourcenames/1: PROPERTY

Is defined as follows:

```
sourcenames(File) :-
    sourcename(File).
sourcenames(Files) :-
    list(sourcename,Files).
```

See `sourcename/1` in Chapter 144 [Basic file/stream handling], page 619

Usage: `sourcenames(Files)`

`Files` is a source name or a list of source names.

9.7 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `toplevel`, `libpaths`, `compiler`, `exemaker`, `c_itf`, `debugger`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

10 The interactive debugger

Author(s): Daniel Cabeza, Manuel C. Rodriguez, Edison Mera, A. Ciepielewski (first version), Mats Carlsson (first version), T. Chikayama (first version), K. Shen (first version).

The Ciao program development environment includes a number of advanced debugging tools, such as a source-level debugger, the `ciaopp` preprocessor, and some execution visualizers. Herein we discuss the interactive debugger available in the standard top-level, which allows tracing the control flow of programs, in a similar way to other popular Prolog systems. This is a classical Byrd *box-type debugger* [Byr80, BBP81], with some enhancements, most notably being able to track the execution on the source program.

We also discuss the embedded debugger, which is a version of the debugger which can be embedded into executables. It allows triggering an interactive debugging session at any time while running an executable, without any need for the top-level shell.

Byrd's Procedure Box model of debugging execution provides a simple way of visualising control flow, including backtracking. Control flow is in principle viewed at the predicate level, rather than at the level of individual clauses. The Ciao debugger has the ability to mark selected modules and/or files for debugging (traditional and source debugging), rather than having to exhaustively trace the program. It also allows to selectively set spy-points and breakpoints. Spy-points allow the programmer to nominate interesting predicates at which program execution is to pause so that the programmer can interact with the debugger. Breakpoints are similar to spy-points, but allow pausing at a specific line in the code, corresponding to a particular literal. There is a wide choice of control and information options available during debugging interaction.

Note: While the debugger described herein can be used in a standalone way (i.e., from an operating system shell or terminal window) in the same way as other Prolog debuggers, the most convenient way of debugging Ciao programs is by using the programming environment (see Chapter 19 [Using Ciao inside GNU emacs], page 97). This environment has many debugging-related facilities, including displaying the source code for the module(s) corresponding to the procedure being executed, and higlighting dynamically the code segments corresponding to the different execution steps.

10.1 Marking modules and files for debugging in the top-level debugger

The Ciao debugger is module-based. This allows skipping during the debugging process all files (including system library files) except those in which a bug is suspected. This saves having to explictly and repetitively skip predicates in unrelated files during the debugging process. Also, there is an efficieny advantage: in order to be able to run the debugger on a module, it must be loaded in *debug (interpreted) mode*, which will execute slower than normal (compiled) modules. Thus, it is interesting to compile in debug mode only those modules that need to be traced. Instead of doing this (loading of modules in one mode or another) by hand each time, in Ciao (re)loading of modules in the appropriate mode is handled automatically by the Ciao compiler. However, this requires the user to *mark explicitly* the modules in which debugging is to be performed. The simplest way of achieving this is by executing in the Ciao shell prompt, for each suspicious module `Module` in the program, the command:

```
?- debug_module(Module).
```

or, alternatively:

```
?- debug_module_source(Module).
```

which in addition instructs the debugger to keep track of the line numbers in the source file and to report them during debugging. This is most useful when running the top-level inside the `emacs` editor since in that case the Ciao emacs mode allows performing full source-level debugging in each module marked as above, i.e., the source lines being executed will be highlighted dynamically during debugging in a window showing the source code of the module.

Note that, since all files with no module declaration belong to the pseudo-module `user`, the command to be issued for debugging a user file, say `foo.pl`, is `debug_module(user)` or `debug_module_source(user)`, and not `debug_module(foo)`.

The two ways of performing source-level debugging are fully compatible between them, i.e., Ciao allows having some modules loaded with `debug_module/1` and others with `debug_module_source/1`. To change from one interpreted mode to the other mode it suffices to select the module with the new interpreted mode (debugger mode), using the appropiate command, and reload the module.

The commands above perform in fact two related actions: first, they let the compiler know that if a file containing a module with this name is loaded, it should be loaded in interpreted mode (source or traditional). In addition, they instruct the debugger to actually prepare for debugging the code belonging to that module. After that, the modules which are to be debugged have to be (re)loaded so that they are compiled or loaded for interpretation in the appropriate way. The nice thing is that, due to the modular behaviour of the compiler/top-level, if the modules are part of a bigger application, it suffices to load the main module of the application, since this will automatically force the dependent modules which have changed to be loaded in the appropriate way, including those whose *loading mode* has changed (i.e., changing the loading mode has the effect of forcing the required re-loading of the module at the appropriate time).

Later in the debugging process, as the bug location is isolated, typically one will want to restrict more and more the modules where debugging takes place. To this end, and without the need for reloading, one can tell the debugger to not consider a module for debugging issuing a `nodebug_module/1` command, which counteracts a `debug_module/1` or `debug_module_source/1` command with the same module name, and reloading it (or the main file).

There are also two top-level commands `set_debug_mode/1` and `set_nodebug_mode/1`, which accept as argument a file spec (i.e., `library(foo)` or `foo`, even if it is a user file) to be able to load a file in interpreted mode without changing the set of modules that the debugger will try to spy.

10.2 The debugging process

Once modules or user files are marked for debugging and reloaded, the traditional debugging shell commands can be used (the documentation of the `debugger` library following this chapter contains all the commands and their description), with the same meaning as in other classical Prolog systems. The differences in their behavior are:

- Debugging takes place only in the modules in which it was activated,
- `nospy/1` and `spy/1` accept sequences of predicate specs, and they will search for those predicates only in the modules marked for debugging (traditional or source-level debugging).
- `breakpt/6` and `nobreakpt/6` allow setting breakpoints at selected clause literals and will search for those literals only in the modules marked for source-level debugging (modules marked with `debug_module_source/1`).

In particular, the system is initially in nodebug mode, in which no tracing is performed. The system can be put in debug mode by a call to `debug/0` in which execution of queries will proceed until the first *spy-point* or *breakpoint*. Alternatively, the system can be put in trace mode by a call to `trace/0` in which all predicates will be trace.

10.3 Marking modules and files for debugging with the embedded debugger

The embedded debugger, as the interpreted debugger, has three different modes of operation: debug, trace or nodebug. These debugger modes can be set by adding *one* of the following package declarations to the module:

```
:- use_package(debug).
:- use_package(trace).
:- use_package(nodebug).
```

and recompiling the application. These declarations *must* appear the last ones of all `use_package` declarations used. Also it is possible, as usual, to add the debugging package(s) in the module declaration using the third argument of the `module/3` declaration (and they should also be the last ones in the list), i.e., using one of:

```
:- module(..., ..., [..., debug]).
:- module(..., ..., [..., trace]).
:- module(..., ..., [..., nodebug]).
```

The nodebug mode allows turning off any debugging (and also the corresponding overhead) but keeping the spy-points and breakpoints in the code. The trace mode will start the debugger for any predicate in the file.

The embedded debugger has limitations over the interpreted debugger. The most important is that the "retry" option is not available. But it is possible to add, and remove, spy-points and breakpoins using the predicates `spy/1`, `nospy/1`, `breakpt/6` and `nobreakpt/6`, etc. These can be used within clauses or as declarations. Also it is possible to add in the code predicates for switching the debugger on or off (i.e., use debug mode, and in a clause add the predicate `trace/0`). Finally, if a spy declaration is placed on the entry point of an executable (`:- spy(main/1)`) the debugger will not start the first time `main/1` predicate is called, i.e., at the beginning of program execution (however, it will if there are any subsequent calls to `main/1`). Starting the embedded debugger at the beginning of the execution of a program can be done easily however by simply using the `trace` package.

Note that there is a particularly interesting way of using the embedded debugger: if an *application* is run in a shell buffer which has been set with Ciao inferior mode (⟨M-x⟩ `ciao-inferior-mode`) and this application starts emitting output from the embedded debugger (i.e., which contains the embedded debugger and is debugging its code) then the Ciao emacs mode will be able to follow these messages, for example tracking execution in the source level code. This also works if the application is written in a combination of languages, provided the parts written in Ciao are compiled with the embedded debugger package and is thus a covenient way of debugging multi-language applications. The only thing needed is to make sure that the output messages appear in a shell buffer that is in Ciao inferior mode.

See the following as a general example of use of the embedded debugger:

```
:- module( foo,[main/1],[assertions, debug]).

:- use_module(engine(io_basic)).

:- entry main/1.

main(X) :-
     display(X),
     spy(foo),
     foo(X),
     notrace,
     nl.
```

```
foo([]).
foo([X|T]) :-
    trace,
    bar(X),
    foo(T).

bar(X) :-
    display(X).
```

10.4 The procedure box control flow model

During debugging the interpreter prints out a sequence of goals in various states of instantiation in order to show the state that the program has reached in its execution. However, in order to understand what is occurring it is necessary to understand when and why the interpreter prints out goals. As in other programming languages, key points of interest are procedure entry and return, but in Prolog there is the additional complexity of backtracking. One of the major confusions that novice Prolog programmers have to face is the question of what actually happens when a goal fails and the system suddenly starts backtracking. The Procedure Box model of Prolog execution views program control flow in terms of movement about the program text. This model provides a basis for the debugging mechanism in the interpreter, and enables the user to view the behaviour of the program in a consistent way. It also provides the basis for the visualization performed on the source level program when source level program when source-level debugging is activated within `emacs`.

Let us look at an example Prolog procedure:

The first clause states that `Y` is a descendant of `X` if `Y` is an offspring of `X`, and the second clause states that `Y` is a descendant of `X` if `Z` is an offspring of `X` and `Y` is a descendant of `Z`. In the diagram a box has been drawn around the whole procedure and labelled arrows indicate the control flow in and out of this box. There are four such arrows which we shall look at in turn.

- **Call**

  This arrow represents initial invocation of the procedure. When a goal of the form `descendant(X,Y)` is required to be satisfied, control passes through the Call port of the descendant box with the intention of matching a component clause and then satisfying any subgoals in the body of that clause. Note that this is independent of whether such a match is possible; i.e. first the box is called, and then the attempt to match takes place. Textually we can imagine moving to the code for descendant when meeting a call to descendant in some other part of the code.

- **Exit**

  This arrow represents a successful return from the procedure. This occurs when the initial goal has been unified with one of the component clauses and any subgoals have been satisfied. Control now passes out of the Exit port of the descendant box. Textually we stop following the code for descendant and go back to the place we came from.

- **Redo**

  This arrow indicates that a subsequent goal has failed and that the system is backtracking in an attempt to find alternatives to previous solutions. Control passes through the Redo port of the descendant box. An attempt will now be made to resatisfy one of the component subgoals in the body of the clause that last succeeded; or, if that fails, to completely rematch the original goal with an alternative clause and then try to satisfy any subgoals in the body of this new clause. Textually we follow the code backwards up the way we came looking for new ways of succeeding, possibly dropping down on to another clause and following that if necessary.

- **Fail**

  This arrow represents a failure of the initial goal, which might occur if no clause is matched, or if subgoals are never satisfied, or if any solution produced is always rejected by later processing. Control now passes out of the Fail port of the descendant box and the system continues to backtrack. Textually we move back to the code which called this procedure and keep moving backwards up the code looking for choice points.

In terms of this model, the information we get about the procedure box is only the control flow through these four ports. This means that at this level we are not concerned with which clause matches, and how any subgoals are satisfied, but rather we only wish to know the initial goal and the final outcome. However, it can be seen that whenever we are trying to satisfy subgoals, what we are actually doing is passing through the ports of *their* respective boxes. If we were following this (e.g., activating source-level debugging), then we would have complete information about the control flow inside the procedure box.

Note that the box we have drawn around the procedure should really be seen as an invocation box. That is, there will be a different box for each different invocation of the procedure. Obviously, with something like a recursive procedure, there will be many different Calls and Exits in the control flow, but these will be for different invocations. Since this might get confusing each invocation box is given a unique integer identifier in the messages, as described below.

Note that not all procedure calls are traced; there are a few basic predicates which have been made invisible since it is more convenient not to trace them. These include debugging directives, basic control structures, and some builtins. This means that messages will never be printed for these predicates during debugging.

10.5 Format of debugging messages

This section explains the two formats of the message output by the debugger at a port. All trace messages are output to the terminal regardless of where the current output stream is directed (which allows tracing programs while they are performing file I/O). The basic format, which will be shown in traditional debug and in source-level debugging within Ciao `emacs` mode, is as follows:

```
S  13  7  Call: T user:descendant(dani,_123) ?
```

`S` is a spy-point or breakpoint indicator. It is printed as '`+`', indicating that there is a spy-point on `descendant/2` in module `user`, as `'B'` denoting a breakpoint, or as ' ', denoting no spy-point or breakpoint. If there is a spy-point and a breakpoint in the same predicate the spy-point indicator takes preference over breakpoint indicator.

`T` is a subterm trace. This is used in conjunction with the `^` command (set subterm), described below. If a subterm has been selected, `T` is printed as the sequence of commands used to select the subterm. Normally, however, `T` is printed as ' ', indicating that no subterm has been selected.

The first number is the unique invocation identifier. It is always nondecreasing (provided that the debugger is switched on) regardless of whether or not the invocations are being actually seen. This number can be used to cross correlate the trace messages for the various ports, since it is unique for every invocation. It will also give an indication of the number of procedure calls made since the start of the execution. The invocation counter starts again for every fresh execution of a command, and it is also reset when retries (see later) are performed.

The number following this is the *current depth*; i.e., the number of direct *ancestors* this goal has. The next word specifies the particular port (`Call`, `Exit`, `Redo` or `Fail`). The goal is then printed so that its current instantiation state can be inspected. The final `?` is the prompt indicating that the debugger is waiting for user interaction. One of the option codes allowed (see below) can be input at this point.

The second format, quite similar to the format explained above, is shown when using source-level debugging outside the Ciao `emacs` mode, and it is as follows:

```
         In /home/mcarlos/ciao/foo.pl (5-9) descendant-1
   S  13  7  Call: T user:descendant(dani,_123) ?
```

This format is identical to the format above except for the first line, which contains the information for location of the point in the source program text where execution is currently at. The first line contains the name of the source file, the start and end lines where the literal can be found, the substring to search for between those lines and the number of substrings to locate. This information for locating the point on the source file is not shown when executing the source-level debugger from the Ciao `emacs` mode.

Ports can be "unleashed" by calling the `leash/1` predicate omiting that port in the argument. This means that the debugger will stop but user interaction is not possible for an unleashed port. Obviously, the `?` prompt will not be shown in such messages, since the user has specified that no interaction is desired at this point.

10.6 Options available during debugging

This section describes the particular options that are available when the debugger prompts after printing out a debugging message. All the options are one letter mnemonics, some of which can be optionally followed by a decimal integer. They are read from the terminal with any blanks being completely ignored up to the next terminator (carriage-return, line-feed, or escape). Some options only actually require the terminator; e.g., the creep option, only requires ⟨RET⟩.

The only option which really needs to be remembered is '`h`' (followed by ⟨RET⟩). This provides help in the form of the following list of available options.

```
<cr>   creep            c      creep
 l     leap             s      skip
 r     retry            r <i>  retry i
 f     fail             f <i>  fail i
 d     display          p      print
 w     write            v <I>  variable(s)
 g     ancestors        g <n>  ancestors n
 n     nodebug          =      debugging
 +     spy this         -      nospy this
 a     abort
 @     command          u      unify
 <     reset printdepth < <n>  set printdepth
 ^     reset subterm    ^ <n>  set subterm
 ?     help             h      help
```

- `c` (*creep*)

  causes the debugger to single-step to the very next port and print a message. Then if the port is leashed the user is prompted for further interaction. Otherwise it continues creeping. If leashing is off, creep is the same as leap (see below) except that a complete trace is printed on the terminal.
- `l` (*leap*)

  causes the interpreter to resume running the program, only stopping when a spy-point or breakpoint is reached (or when the program terminates). Leaping can thus be used to follow the execution at a higher level than exhaustive tracing. All that is needed to do is to set spy-points and breakpoints on an evenly spread set of pertinent predicates or lines, and then follow the control flow through these by leaping from one to the other.
- `s` (*skip*)

  is only valid for Call and Redo ports, if it is issued in Exit or Fail ports it is equivalent to creep. It skips over the entire execution of the predicate. That is, no message will be seen until control comes back to this predicate (at either the Exit port or the Fail port). Skip is particularly useful while creeping since it guarantees that control will be returned after the (possibly complex) execution within the box. With skip then no message at all will appear until control returns to the Exit port or Fail port corresponding to this Call port or Redo port. This includes calls to predicates with spy-points and breakpoints set: they will be masked out during the skip. There is a way of overriding this: the `t` option after a ^C interrupt will disable the masking. Normally, however, this masking is just what is required!

- `r` (*retry*)

  can be used at any of the four ports (although at the Call port it has no effect). It transfers control back to the Call port of the box. This allows restarting an invocation when, for example, it has left the programmer with some weird result. The state of execution is exactly the same as in the original call (unless the invocation has performed side effects, which will not be undone). When a retry is performed the invocation counter is reset so that counting will continue from the current invocation number regardless of what happened before the retry. This is in accord with the fact that execution has, in operational terms, returned to the state before anything else was called.

  If an integer is supplied after the retry command, then this is taken as specifying an invocation number and the system tries to get to the Call port, not of the current box, but of the invocation box specified. It does this by continuously failing until it reaches the right place. Unfortunately this process cannot be guaranteed: it may be the case that the invocation the programmer is looking for has been cut out of the search space by cuts in the program. In this case the system fails to the latest surviving Call port before the correct one.

- `f` (*fail*)

  can be used at any of the four ports (although at the Fail port it has no effect). It transfers control to the Fail port of the box, forcing the invocation to fail prematurely. If an integer is supplied after the command, then this is taken as specifying an invocation number and the system tries to get to the Fail port of the invocation box specified. It does this by continuously failing until it reaches the right place. Unfortunately, as before, this process cannot be guaranteed.

- `d` (*display*)

  displays the current goal using `display/1`. See `w` below.

- `p` (*print*)

  re-prints the current goal using `print/1`. Nested structures will be printed to the specified *printdepth* (see below).

- `w` (*write*)

  writes the current goal on the terminal using `write/1`.

- `v` (*variables*)

  writes the list of the modified variables and their values. If a variable name (identifier) `N` is supplied, then the value of variable `N` is shown.

- `g` (*ancestors*)

  provides a list of ancestors to the current goal, i.e., all goals that are hierarchically above the current goal in the calling sequence. It is always possible to jump to any goal in the ancestor list (by using retry, etc.). If an integer `n` is supplied, then only `n` ancestors will be printed. That is to say, the last `n` ancestors will be printed counting back from the current goal. Each entry in the list is preceded by the invocation number followed by the depth number (as would be given in a trace message).

- `n` (*nodebug*)

  switches the debugger off. Note that this is the correct way to switch debugging off at a trace point. The `@` option cannot be used because it always returns to the debugger.

- `=` (*debugging*)

  outputs information concerning the status of the current debugging session.
- `+` *spy*

  sets a spy-point on the current goal.
- `-` (*nospy*)

  removes the spy-point from the current goal.
- `a` (*abort*)

  causes an abort of the current execution. All the execution states built so far are destroyed and the system is put right back at the top-level of the interpreter. (This is the same as the built-in predicate `abort/0`.)
- `@` (*command*)

  allows calling arbitrary goals. The initial message `| ?-` will be output on the terminal, and a command is then read from the terminal and executed as if it was at top-level.
- `u` (*unify*)

  is available at the Call port and gives the option of providing a solution to the goal from the terminal rather than executing the goal. This is convenient, e.g., for providing a "stub" for a predicate that has not yet been written. A prompt `|:` will be output on the terminal, and the solution is then read from the terminal and unified with the goal.
- `<` (*printdepth*)

  sets a limit for the subterm nesting level that is printed in messages. While in the debugger, a printdepth is in effect for limiting the subterm nesting level when printing the current goal. When displaying or writing the current goal, all nesting levels are shown. The limit is initially 10. This command, without arguments, resets the limit to 10. With an argument of `n` the limit is set to `n`.
- `^` (*subterm*)

  sets the subterm to be printed in messages. While at a particular port, a current subterm of the current goal is maintained. It is the current subterm which is displayed, printed, or written when prompting for a debugger command. Used in combination with the printdepth, this provides a means for navigating in the current goal for focusing on the part which is of interest. The current subterm is set to the current goal when arriving at a new port. This command, without arguments, resets the current subterm to the current goal. With an argument of `n` (greater than 0 and less or equal to the number of subterms of the current subterm), the current subterm is replaced by its `n`'th subterm. With an argument of `0`, the current subterm is replaced by its parent term.
- `?` or `h` (*help*)

  displays the table of options given above.

10.7 Calling predicates that are not exported by a module

The Ciao module system does not allow calling predicates which are not exported during debugging. However, as an aid during debugging, this is allowed (only from the top-level and for modules which are in debug mode or source-level debug mode) using the `call_in_module/2` predicate.

Note that this does not affect analysis or optimization issues, since it only works on modules which are loaded in debug mode or source-level debug mode, i.e., unoptimized.

10.8 Acknowledgements (debugger)

Originally written by Andrzej Ciepielewski. Minor modifications by Mats Carlsson. Later modifications (17 Dec 87) by Takashi Chikayama (making tracer to use `print/1` rather than `write/1`, temporarily switching debugging flag off while writing trace message and within "break" level). Additional modifications by Kish Shen (May 88): subterm navigation, handle unbound args in `spy/1` and `nospy/1`, trapping arithmetics errors in debug mode. Adapted then to &-Prolog and Ciao by Daniel Cabeza and included in the Ciao version control system. Extended for source-level debugging by Manuel C. Rodríguez. Option that shows variable names and values (`v <N>`) implemented by Edison Mera (2009). (See changelog if included in the document or in the version maintenance system for more detailed documentation on changes.)

11 Predicates controlling the interactive debugger

Author(s): A. Ciepielewski, Mats Carlsson, T. Chikayama, K. Shen, Daniel Cabeza, Manuel C. Rodriguez, Edison Mera, Jose F. Morales.

This library implements predicates which are normally used in the interactive top-level shell to debug programs. A subset of them are available in the embeddable debugger.

11.1 Usage and interface

- **Library usage:**
  `:- use_module(library(debugger)).`
- **Exports:**
  - *Predicates:*
    `call_in_module/2`.

11.2 Documentation on exports

call_in_module/2: PREDICATE

Usage: `call_in_module(Module,Predicate)`

Calls predicate `Predicate` belonging to module `Module`, even if that module does not export the predicate. This only works for modules which are in debug (interpreted) mode (i.e., they are not optimized).

- *The following properties should hold at call time:*
  `Module` is an atom. (`basic_props:atm/1`)
  `Predicate` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

breakpt/6: (UNDOC_REEXPORT)

Imported from `debugger_lib` (see the corresponding documentation for details).

current_debugged/1: (UNDOC_REEXPORT)

Imported from `debugger_lib` (see the corresponding documentation for details).

debug/0: (UNDOC_REEXPORT)

Imported from `debugger_lib` (see the corresponding documentation for details).

debug_module/1: (UNDOC_REEXPORT)

Imported from `debugger_lib` (see the corresponding documentation for details).

debug_module_source/1: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

debugging/0: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

get_debugger_state/1: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

get_debugger_state/1: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

leash/1: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

list_breakpt/0: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

maxdepth/1: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

nobreakall/0: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

nobreakpt/6: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

nodebug/0: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

nodebug_module/1: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

nospy/1: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

nospyall/0: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

notrace/0: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

spy/1: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

trace/0: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

debugrtc/0: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

nodebugrtc/0: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

tracertc/0: (UNDOC_REEXPORT)
Imported from `debugger_lib` (see the corresponding documentation for details).

11.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `debugger_lib`, `format`, `toplevel_io`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

12 Bundle management

Author(s): Jose F. Morales, The Ciao Development Team.

The `ciao` command implements most bundle management operations. The most important commands are:

```
$ ciao list             # List bundles in current workspaces
$ ciao build [<tgts>]    # Build
$ ciao build -r [<tgts>] # Build, including dependencies
$ ciao fetch [<tgts>]    # Fetch all dependencies of a bundle
$ ciao get [<tgts>]      # Fetch, build, and install a bundle
$ ciao rm [<tgts>]       # Remove a bundle
```

Most commands accept one ore more targets (workspace paths, bundle directories, bundle names, or bundle aliases for location on the network). For example, valid targets are:

- (empty) (in which case Ciao locates the bundle above the current directory)
- bundle names (e.g., `chat80`)
- paths to bundles (e.g., `~/ciao/chat80`)
- URL-like aliases to bundles (e.g., `github.com/jfmc/chat80`, where `chat80` is the bundle name)

Execute `ciao help` to obtain a detailed list of commands.

12.1 Setting up workspaces

Ciao locates bundles by looking at one or more workspaces, given by the following (optional) environment variables:

- `CIAOPATH`: colon-separated list of paths to individual workspaces
- `CIAOROOT`: the root directory of this Ciao installation (either in binary or source form)

Workspaces from `CIAOPATH` are consulted from left to right. If `CIAOPATH` is undefined, a default value of `~/.ciao` will be used. The first encountered workspace directory is denoted as the *top* workspace. The variable `CIAOROOT` is usually not needed, except when the binaries is being relocated on the filesystem.

Note: `CIAOPATH` should not point to `CIAOROOT` (it is not needed and it may produce inconsistent builds).

As mentioned in ⟨undefined⟩ [Full installation instructions], page ⟨undefined⟩, setting up the environment uses the `ciao-env` tool, which updates the `PATH` and other environment variables to make commands and documentation (`man`, `info`) accessible by default.

Configuring workspaces: The directory `~/ciao` can be added permanently as a workspace as follows. Include the following at the `~/.emacs` file:

```
(setenv "CIAOPATH" (expand-file-name "~/ciao"))
```

Then add these lines to the `~/.bashrc` file:

```
export CIAOPATH=~/ciao
```

It is possible to change `CIAOPATH` temporarily from `emacs` (`M-x setenv`) or a shell in a terminal (executing `export CIAOPATH=<path>`)

12.2 Using and creating bundles

Creating new bundles: For creating new bundles it is recommended to define a workspace directory (e.g., `~/ciao`) and set the `CIAOPATH` environment variable to that directory.

Bundles can be created from scratch or cloned from repositories, e.g.:

```
$ cd ~/ciao
$ git clone https://github.com/jfmc/chat80
```

At that point bundles can be managed with the `ciao` command. The `ciao list` should display `chat80` as a recognized bundle, and it should be possible to configure, build, or install it (e.g., `ciao build` at `chat80` directory or `ciao build chat80` anywhere). Most commands will rescan the bundles at the workspace, but it can be done manually with:

```
$ ciao rescan-bundles
```

Note that existing toplevels may need to be restarted to recognize modules at the new bundles.

Network installation: It is possible to download, configure, build, and install automatically bundles from the network. For example:

```
ciao get github.com/jfmc/chat80
```

will fetch and install `chat80` under the top workspace (`~/.ciao` if `CIAOPATH` is unset) (see `bundle_fetch`). The bundle name will be `chat80`, while the rest of the bundle alias is used to locate the location on the network. The command:

```
ciao rm chat80
```

removes the bundle (to prevent accidental data losses, only iff it has been marked as *downloaded*).

12.3 Workspaces during Ciao bootstrap

In the context of calls to `<srcdir>/ciao-boot.sh`, `CIAOROOT` is implicitly set to the directory where `ciao-boot.sh` is located (`<srcdir>`) and `CIAOPATH` is unset. Thus only bundles at `<srcdir>` (including bundle *catalog* directories) are considered.

This is useful to:

- prevent existing `CIAOPATH` values interact with system builds
- fetch bundles during system build (e.g., the `devenv` bundle)

Example: the following command will bootstrap Ciao from source, as well as fetch and install the full development environment:

```
./ciao-boot.sh get devenv
```

12.4 Generating binary distributions

The `ciaoc_sdyn` command can be used to generate binary distributions of executables with complex dependencies, including any 3rd-party dynamic libraries.

Example:

```
mkdir dist; cd dist
ciaoc_sdyn ../src/YOURMAINEXEC
cp ../src/SOMEASSET
```

This creates a platform specific binary YOURMAINEXEC at `dist/` directory, together with the collection of shared libraries for the dependencies.

Copying `.so` or `.dynlib` that corresponds to system libraries are not a good idea. To fix the issue the following command can be used:

```
rm -f libc.so.* libgcc_s.so.* libm.so.* libstdc++.so.*
```

12.4.1 Adding assets to binary distributions

The `bundle_path/3` predicate (at `bundle/bundle_paths`) can be used to obtain the absolute path name of a bundle or a relative path in a bundle.

Example:

```
:- use_module(library(system), [file_exists/1]).
:- use_module(library(pathnames), [path_concat/3]).
:- use_module(library(bundle/bundle_paths), [bundle_path/3]).

% Find file X in the same directory as the executable
% or in the sources.
find_asset(Name, Path) :-
( current_executable(ExecPath),
  path_split(ExecPath, Dir, _)
; bundle_path(<SomeBundle>, 'images', Dir)
),
path_concat(Dir, Name, Path0),
file_exists(Path0),
!,
Path = Path0.
```

The example predicate `find_assert/2` above will look for arbitrary files in either the `images/` directory or the directory of the current executable.

```
find_asset('spaceship.png', Sprite)
```

12.5 Troubleshooting

Q: Changing the (workspace) directory of a bundle.

A: It would be enough to rescan bundles and configure (clean and build is recommended too). Go to the workspace and type the following:

```
ciao rescan-bundles
ciao configure [<configuration-options>]
```

Q: Compilation of some module complains about missing dependencies (to seemingly auto-generated code and third-party libraries).

A: Make sure that the bundle and its dependencies are properly configured and build (e.g., calling `ciao build` for each of them or with `ciao build -r`). Some bundles (e.g., interfaces to external solvers) require pre-build steps to download and compile third-party code. This part of the compilation is not currently handled by the Ciao compiler.

Q: Alias paths configured at `.ciaorc` are not visible in executables.

A: The `.ciaorc` file is read only by the Ciao toplevel (not arbitrary Ciao executables). Configure alias paths through manifest files instead.

13 The standalone command-line compiler

Author(s): The Ciao Development Team.

`ciaoc` [CH00b] is the Ciao standalone command-line compiler. `ciaoc` can be used to create executables or to compile individual files to object code (to be later linked with other files). `ciaoc` is specially useful when working from the command line. Also, it can be called to compile Ciao programs from other tools such as, e.g., shell scripts, `Makefiles`, or project files. All the capabilities of `ciaoc` are also available from the interactive top-level shell, which uses the `ciaoc` modules as its components.

13.1 Introduction to building executables

An *executable* can be built from a single file or from a collection of inter-related files. In the case of only one file, this file must define the predicate `main/0` or `main/1`. This predicate is the one which will be called when the executable is started. As an example, consider the following file, called `hello.pl`:

```
main :-
     write('Hello world'),
     nl.
```

To compile it from the command line using the `ciaoc` standalone compiler it suffices to type "`ciaoc hello`" (in Win32 you may have to put the complete path to the `ciaoc` folder of the Ciao distribution, where the installation process leaves a `ciaoc.bat` file):

```
$ ciaoc hello
```

This produces an executable called `hello` in Unix systems and `hello.cpx` under Win32 systems. This executable can then be run in Win32 by double-clicking on it and on Unix systems by simply typing its name (see ⟨undefined⟩ [Running executables from the command line], page ⟨undefined⟩ for how to run executables from the command line in Win32):

```
$ ./hello
Hello world
```

If the application is composed of several files the process is identical. Assume `hello.pl` is now:

```
:- use_module(aux, [p/1]).

main :-
     p(X),
     write(X),
     nl.
```

where the file `aux.pl` contains:

```
:- module(aux,[p/1]).

p('Hello world').
```

This can again be compiled using the `ciaoc` standalone compiler as before:

```
$ ciaoc hello
$ ./hello
Hello world
```

The invocation of `ciaoc hello` compiles the file `hello.pl` and all connected files that may need recompilation – in this case the file `aux.pl`. Also, if any library files used had not been compiled previously they would be compiled at this point (See ⟨undefined⟩ [Intermediate files in the compilation process], page ⟨undefined⟩). Also, if, say, `hello.pl` is changed and recompiled, the object code resulting from the previous compilation of `aux.pl` will be reused. This is all done without any need for `Makefiles`, and considerably accelerates the development process for large applications. This process can be observed by selecting the `-v` option when invoking `ciaoc` (which is equivalent to setting the `verbose_compilation` Prolog flag to `on` in the top-level interpreter).

If `main/1` is defined instead of `main/0` then when the executable is started the argument of `main/1` will be instantiated to a list of atoms, each one of them corresponding to a command line option. Consider the file `say.pl`:

```
main(Argv) :-
     write_list(Argv), nl.

write_list([]).
write_list([Arg|Args]) :-
     write(Arg),
     write(' '),
     write_list(Args).
```

Compiling this program and running it results in the following output:

```
$ ciaoc say
$ ./say hello dolly
hello dolly
```

The name of the generated executable can be controlled with the `-o` option (See ⟨undefined⟩ [Usage (ciaoc)], page ⟨undefined⟩).

13.2 Running executables from the command line

As mentioned before, what the `ciaoc` compiler generates and how it is started varies somewhat from OS to OS. In general, the product of compiling an application with `ciaoc` is a file that contains the bytecode (the product of the compilation) and invokes the Ciao engine on it.

- In Unix this is a *script* (see the first lines of the file) which invokes the ciao engine on this file. To run the generated executable from a Unix shell it suffices to type its name at the shell command line, as in the examples above.
- In a Win32 system, the compiler produces a similar file with a `.cpx` ending and an additional `.bat` file.

  The Ciao installation process typically makes sure that the Windows registry contains the right entries so that `.cpx` executables will run upon double-clicking and from a command shell (in NT systems).

  The `.bat` files allow running the Ciao executable from any other processes (which typically does not use the Windows registry).

  Finally, in a system in which Cygwin is installed, executables can also be used directly from the `bash` shell command line, without any associated `.bat` files, by simply typing their name at the `bash` shell command line, in the same way as in Unix.

Except for a couple of header lines, the contents of executables are almost identical under different OSs (except for self-contained ones). The bytecode they contain is architecture-independent. In fact, it is possible to create an executable under Unix and run it on Windows or viceversa, by making only minor modifications (e.g., creating the `.bat` file and/or setting environment variables or editing the start of the file to point to the correct engine location).

13.3 Types of executables generated

While the default options used by `ciaoc` are sufficient for normal use, by selecting other options `ciaoc` can generate several different types of executables, which offer interesting tradeoffs among size of the generated executable, portability, and startup time [CH00b]:

Dynamic executables:
: `ciaoc` produces by default *dynamic* executables. In this case the executable produced is a platform-independent file which includes in compiled form all the user defined files. On the other hand, any system libraries used by the application are loaded dynamically at startup. More precisely, any files that appear as `library(...)` in `use_module/1` and `ensure_loaded/1` declarations will not be included explicitly in the executable and will instead be loaded dynamically. Is is also possible to mark other path aliases (see the documentation for `file_search_path/2`) for dynamic loading by using the `-d` option. Files accessed through such aliases will also be loaded dynamically.

  Dynamic loading allows making smaller executables. Such executables may be used directly in the same machine in which they were compiled, since suitable paths to the location of the libraries will be included as default in the executable by `ciaoc` during compilation.

  The executable can also be used in another machine, even if the architecture and OS are different. The requirement is that the Ciao libraries (which will also include the appropriate Ciao engine for that architecture and OS) be installed in the target machine, and that environment variables are set appropriately for the executable to be able to find them (see ⟨undefined⟩ [Environment variables used by Ciao executables], page ⟨undefined⟩). How to do this differs slightly from OS to OS.

Static executables:
: Selecting the `-s` option `ciaoc` produces a *static* executable. In this case the executable produced (again a platform-independent file) will include in it all the auxiliary files and any system libraries needed by the application. Thus, such an executable is almost complete, needing in order to run only the Ciao engine, which is platform-specific.[1] Again, if the executable is run in the same machine in which it was compiled then the engine is found automatically. If the executable is moved to another machine, the executable only needs access to a suitable engine (which can be done by setting the appropriate environment variables, see ⟨undefined⟩ [Environment variables used by Ciao executables], page ⟨undefined⟩).

  This type of compilation produces larger executables, but has the advantage that these executables can be installed and run in a different machine, with different architecture and OS, even if Ciao is not installed on that machine. To install (or distribute) such an executable, one only needs to copy the executable file itself and the appropriate engine for the target platform (See Chapter 4 [Installing Ciao from the source distribution], page 39 or ⟨undefined⟩ [Installing Ciao from a Win32 binary distribution], page ⟨undefined⟩ and ⟨undefined⟩ [Multiarchitecture installation], page ⟨undefined⟩), and to set things so that the executable can find the engine.[2]

---

[1] Currently there is an exception to this related to libraries which are written in languages other than Prolog, as, e.g., C. C files are currently always compiled to dynamically loadable object files (`.so` files), and they thus need to be included manually in a distribution of an application. This will be automated in upcoming versions of the Ciao system.

[2] It is also possible to produce real standalone executables, i.e., executables that do not need to have an engine around. However, this is not automated yet, although it is planned for an upcoming version of the compiler. In particular, the compiler can generate a `.c` file for each `.pl` file. Then all the `.c` files can be compiled together into a real executable (the engine

Dynamic executables, with lazy loading:
: Selecting the `-l` option is very similar to the case of dynamic executables above, except that the code in the library modules is not loaded when the program is started but rather it is done during execution, the first time a predicate defined in that file is called. This is advantageous if a large application is composed of many parts but is such that typically only some of the parts are used in each invocation. An executable with lazy load has the advantage that it starts fast, loading a minimal functionality on startup, and then loads the different modules automatically as needed.

Self-contained executables:
: *Self-contained* executables are static executables (i.e., this option also implies *static* compilation) which include a Ciao engine along with the bytecode, so they do not depend on an external one for their execution. This is useful to create executables which run even if the machine where the program is to be executed does not have a Ciao engine installed and/or libraries. The disadvantage is that such execuatbles are platform-dependent (as well as larger than those that simply use an external library). This type of compilation is selected with the `-S` option. Cross-compilation is also possible with the `-SS` option, so you can specify the target OS and architecture. To be able to use the latter option, it is necessary to have installed a `ciaoengine` for the target machine in the Ciao library (this requires compiling the engine in that OS/architecture and installing it, so that it is available in the library).

13.4 Intermediate files in the compilation process

Compiling an individual source (i.e., `.pl`) file produces a `.itf` file and a `.po` file. The `.itf` file contains information of the *modular interface* of the file, such as information on exported and imported predicates and on the other modules used by this module. This information is used to know if a given file should be recompiled at a given point in time and also to be able to detect more errors statically including undefined predicates, mismatches on predicate charaterictics across modules, etc. The `.po` file contains the platform-independent object code for a file, ready for linking (statically or dynamically).

It is also possible to use `ciaoc` to explicitly generate the `.po` file for one or more `.pl` files by using the `-c` option.

If you want to view the WAM instructions of one or more `.pl` files you can use the `-w` option. That will generate a `.wam` file with such instructions in a pretty format per each `.pl` file.

13.5 Usage (ciaoc)

The following provides details on the different command line options available when invoking `ciaoc`:

```
ciaoc [Opts] <Files>

Compile the listed files. If there is more than one file, the first
one is considered the main module (it must include the main predicate
when creating an executable).
```

is added one more element during link time) producing a complete executable for a given architecture. The downside of course is that such an executable will not be portable to other architectures without recompilation.

```
The default extension for files is '.pl'.

-h, --help
Show this help.
-u
<File> Use File for compilation, often used to include LibDir paths, etc.
--iso-strict
Turn on stricter ISO compatibility for user files and modules declared with module/2.
-op
<Suffix> Use Suffix as the suffix for optimized (or otherwise tuned) code.
-L <LibDir> Look for libraries also in the LibDir directory.
-c Generate .po objects for the input modules.
-w Generate .wam files (WAM code) for the input modules.
-S
Make standalone executable for the current OS and architecture, implies -s.
-SS
<EngCfg> Make standalone executable for the EngCfg OS and architecture (and
optionally debugging level) (see ciao_sysconf for valid values for Target), implies -s.
-ll <Module> Force Module to be loaded lazily, implies -l.
-ac <Packages> Use Packages for compiling all modules.
-acm
<Module> <Packages> Use the given Packages (term) for compiling Module.
-d <Path> Files using this path alias are dynamic (default: library).
-o <File> Specify output file name.
-v, --verbose-compilation
Verbose mode.
-ri, --itf-format-r
Generate human-readable .itf files.
-x, --check-libraries
Check for changes also in the Ciao standard library during incremental
compilation (for developers).
-s, --executables-static
Make a static executable (otherwise dynamic files are not included).
-l, --executables-lazyload
Idem with lazy load of dynamic files (except insecure cases).
-np, --use-global-module-options-no
Ignore global module options.
-na, --read-assertions-no
Do not read the assertions in the code.
-rc, --runtime-checks
Generate code with runtime checks; requires reading assertions.
--rtchecks-trust-no
Disable rtchecks for trust assertions.
--rtchecks-entry-no
Disable rtchecks for entry assertions.
--rtchecks-exit-no
Disable rtchecks for exit assertions.
--rtchecks-test
Enable rtchecks for test assertions (for debugging
purposes only, unittest library is recommended).
--rtchecks-level-exports
```

```
Use rtchecks only for external calls of the exported predicates.
--rtchecks-asrloc-no
Do not use assertion locators in the error messages.
--rtchecks-predloc-no
Do not use predicate locators in the error messages.
--rtchecks-namefmt-short
Show the name of predicates and properties in a reduced format.
--rtchecks-callloc-no
Do not show the stack of predicates that caused the failure.
--rtchecks-callloc-literal
Show the stack of predicates that caused the failure. Instrument it
in the literal. This mode provides more information, because reports
also the literal in the body of the predicate.
```

14 The script interpreter

Author(s): Daniel Cabeza, Manuel Hermenegildo, Jose F. Morales (minor).

`ciao-shell` is the Ciao script interpreter. It can be used to write *Prolog shell scripts* (see [Her96]), that is, executable files containing source code, which are compiled on demand.

Writing Prolog scripts can sometimes be advantageous with respect to creating binary executables for small- to medium-sized programs that are modified often and perform relatively simple tasks. The advantage is that no explicit compilation is necessary, and thus changes and updates to the program imply only editing the source file. The disadvantage is that startup of the script (the first time after it is modified) is slower than for an application that has been compiled previously.

14.1 How it works

Essentially, `ciao-shell` is a smaller version of the Ciao top-level, which starts by loading the file given to it as the first argument and then starts execution at `main/1` (the argument is instantiated to a list containing the command line options, in the usual way). Note that the Prolog script cannot have a `module` declaration for this to work. While loading the file, `ciao-shell` changes the prolog flag `quiet` so that no informational or warning messages are printed (error messages will be reported to `user_error`, however). The operation of `ciao-shell` in Unix-like systems is based in a special compiler feature: when the first character of a file is '`#`', the compiler skips the first lines until an empty line is found. In Windows, its use is as easy as naming the file with a `.pls` extension, which will launch `ciao-shell` appropriately.

For example, in a Linux/Unix system, assume a file called `hello` contains the following program:

```
#!/usr/bin/env ciao-shell
% -*- mode: ciao; -*-

main(_) :-
     write('Hello world'), nl.
```

Then, the file `hello` can be *run* by simply making it executable and invoking it from the command line:

```
$ chmod +x hello
$ ./hello
Hello world
```

The lines:

```
#!/usr/bin/env ciao-shell
% -*- mode: ciao; -*-
```

invokes `ciao-shell` through `/usr/bin/env` (POSIX.2 compliant), instructing it to read this same file, and passing it the rest of the arguments to `hello` as arguments to the Prolog program. The second line `% -*- mode: ciao; -*-` is simply a comment which is seen by `emacs` and instructs it to edit this file in Ciao mode (this is needed because these script files typically do not have a `.pl` ending). When `ciao-shell` starts, if it is the first time, it compiles the program (skipping the first lines, as explained above), or else at successive runs loads the `.po` object file, and then calls `main/1`.

Note that the process of creating Prolog scripts is made very simple by the Ciao emacs mode, which automatically inserts the header and makes the file executable (See Chapter 19 [Using Ciao inside GNU emacs], page 97).

14.2 Command line arguments in scripts

The following example illustrates the use of command-line arguments in scripts. Assume that a file called `say` contains the following lines:

```
#!/usr/bin/env ciao-shell
% -*- mode: ciao; -*-

main(Argv) :-
     write_list(Argv), nl.

write_list([]).
write_list([Arg|Args]) :-
     write(Arg),
     write(' '),
     write_list(Args).
```

An example of use is:

```
$ say hello dolly
hello dolly
```

15 Other miscellaneous standalone utilities

This is the documentation for a set of other miscellaneous standalone utilities that can be used as part of the development environment.

16 Display information about Ciao object files

Author(s): Manuel Hermenegildo, Daniel Cabeza, Manuel Carro, Jose F. Morales.

This program outputs in symbolic form the information contained in a Ciao object file produced by the Ciao compiler. This information includes:

- declarations and code after compiler's pass one: assertion information (predicate declarations, property declarations, type declarations, etc.) and code-related information (imports, exports, libraries used, etc.)
- contents of a module interface file (`.itf`)
- contents of a bytecode file (`.po`), include WAM bytecode
- plain terms in `fastrw` format

This program is specially useful for example for checking what assertions the assertion normalizer is producing from the original assertions in the file, what the compiler is actually seeing after some of the syntactic expansions (but before goal translations), or what bytecode instructions are generated.

16.1 Usage

```
ciaodump <options> <filename>

where the possible options are:

  -h
     Print this information

  -itf <filename>[.itf]
     Print .itf contents in symbolic form

  -asr0 <filename>[.asr]
     Print .asr contents in symbolic form (raw form)

  -asr <filename>[.asr]
     Pretty print .asr contents in symbolic form

  -fastrw <filename>
     Decode a file written with the fastrw library

  -norm [-v] [-m] <-a|-f|-c|-e> <filename>[.pl]
     Print output of compiler's pass one

     -v : verbose output (e.g., lists all files read)

     -m : restrict info to current module (omit imported)

     -a : print assertions (only code-oriented assertions -- not
      comment-oriented assertions), after normalization

     -f : print declarations, code, and interface (imports/exports, etc.)
      after compiler's pass one
```

```
   -c : print code only

   -e : print only errors - useful to check syntax of assertions in file

   <filename> must be the name of a Ciao source file.

-po <filename>[.po]
   Print .po contents in symbolic form

-wam <filename>[.pl]
   Print WAM code (simplified) for a .pl file
```

17 callgraph (library)

17.1 Usage and interface

- **Library usage:**
  `:- use_module(library(xrefs/callgraph)).`
- **Exports:**
  - *Predicates:*
    `call_graph/2`, `reachability/4`.

17.2 Documentation on exports

call_graph/2: PREDICATE

Usage: `call_graph(File,Graph)`

`Graph` is the call-graph of the code in `File`.

- *The following properties should hold at call time:*
  `File` is an atom describing the name of a file. (`c_itf_props:filename/1`)
  `Graph` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Graph` is an ugraph. (`ugraphs:ugraph/1`)

reachability/4: PREDICATE

Usage: `reachability(Graph,Sources,Reached,UnReached)`

`Reached` are the vertices in `Graph` reachable from `Sources`, `UnReached` are the rest.

- *The following properties should hold at call time:*
  `Graph` is an ugraph. (`ugraphs:ugraph/1`)
  `Sources` is a list. (`basic_props:list/1`)
  `Reached` is a free variable. (`term_typing:var/1`)
  `UnReached` is a free variable. (`term_typing:var/1`)

17.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `c_itf_props`, `sets`, `terms`, `ugraphs`, `xrefsread`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

18 Finding differences between two Prolog files

Author(s): Francisco Bueno.

This simple program works like the good old diff but for files that contain Prolog code. It prints out the clauses that it finds are different in the files. Its use avoids textual differences such as different variable names and different formatting of the code in the files.

18.1 Usage (pldiff)

```
pldiff <file1> <file2>
   : find differences

pldiff -h
   : print this information
```

but you can also use the program as a library and invoke the predicate:

```
pldiff( <filename> , <filename> )
```

19 Using Ciao inside GNU emacs

Author(s): Manuel Hermenegildo, Manuel C. Rodriguez, Daniel Cabeza.

The Ciao emacs interface (or *mode* in `emacs` terms) provides a rich, integrated user interface to the Ciao *program development environment* components, including the `ciaosh` interactive top level, the `lpdoc` documentation generator, the testing system, and the `ciaopp` preprocessor. Most features of these Ciao development environment components are available from the command line of the top-level shell and the preprocessor and as standalone tools. However, using Ciao from inside `emacs` is highly recommended. The facilities that this mode provides include:

- *Syntax-based highlighting* (coloring), *auto-indentation*, *auto-fill*, etc. of code. This includes the assertions used by the preprocessor and the documentation strings used by the Ciao auto-documenter, `lpdoc`.
- Providing automatic access to on-line help for all predicates by accessing the Ciao system manuals in `info` format.
- Starting and communicating with the *Ciao top-level*, running in its own sub-shell. This facilitates loading programs, checking the *syntax* of programs (and of *assertions* within programs), marking and unmarking modules for interactive debugging, *tracing the source code* during debugging, making standalone executables, compiling modules to dynamically linkable Ciao objects, compiling modules to active objects, etc.
- Starting and communicating with `lpdoc`, the *Ciao auto-documenter*, running in its own sub-shell. This allows generating in a very convenient way manuals for any file(s) being edited, in a variety of output formats, and is very useful for quickly checking how the auto-generated documentation will look.
- Running unit tests on files or applications.
- Starting and communicating with `ciaopp`, the *Ciao preprocessor*, running in its own sub-shell. This allows easily performing certain kinds of *static checks* (useful for finding errors in programs before running them), program analysis tasks, and *program transformations* on source programs.
- Syntax highlighting and coloring of the error and warning messages produced by the top level, unit testing, preprocessor, or any other tool using the same message format (such as the `lpdoc` auto-documenter), and *locating automatically the points in the source files where such errors occur*.
- This mode also includes a very simple automatic *version control* system which allows keeping a *changelog* for individual files or for whole applications. This is done by automatically including changelog entries in source files, which can then be processed by the `lpdoc` auto-documenter. This is useful for smaller projects that are not stored in a repository and can also be used for maintaining changelogs even for projects that are repository-based.

This chapter explains how to use the Ciao `emacs` interface (and how to set up your `emacs` environment for correct operation, even though this is normally done automatically by the installation process). The Ciao `emacs` interface can also be used to work with traditional Prolog or CLP systems.

19.1 Conventions for writing Ciao programs under Emacs

There are currently a number of syntactic conventions for Ciao programs which greatly help operation of the Emacs development environment. These conventions are particularly important for the source-level debugger and the syntax-based coloring capabilities. The need for such conventions comes from the fact that it would be unrealistic to write a complete Ciao parser in Emacs lisp. These conventions are the following, more or less in order of importance:

- Clauses should begin on the first column (this is used to recognize the beginning of a clause).

- C style comments should not be used in a clause, but can be used outside any clause.

The following suggestions are not strictly necessary but can improve operation. In particular, they allow much greater precision in the location of program points during source-level debugging (for line by line tracing, when marking breakpoints, etc.):

- Body literals should be indented.
- There should be no more than one literal per line.

Other issues:

- Comments which start with `%`s are indented to the right if indentation is requested.
- For syntax-based highlighting to be performed, font-lock must be available and not disabled (the Ciao mode enables it but it may be disabled elsewhere in, e.g., the `.emacs` file).

19.2 Checking the installation

Typically, a complete pre-installation of the Ciao `emacs` interface is performed during Ciao installation. To check that installation was completed sucessfully, open a file with a `.pl` ending. You should see that `emacs` enters Ciao mode: the mode is identified in the status bar below the buffer and, if the emacs menu bar is enabled, you should see the Ciao menus. You should be able from the menu-bar, for example, to go to the Ciao manuals in the info or load the `.pl` file that you just opened into a Ciao top level.

If things don't work properly, see the section ⟨undefined⟩ [Installation of the Ciao emacs interface], page ⟨undefined⟩ later in this chapter.

19.3 Functionality and associated key sequences (bindings)

The following sections summarize the capabilities of the Ciao emacs interface and the (default) *key sequences* used to access those capabilities. Note however that most of these functions are also accessible from the menu bar, so learning these key combinations is not necessary: the list is provided mainly for illustration of the capabilities available, as well as completeness and documentation.

When referring to keyboard characters, printing characters are written thus: ⟨a⟩, while control characters are written like this: ⟨^A⟩. Thus ⟨^C⟩ is the character you get by holding down the ⟨CTL⟩ key while you type ⟨c⟩. Finally, the special control characters carriage-return, line-feed and space are often abbreviated to ⟨RET⟩, ⟨LFD⟩ and ⟨SPC⟩ respectively.

19.4 Syntax coloring and syntax-based editing

Syntax-based highlighting (coloring) of code is provided automatically when opening Ciao files. This includes also the assertions used by the preprocessor and the documentation strings used by the Ciao auto-documenter, `lpdoc`. The mode should be set to Ciao and the Ciao mode menus should appear on the menu bar. The colors and fonts used can be changed through the *customize* options in the help menu (see ⟨undefined⟩ [Customization], page ⟨undefined⟩).

During editing this coloring may be refreshed by calling the appropriate function (see below).

Limited syntax-based auto-indentation and auto-fill of code and comments is also provided. Syntax highlighting and coloring is also available for the error and warning messages produced by the top level, preprocessor, and auto-documenter, and, in general, for the output produced by these tools.

Commands:

⟨TAB⟩ Indent current line as Ciao code. With argument, indent any additional lines of the same clause rigidly along with this one.

19.5 Getting on-line help

The following commands are useful for getting on-line help. This is done by accessing the `info` version of the Ciao manuals or the `emacs` built-in help strings. Note also that the `info` standard `search` command (generally bound to ⟨s⟩) can be used inside `info` buffers to search for a given string.

⟨C-h⟩ ⟨S⟩ Display the definition of a Ciao SYMBOL, as found in the relevant manual.

⟨C-h⟩ ⟨M-s⟩ Grovel indices from Ciao manuals for STRING. Build a menu of the possible matches.

⟨C-h⟩ ⟨TAB⟩ Completions at point for ciao-mode using info-look.

⟨C-c⟩ ⟨RET⟩ Go to the part of the info directory containing the Ciao manuals.

⟨C-h⟩ ⟨m⟩ Show a short description of the Ciao Emacs mode, including all key bindings.

19.6 Loading and compiling programs

These commands allow *loading programs*, *creating executables*, etc. by issuing the appropriate commands to a Ciao top level shell, running in its own buffer as a subprocess. See Chapter 9 [The interactive top-level shell], page 55 for details. The following commands implement the communication with the Ciao top level:

⟨C-c⟩ ⟨t⟩ Ensure that an inferior Ciao top-level process is running.

This opens a top-level window (if one did not exist already) where queries can be input directly. Programs can be loaded into this top level by typing the corresponding commands in this window (such as use_module, etc.), or, more typically, by opening the file to be loaded in an emacs window (where it can be edited) and issuing a load command (such as C-c l or C-c L) directly from there (see the loading commands of this mode and their bindings).

Note that many useful commands (e.g., to repeat and edit previous commands, interrupt jobs, locate errors, automatic completions, etc.) are available in this top-level window (see ⟨undefined⟩ [Commands available in toplevel and preprocessor buffers], page ⟨undefined⟩).

Often, it is not necessary to use this function since execution of any of the other functions related to the top level (e.g., loading buffers into the top level) ensures that a top level is started (starting one if required).

⟨C-c⟩ ⟨l⟩ Load the current buffer (and any auxiliary files it may use) into the top level.

The type of compilation performed (*compiling* or *interpreting*) is selected automatically depending on whether the buffer has been marked for debugging or not – see below. In case you try to load a file while in the middle of the debugging process the debugger is first aborted and then the buffer is loaded. Also, if there is a defined query, the user is asked whether it should be called.

⟨C-c⟩ ⟨f⟩ Load CiaoPP and then the current buffer (and any auxiliary files it may use) into the top level. Use CiaoPP auto_check_assrt predicate to check current buffer assertions and then load the buffer if there was no error.

⟨C-c⟩ ⟨x⟩ Make an executable from the code in the current buffer. The buffer must contain a `main/0` or `main/1` predicate. Note that compiler options can be set to determine whether the libraries and auxiliary files used by the executable will be statically linked, dynamically linked, auto-loaded, etc.

⟨C-c⟩ ⟨o⟩ Make a Ciao object (.po) file from the code in the current buffer. This is useful for example while debugging during development of a very large application which is compiled into an excutable, and only one or a few files are modified. If the application executable is dynamically linked, i.e., the component .po files are loaded dynamically during startup of the application, then this command can be used to recompile only the file or files which have changed, and the correct version will be loaded dynamically the next time the application is started. However, note that this must be done with care since it only works if the inter-module interfaces have not changed. The recommended, much safer way is to generate the executable again, letting the Ciao compiler, which is inherently incremental, determine what needs to be recompiled.

⟨C-c⟩ ⟨s⟩ Set the current buffer as the principal file in a multiple module programming environment.

⟨C-c⟩ ⟨L⟩ Load the module designated as *main module* (and all related files that it uses) into the top level. If no main module is defined it will load the current buffer.

The type of compilation performed (*compiling* or *interpreting*) is selected automatically depending on whether the buffer has been marked for debugging or not – see below. In case you try to load a file while in the middle of the debugging process the debugger is first aborted and then the buffer is loaded. Also, if there is a defined query, the user is asked whether it should be called.

19.7 Commands available in toplevel and preprocessor buffers

The interactive top level and the preprocessor both are typically run in an iteractive buffer, in which it is possible to communicate with them in the same way as if they had been started from a standard shell. These interactive buffers run in the so-called *Ciao inferior mode*. This is a particular version of the standard emacs shell package (comint) and thus all the commands typically available when running shells inside emacs also work in these buffers. In addition, many of the commands and key bindings available in buffers containing Ciao source code are also available in these interactive buffers, when applicable. The Ciao-specific commands available include:

⟨C-h⟩ ⟨S⟩ Display the definition of a Ciao SYMBOL, as found in the relevant manual.

⟨C-h⟩ ⟨M-s⟩ Grovel indices from Ciao manuals for STRING. Build a menu of the possible matches.

⟨C-c⟩ ⟨TAB⟩ Completions at point for ciao-mode using info-look.

⟨M-]⟩ Go to the location in the source file containing the next error reported by the last Ciao subprocess (preprocessor or toplevel) which was run.

⟨M-[⟩ Go to the location in the source file containing the previous error reported by the last Ciao subprocess (preprocessor or toplevel) which was run.

⟨C-c⟩ ⟨‘⟩ Go to the location in the source file containing the next error reported by the last Ciao subprocess (preprocessor or toplevel) which was run.

⟨C-c⟩ ⟨e⟩ Remove error marks from last run (and also debugging marks if present). This finish the error finding session.

⟨C-g⟩ Same as keyboard-quit (i.e., the normal Emacs quit, typically bound to C-g) but also clears the error markings from the last Ciao-related inferior process run.

⟨C-c⟩ ⟨q⟩ Set a default query. This may be useful specially during debugging or testing sessions. However, as mentioned elsewhere, note that commands that repeat previous queries are also available.

This query can be recalled at any time using C-c Q. It is also possible to set things up so that this query will be issued automatically any time a program is (re)loaded. The functionality is available in the major mode (i.e., from a buffer containing a source file) and in the inferior mode (i.e., from the buffer running the top-level shell).

When called from the major mode (i.e., from window containing a source file) then the user is prompted in the minibuffer for the query. When called from the inferior mode (i.e., from a top-level window) then the query on the current line, following the Ciao prompt, is taken as the default query.

To clear the default query use M-x ciao-clear-query or simply set it to an empty query: i.e., in a source buffer select C-c q and enter an empty query. In an inferior mode simply select C-c q on a line that contains only the system prompt.

⟨C-c⟩ ⟨Q⟩ Issue predefined query.

⟨C-c⟩ ⟨C-v⟩ Show last output file produced by CiaoPP. The preprocessor works by producing a file which is a transformed and/or adorned (with assertions) version of the input file. This command is often used after running the preprocessor in order to visit the output file and see the results from running the preprocessor.

⟨C-c⟩ ⟨v⟩ Report the version of the emacs Ciao mode.

The following are some of the commands from the comint shell package which may be specially useful (type `<f1> m` while in a Ciao interactive buffer for a complete list of commands):

⟨M-p⟩ Cycle backwards through input history, saving input.

(fn ARG)

⟨M-n⟩ Cycle forwards through input history.

(fn ARG)

⟨M-r⟩ Search for a regular expression backward in input history using Isearch.

⟨TAB⟩ Dynamically find completion of the item at point. Note that this completion command refers generally to filenames (rather than, e.g., predicate names, as in the previous functions).

⟨M-?⟩ List all (filename) completions of the item at point.

⟨RET⟩ Return at any point of the a line at the end of a buffer sends that line as input. Return not at end copies the rest of the current line to the end of the buffer and sends it as input.

⟨^D⟩ Delete ARG characters forward or send an EOF to subprocess. Sends an EOF only if point is at the end of the buffer and there is no input.

(fn ARG)

⟨^C⟩ ⟨^U⟩ Kill all text from last stuff output by interpreter to point.

⟨^C⟩ ⟨^W⟩ Kill characters backward until encountering the beginning of a word. With argument ARG, do this that many times.

(fn ARG)

⟨^C⟩ ⟨^C⟩ Interrupt the current subjob.

⟨^C⟩ ⟨^Z⟩ Stop the current subjob.

WARNING: if there is no current subjob, you can end up suspending the top-level process running in the buffer. If you accidentally do this, use M-x comint-continue-subjob to resume the process. (This is not a problem with most shells, since they ignore this signal.)

⟨^C⟩ ⟨^\⟩ Send quit signal to the current subjob.

19.8 Locating errors and checking the syntax of assertions

These commands allow locating quickly the point in the source code corresponding to errors flagged by the compiler or preprocessor as well as performing several syntactic checks of assertions:

⟨M-]⟩ Go to the location in the source file containing the next error reported by the last Ciao subprocess (preprocessor or toplevel) which was run.

⟨M-[⟩ Go to the location in the source file containing the previous error reported by the last Ciao subprocess (preprocessor or toplevel) which was run.

⟨C-c⟩ ⟨‘⟩ Go to the location in the source file containing the next error reported by the last Ciao subprocess (preprocessor or toplevel) which was run.

⟨C-c⟩ ⟨e⟩ Remove error marks from last run (and also debugging marks if present). This finish the error finding session.

⟨C-g⟩ Same as keyboard-quit (i.e., the normal Emacs quit, typically bound to C-g) but also clears the error markings from the last Ciao-related inferior process run.

19.9 Commands which help typing in programs

The following commands are intended to help in the process of writing programs:

⟨C-c⟩ ⟨I⟩ ⟨S⟩ Insert a (Unix) header at the top of the current buffer so that the Ciao script interpreter will be called on this file if *run* from the command line. It also makes the file executable (e.g., `chmod +x <file>` in Unix). See Chapter 14 [The script interpreter], page 87 for details.

⟨C-c⟩ ⟨C-f⟩ Format a Ciao program using ciaofmt.

19.10 Debugging programs

These commands allow marking modules for *debugging* by issuing the appropiate commands to a Ciao top level shell, running in its own buffer as a subprocess. There are two differents types of debugging: traditional debugging (using the byrd-box model and spy-points) and *source-level debugging* (same as traditional debugging plus source tracing and breakpoints). In order to use *breakpoints*, source debugging must be on. The following commands implement comunication with the Ciao top level:

⟨C-c⟩ ⟨d⟩ Debug (or stop debugging) buffer source. This is a shortcut which is particularly useful when using the source debugger on a single module. It corresponds to several lower-level actions. Those lower-level actions depend on how the module was selected for debugging. In case the module was not marked for source-level debugging, it marks the module corresponding to the current buffer for source-level debugging, reloads it to make sure that it is loaded in the correct way for debugging (same as C-c l), and sets the debugger in trace mode (i.e., issues the `trace.` command to the top-level shell). Conversely, if the module was already marked for source-level debugging then it will take the opposite actions, i.e., it unmarks the module for source-level debugging, reloads it, and sets the debugger to non-debug mode.

⟨C-c⟩ ⟨m⟩ Mark, or unmark, the current buffer for debugging (traditional debugging or source debugging). Note that if the buffer has already been loaded while it was unmarked for debugging (and has therefore been loaded in compile mode) it has to be loaded again. The minibuffer shows how the module is loaded now and allows selecting another mode for it. There are three posibilities: N for no debug, S for source debug and D for traditional debug.

⟨C-c⟩ ⟨M-m⟩ Visits all Ciao files which are currently open in a buffer allowing selecting for each of them whether to debug them or not and the type of debugging performed. When working on a multiple module program, it is possible to have many modules open at a time. In this case, you will navigate through all open Ciao files and select the debug mode for each of them (same as doing C-c m for each).

⟨C-c⟩ ⟨S⟩ ⟨b⟩ Set a breakpoint on the current literal (goal). This can be done at any time (while debugging or not). The cursor must be *on the predicate symbol of the literal*. Breakpoints are only useful when using source-level debugging.

⟨C-c⟩ ⟨S⟩ ⟨v⟩ Remove a breakpoint from the current literal (goal). This can be done at any time (while debugging or not). The cursor must be *on the predicate symbol of the literal*.

⟨C-c⟩ ⟨S⟩ ⟨n⟩ Remove all breakpoints. This can be done at any time (while debugging or not).

⟨C-c⟩ ⟨S⟩ ⟨l⟩ Redisplay breakpoints in all Ciao buffers. This ensures that the marks in the source files and the Ciao toplevel are synchronized.

⟨C-c⟩ ⟨S⟩ ⟨r⟩ Remove breakpoint coloring in all Ciao files.

⟨C-c⟩ ⟨S⟩ ⟨t⟩ Set the debugger to the trace state. In this state, the program is executed step by step.

⟨C-c⟩ ⟨S⟩ ⟨d⟩ Set the debugger to the debug state. In this state, the program will only stop in breakpoints and spypoints. Breakpoints are specially supported in `emacs` and using source debug.

⟨C-c⟩ ⟨r⟩ Load the current region (between the cursor and a previous mark) into the top level. Since loading a region of a file is typically done for debugging and/or testing purposes, this command always loads the region in debugging mode (interpreted).

⟨C-c⟩ ⟨p⟩ Load the predicate around the cursor into the top level. Since loading a single predicate is typically done for debugging and/or testing purposes, this command always loads the predicate in debugging mode (interpreted).

19.11 Testing programs

These commands allow testing predicates and modules, based on interactively defined queryies or more sophisticated tests specified within the source code.

⟨C-c⟩ ⟨q⟩ Set a default query. This may be useful specially during debugging or testing sessions. However, as mentioned elsewhere, note that commands that repeat previous queries are also available.

This query can be recalled at any time using C-c Q. It is also possible to set things up so that this query will be issued automatically any time a program is (re)loaded. The functionality is available in the major mode (i.e., from a buffer containing a source file) and in the inferior mode (i.e., from the buffer running the top-level shell). When called from the major mode (i.e., from window containing a source file) then the user is prompted in the minibuffer for the query. When called from the inferior mode (i.e., from a top-level window) then the query on the current line, following the Ciao prompt, is taken as the default query.

To clear the default query use M-x ciao-clear-query or simply set it to an empty query: i.e., in a source buffer select C-c q and enter an empty query. In an inferior mode simply select C-c q on a line that contains only the system prompt.

⟨C-c⟩ ⟨Q⟩ Issue predefined query.

⟨C-c⟩ ⟨u⟩ Run the tests in the current buffer.

The tests should be specified using test assertions in the module.

⟨C-c⟩ ⟨U⟩ Run the tests in the current buffer and check the assertions of exported predicates.

The tests should be specified using test assertions in the module.

19.12 Preprocessing programs

These commands allow *preprocessing programs* with `ciaopp`, the *CiaoPP Program Processor*.

See the preprocessor manual for details. The following commands implement the communication with CiaoPP:

⟨C-c⟩ ⟨A⟩ Call the preprocessor to perform a number of pre-selected analyses on the current buffer (and related modules).

⟨C-c⟩ ⟨V⟩ Call the preprocessor to perform compile-time checking of the assertions (types, modes, determinacy, nonfailure, cost, ...) in the current buffer (and against those in related modules).

⟨C-c⟩ ⟨O⟩ Uses the preprocessor to perform optimizations (partial evaluation, abstract specialization, parallelization, ...) on the current buffer (and related modules).

⟨C-c⟩ ⟨M⟩ Browse and select (using the preprocessor menus) the actions to be performed by the preprocessor when performing analisys used by M-x ciao- C-c A, C-c V, C-c O, and the corresponding toolbar buttons.

⟨C-c⟩ ⟨C-v⟩ Show last output file produced by CiaoPP. The preprocessor works by producing a file which is a transformed and/or adorned (with assertions) version of the input file. This command is often used after running the preprocessor in order to visit the output file and see the results from running the preprocessor.

⟨C-c⟩ ⟨C-r⟩ Ensure that an inferior CiaoPP process is running.

This opens a preprocessor top-level window (if one did not exist already) where preprocessing commands and preprocessing menu options can be input directly. Programs can be preprocessed by typing commands in this window, or, more typically, by opening the file to be preprocessed in an emacs window (where it can be edited) and issuing a command (such as C-c A, C-c V, C-c O, or C-c M) directly from there (see the preprocessing commands of this mode and their bindings).

Note that many useful commands (e.g., to repeat and edit previous commands, interrupt jobs, locate errors, automatic completions, etc.) are available in this top-level window (see ⟨undefined⟩ [Commands available in toplevel and preprocessor buffers], page ⟨undefined⟩).

Often, it is not necessary to use this function since execution of any of the other functions related to the top level (e.g., loading buffers into the top level) ensures that a top level is started (starting one if required).

19.13 Version control

The following commands can be used to carry out a simple but effective form of version control by keeping a log of changes on a file or a group of related files. Interestingly, this log is kept in a format that is understood by `lpdoc`, the Ciao documenter [Her99]. As a result, if these version comments are present, then `lpdoc` will be able to automatically assign up to date version numbers to the manuals that it generates. This way it is always possible to identify to which version of the software a manual corresponds. Also, `lpdoc` can create automatically sections describing the changes made since previous versions, which are extracted from the comments in the changelog entries.

The main effect of these commands is to automatically associate the following information to a set of changes performed in the file and/or in a set of related files:

- a *version number* (such as, e.g., `1.2`, where `1` is the major version number and `2` is the minor version number),
- a patch number (such as, e.g., the `4` in `1.2#4`),
- a time stamp (such as, e.g., `1998/12/14,17:20*28+MET`),
- the author of the change, and
- a comment explaining the change.

The version numbering used can be local to a single file or common to a number of related files. A simple version numbering policy is implemented: when a relevant change is made, the user typically inserts a changelog entry for it, using the appropriate command (or selecting the corresponding option when prompted while saving a file). This will cause the *patch number* for the file (or for the whole system that the file is part of) to be incremented automatically and the corresponding machine-readable comment to be inserted in the file. Major and minor version numbers can also be changed, but this is always invoked by hand (see below).

The changelog entry is written in the form of a `comment/2` declaration. As mentioned before, the advantage of using this kind of changelog entries is that these declarations can be processed by the `lpdoc` automatic documenter (see the `lpdoc` reference manual [Her99] or the `assertions` library documentation for more details on these declarations).

Whether the user is asked or not to introduce such changelog entries, and how the patch and version numbers should be increased is controlled by the presence in the file of a `comment/2` declaration of the type:

```
:- doc(version_maintenance,<type>).
```

(note that this requires including the `assertions` library in the source file). These declarations themselves are also typically introduced automatically when using this mode (see below).

The version maintenance mode can also be set alternatively by inserting a comment such as:

```
%% Local Variables:
%% mode: ciao
%% update-version-comments: "off"
%% End:
```

The lines above instruct emacs to put the buffer visiting the file in emacs Ciao mode and to turn version maintenance off. Setting the version maintenance mode in this way has the disadvantage that `lpdoc`, the auto-documenter, and other related tools will not be aware of the type of version maintenance being performed (the lines above are comments for Ciao). However, this can be useful in fact for setting the *version maintenance mode for packages* and other files meant for inclusion in other files, since that way the settings will not affect the file in which the package is included.

The following commands implement the version control support:

⟨C-c⟩ ⟨C-a⟩ Used to turn on or off version control for the file being visited in the current buffer. The user will be prompted to choose among the following options:

- ⟨y⟩ Turn version control on for this file.
- ⟨n⟩ Turn version control off for this file. A version control comment such as:

  ```
  :- doc(version_maintenance,off).
  ```

  will be added to the buffer and the file saved. No version control will be performed on this file until the line above is removed or modified (i.e., from now on C-x C-s simply saves the buffer).
- ⟨q⟩ Turn off prompting for the introduction of changelog entries for now. `emacs` will not ask again while the buffer is loaded, but it may ask again when saving after the next time you load the buffer (if `ciao-ask-for-version-maintenance-type` is set to `yes`).

If ⟨y⟩ is selected, then the system prompts again regarding how and where the version and patch number information is to be maintained. The following options are available:

`on` All version control information will be contained within this file. When saving a buffer (C-x C-s) emacs will ask if a changelog entry should be added to the file before saving. If a comment is entered by the user, a new patch number is assigned to it and the comment is added to the file. This patch number will be the one that follows the most recent changelog entry already in the file. This is obviously useful when maintaining version numbers individually for each file.

`<directory_name>`
Global version control will be performed coherently on several files. When saving a buffer (C-x C-s) emacs will ask if a changelog entry should be added to the file before saving. If a comment is given, the global patch number (which will be kept in the file: `<directory_name>/GlobalPatch`) is atomically incremented and the changelog entry is added to the current file, associated to that patch number. Also, a small entry is added to a file `<directory_name>/GlobalChangeLog` which points to the current file. This allows inspecting all changes sequentially by visiting all the files where the changes were made (see C-c C-n). This is obviously useful when maintaining a single thread of version and patch numbers for a set of files.

`off` Turns off version control: C-x C-s then simply saves the file as usual.

Some useful tips:

- If a changelog entry is in fact introduced, the cursor is left at the point in the file where the comment was inserted and the mark is left at the original file point. This allows inspecting (and possibly modifying) the changelog entry, and then returning to the original point in the file by simply typing C-x C-x.
- The first changelog entry is entered by default at the end of the buffer. Later, the changelog entries can be moved anywhere else in the file. New changelog entries are always inserted just above the first changelog entry which appears in the file.
- The comments in changelog entries can be edited at any time.
- If a changelog entry is moved to another file, and version numbers are shared by several files through a directory, the corresponding file pointer in the `<directory_name>/GlobalChangeLog` file needs to be changed also, for the entry to be locatable later using C-c C-n.

⟨C-x⟩ ⟨C-s⟩ This is the standard `emacs` command that saves a buffer by writing the contents into the associated `.pl` file. However, in the Ciao mode, if version control is set to on for ths file, then this command will ask the user before saving whether to introduce a changelog entry documenting the changes performed.

In addition, if:

- the buffer does not already contain a comment specifying the type of version control to be performed,
- and the customizable variable `ciao-ask-for-version-maintenance-type` is set to `yes` (go to the Ciao options menu, LPdoc area to change this, which is by default set to `no`),

then, before saving a buffer, the user will be also automatically asked to choose which kind of version control is desired for the file, as in C-c C-a.

⟨C-c⟩ ⟨C-s⟩ Same as C-x C-s except that it forces prompting for inclusion of a changelog entry even if the buffer is unmodified.

⟨C-c⟩ ⟨n⟩ Force a move to a new major/minor version number (the user will be prompted for the new numbers). Only applicable if using directory-based version maintenance. Note that otherwise it suffices with introducing a changelog entry in the file and changing its version number by hand.

⟨C-c⟩ ⟨C-n⟩ When a unique version numbering is being maintained across several files, this command allows inspecting all changes sequentially by visiting all the files in which the changes were made:

- If in a source file, find the next changelog entry in the source file, open in another window the corresponding `GlobalChangeLog` file, and position the cursor at the corresponding entry. This allows browsing the previous and following changes made, which may perhaps reside in other files in the system.
- If in a `GlobalChangeLog` file, look for the next entry in the file, and open in another window the source file in which the corresponding comment resides, positioning the corresponding comment at the top of the screen. This allows going through a section of the `GlobalChangeLog` file checking all the corresponding comments in the different files in which they occur.

19.14 Generating program documentation

These commands provide some bindings and facilities for generating and viewing the documentation corresponding to the current buffer. The documentation is generated in a temporary directory, which is created automatically. This is quite useful while modifying the documentation for a file, in order to check the output that will be produced, whithout having to set up a documentation directory by hand or to regenerate a large manual of which the file may be a part.

⟨C-c⟩ ⟨D⟩ ⟨B⟩ Generate the documentation for the current buffer in the default format. This allows generating a simple document for the current buffer. Basically, it creates a simple, default `SETTINGS.pl` file, sets `mainfile` in `SETTINGS.pl` to the current buffer file and then generates the documentation in a temporary directory. This is useful for seeing how the documentation of a file will format. Note that for generating manuals the best approach is to set up a permanent documentation directory with the appropriate `SETTINGS.pl` file (see the LPdoc manual).

⟨C-c⟩ ⟨D⟩ ⟨F⟩ Change the default output format used by the LPdoc auto-documenter. It is set by default to `html` or to the environment variable `LPDOCFORMAT` if it is defined.

⟨C-c⟩ ⟨D⟩ ⟨S⟩ Visit, or create, the `SETTINGS.pl` file (which controls all auto-documenter options) for the current buffer.

⟨C-c⟩ ⟨D⟩ ⟨G⟩

Generate the documentation according to `SETTINGS.pl` in the default format. This allows generating complex documents but it assumes that `SETTINGS.pl` exists and that the options that it contains (main file, component files, paths, etc.) have been set properly. Documentation is generated in a temporary directory. Note however that for generating complex manuals the best approach is to set up a permanent documentation directory with the appropriate `SETTINGS.pl` (see the LPdoc manual).

⟨C-c⟩ ⟨D⟩ ⟨V⟩

Start a viewer on the documentation for the current buffer in the default format.

⟨C-c⟩ ⟨D⟩ ⟨W⟩
: Change the root directory of the scratchpad for temporary source files. This is used, e.g., by the LPdoc auto-documenter when generating temporary configuration files and documentation for buffers. It is set by default to a new dir under `/tmp` or to the environment variable `CIAOSCRATCHDIR` if it is defined.

19.15 Setting top level preprocessor and documenter executables

These commands allow *changing the executables used* when starting the top-level, the preprocessor, or the auto-documenter. They also allow changing the arguments that these executables take, and changing the path where the libraries reside. In the case of the top-level and preprocessor, this should be done only by users which understand the implications, but it is very useful if several versions of Ciao or the preprocessor are available in the system. All these settings can be changed through the *customize* options in the help menu (see ⟨undefined⟩ [Customization], page ⟨undefined⟩).

⟨C-c⟩ ⟨S⟩ ⟨A⟩

⟨C-c⟩ ⟨S⟩ ⟨C⟩
: Change the Ciao executable used to run the top level. It is set by default to `ciao` or, to the environment variable `CIAO` if it is defined.

⟨C-c⟩ ⟨S⟩ ⟨C-c⟩
: Change the arguments passed to the Ciao executable. They are set by default to none or, to the environment variable `CIAOARGS` if it is defined.

⟨C-c⟩ ⟨S⟩ ⟨P⟩
: Change the executable used to run the CiaoPP toplevel. It is set by default to `ciaopp` or, to the environment variable `CIAOPP` if it is defined.

⟨C-c⟩ ⟨S⟩ ⟨C-p⟩
: Change the arguments passed to the CiaoPP executable. They are set by default to none or to the environment variable `CIAOPPARGS` if it is defined.

⟨C-c⟩ ⟨S⟩ ⟨D⟩
: Change the executable used to run the LPdoc auto-documenter. It is set by default to `lpdoc` or to the environment variable `LPDOC` if it is defined.

⟨C-c⟩ ⟨S⟩ ⟨C-d⟩
: Change the arguments passed to the LPdoc auto-documenter. They are set by default to none or to the environment variable `LPDOCARGS` if it is defined.

19.16 Other commands

Some other commands which are active in the Ciao mode:

⟨C-c⟩ ⟨C-l⟩
: Recenter the most recently used Ciao inferior process buffer (e.g., top level, preprocessor, etc.).

19.17 Traditional Prolog Mode Commands

These commands provide some bindings and facilities for loading programs, which are present in emacs Prolog modes of traditional Prolog systems (e.g., SICStus). This is useful mainly if the Ciao emacs mode is used with such Prolog systems. Note that these commands (`compile/1` and `consult/1`) are deprecated in Ciao (due to the more advanced, separate compilation model in Ciao) and their use in the Ciao top-level is not recommended.

⟨C-c⟩ ⟨K⟩
: Compile the entire buffer.

⟨C-c⟩ ⟨k⟩
: Compile a given region.

⟨C-c⟩ ⟨C-k⟩ Compile the predicate around point.

⟨C-c⟩ ⟨C⟩ Consult the entire buffer.

⟨C-c⟩ ⟨c⟩ Consult a given region.

⟨C-c⟩ ⟨C-c⟩ Consult the predicate around point.

19.18 Coexistence with other Prolog-like interfaces

As mentioned previously, the Ciao `emacs` interface can also be used to work with traditional Prolog or CLP systems. Also, the Ciao `emacs` interface (*mode*) can coexist with other Prolog-related `emacs` interfaces (*modes*) (such as, e.g., the `SICStus` Prolog interface). Only one of the interfaces can be active at a time for a given buffer (i.e., for each given file opened inside `emacs`). In order the change a buffer to a given interface, move the cursor to that buffer and type `M-x ...-mode` (e.g., for the Ciao mode, `M-x ciao-mode`).

If several Prolog-related `emacs` interfaces are loaded, then typically the *last* one to be loaded takes precedence, in the sense that this will be the interface in which `emacs` will be set when opening files which have a `.pl` ending (this depends a bit on how things are set up in your `.emacs` file).

19.19 Getting the Ciao mode version

⟨C-c⟩ ⟨v⟩ Report the version of the emacs Ciao mode.

19.20 Using Ciao mode capabilities in standard shells

The capabilities (commands, coloring, error location, ...) which are active in the Ciao *inferior* mode can also be made available in any standard command line shell which is being run within emacs. This can be enabled by going to the buffer in which the shell is running and typing "⟨M-x⟩ `ciao-inferior-mode`". This is very useful for example when running the standalone compiler, the `lpdoc` auto-documenter, or even certain user applications (those that use the standard error message library) in an emacs sub-shell. Turning the Ciao inferior mode on on that sub-shell will highlight and color the error messages, and automatically find and visit the locations in the files in which the errors are reported.

Finally, one the most useful applications of this is when using the embedded debugger (a version of the debugger which can be embedded into executables so that an interactive debugging session can be triggered at any time while running that executable without needing the top-level shell). If an application is run in a shell buffer which has been set with Ciao inferior mode (⟨M-x⟩ `ciao-inferior-mode`) and this application starts emitting output from the embedded debugger (i.e., which contains the embedded debugger and is debugging its code) then the Ciao emacs mode will be able to follow these messages, for example tracking execution in the source level code. This also works if the application is written in a combination of languages, provided the parts written in Ciao are compiled with the embedded debugger package and is thus a covenient way of debugging multi-language applications. The only thing needed is to make sure that the output messages appear in a shell buffer that is in Ciao inferior mode.

19.21 Customization

This section explains all variables used in the Ciao emacs mode which can be customized by users. Such customization can be performed (in later versions of `emacs`) from the `emacs` menus (`Help -> Customize -> Top-level Customization Group`), or also by adding a `setq` expression in the `.emacs` file. Such `setq` expression should be similar to:

```
(setq <variable> <new_value>)
```

The following sections list the different variables which can be customized for `ciao`, `ciaopp` and `lpdoc`.

19.21.1 Ciao general variables

`ciao-ask-for-version-maintenance-type` (*string*)
: If turned to yes the system asks prompts to set version control when saving files that do not set a version control system explicitly within the file.

`ciao-create-sample-file-on-startup` (*boolean*)
: When starting the Ciao environment using ciao-startup two buffers are opened: one with a toplevel and another with a sample file. This toggle controls whether the sample file, meant for novice users, is created or not. Set by default, non-novice users will probably want to turn it off.

`ciao-indent-width` (*integer*)
: Indentation level.

`ciao-inhibit-toolbar` (*boolean*)
: *Non-nil means dont use the specialized Ciao toolbar.

`ciao-locate-also-note-messages` (*boolean*)
: If set, also when errors of type NOTE are detected the corresponding file is visited and the location marked. It is set to nil by default because sometimes the user prefers not to take any action with respect to these messages (for example, many come from the documenter, indicating that adding certain declarations the documentation would be improved).

`ciao-locate-errors-after-run` (*boolean*)
: If set, location of any errors produced when running Ciao tools (loading or preprocessing code, running the documenter, etc.) will be initiated automatically. I.e., after running a command, the system will automatically highlight any error messages and the corresponding areas in source files if possible. If set to nil this location will only happen after typing M-] or accessing the corresponding menu or tool bar button.

`ciao-os-shell-prompt-pattern` (*string*)
: Regular expression used to describe typical shell prompt patterns (csh and bash), so that error location works in inferior shells. This is useful for example so that errors are located when generating documentation, and also when using the embedded debugger or any other application in a shell. It is best to be as precise as possible when defining this so that the standard Ciao error location does not get confused.

`ciao-path-dirs` (*string*)
: Colon-separated paths to collections of Ciao bundles (reads/sets the CIAOPATH environment variable). Left empty to use the default value.

`ciao-root-dir` (*string*)
: Path to the Ciao system path (reads/sets the CIAOROOT environment variable). Typically left empty, since Ciao executables know its installation path.

`ciao-scratchpad-root` (*directory*)
: Name of root directory of the scratchpad for temporary source files and directories.

`ciao-system` (*string*)
: Name of Ciao executable which runs the classical top level.

`ciao-system-args` (*string*)
: Arguments passed to Ciao toplevel executable.

`ciao-toplevel-buffer-name` (*string*)
: Basic name of the buffer running the Ciao toplevel inferior process.

`ciao-user-directives` (*list*)
: List of identifiers of any directives defined by users which you would like highlighted (colored). Be careful, since wrong entries may affect other syntax highlighting.

19.21.2 CiaoPP variables

`ciao-ciaopp-buffer-name` (*string*)
: Basic name of the buffer running the CiaoPP inferior process.

`ciao-ciaopp-gmenu-buffer-name` (*string*)
: Name of the buffer running the CiaoPP graphical menu interface.

`ciao-ciaopp-system` (*string*)
: Name of CiaoPP executable.

`ciao-ciaopp-system-args` (*string*)
: Arguments passed to CiaoPP executable.

`ciao-ciaopp-use-graphical-menu` (*boolean*)
: If set, an interactive graphical menu is used for controlling CiaoPP, instead of asking ascii questions in the CiaoPP buffer.

19.21.3 LPdoc variables

`ciao-lpdoc-buffer-name` (*string*)
: Basic name of the buffer running the auto-documenter inferior process.

`ciao-lpdoc-docformat` (*symbol*)
: Name of default output format used by LPdoc.

`ciao-lpdoc-system` (*string*)
: Name of LPdoc auto-documenter executable.

`ciao-lpdoc-system-args` (*string*)
: Arguments passed to LPdoc executable.

19.21.4 Faces used in syntax-based highlighting (coloring)

`ciao-button-pressed-widget-face` (*face*)
: Face used for documentation text.

`ciao-button-widget-face` (*face*)
: Face used for documentation text.

`ciao-edit-widget-face` (*face*)
: Face used for documentation text.

`ciao-face-aborted-mess` (*face*)
: Face to use for aborted test messages.

`ciao-face-answer-val` (*face*)
: Face to use for answer values in top level.

`ciao-face-answer-var` (*face*)
: Face to use for answer variables in top level.

`ciao-face-builtin-directive` (*face*)
: Face to use for the standard directives.

`ciao-face-check-assrt` (*face*)
: Face to use for check assertions.

`ciao-face-checked-assrt` (*face*)
: Face to use for checked assertions.

`ciao-face-ciaopp-option` (*face*)
: Face to use for CiaoPP option menus.

`ciao-face-clauseheadname` (*face*)
: Face to use for clause head functors.

`ciao-face-comment` (*face*)
: Face to use for code comments using fixed pitch (double %).

`ciao-face-concurrency-op` (*face*)
: Face to use for concurrency operators.

`ciao-face-condcode-directive` (*face*)
: Face to use for the conditional code directives.

`ciao-face-cut` (*face*)
: Face to use for cuts.

`ciao-face-debug-breakpoint` (*face*)
: Face to use with breakpoints in source debugger.

`ciao-face-debug-call` (*face*)
: Face to use when at call port in source debugger.

`ciao-face-debug-exit` (*face*)
: Face to use when at exit port in source debugger.

`ciao-face-debug-expansion` (*face*)
: Face to use in source debugger when source literal not located.

`ciao-face-debug-fail` (*face*)
: Face to use when at fail port in source debugger.

`ciao-face-debug-mess` (*face*)
: Face to use for debug messages.

`ciao-face-debug-redo` (*face*)
: Face to use when at redo port in source debugger.

`ciao-face-entry-assrt` (*face*)
: Face to use for entry assertions.

`ciao-face-error-mess` (*face*)
: Face to use for error messages.

`ciao-face-failed-mess` (*face*)
: Face to use for failed test messages.

`ciao-face-false-assrt` (*face*)
: Face to use for false assertions.

`ciao-face-fontify-sectioning` (*symbol*)
: Whether to fontify sectioning macros with varying height or a color face.

  If it is a number, use varying height faces. The number is used for scaling starting from ciao-face-sectioning-5-face. Typically values from 1.05 to 1.3 give best results, depending on your font setup. If it is the symbol color, use font-lock-type-face.

Caveats: Customizing the scaling factor applies to all sectioning faces unless those faces have been saved by customize. Setting this variable directly does not take effect unless you call ciao-face-update-sectioning-faces or restart Emacs.

Switching from color to a number or vice versa does not take effect unless you call M-x font-lock-fontify-buffer or restart Emacs.

`ciao-face-funexp-atom` (*face*)
: Face to use for atoms in functional notation.

`ciao-face-highlight-code` (*face*)
: Face to use for highlighting code areas (e.g., when locating the code area that an error message refers to).

`ciao-face-library-directive` (*face*)
: Face to use for directives defined in the library.

`ciao-face-lpdoc-bug-comment` (*face*)
: Face to use for LPdoc bug comments.

`ciao-face-lpdoc-command` (*face*)
: Face to use LPdoc commands inserted in documentation text.

`ciao-face-lpdoc-comment` (*face*)
: Face to use for LPdoc textual comments.

`ciao-face-lpdoc-verbatim` (*face*)
: Face to use for LPdoc verbatim text.

`ciao-face-modedef-assrt` (*face*)
: Face to use for modedef definitions.

`ciao-face-module-directive` (*face*)
: Face to use for the module-related directives.

`ciao-face-no-answer` (*face*)
: Face to use for no answer in top level.

`ciao-face-note-mess` (*face*)
: Face to use for note messages.

`ciao-face-other-mess` (*face*)
: Face to use for other messages.

`ciao-face-passed-mess` (*face*)
: Face to use for passed teset messages.

`ciao-face-predicate-directive` (*face*)
: Face to use for the predicate-related directives.

`ciao-face-prompt` (*face*)
: Face to use for prompts in top-level and shells.

`ciao-face-prop-assrt` (*face*)
: Face to use for property definitions.

`ciao-face-quoted-atom` (*face*)
: Face to use for quoted atoms.

`ciao-face-script-header` (*face*)
: Face to use for script headers.

`ciao-face-sectioning-0-face` (*face*)
: Face for sectioning commands at level 0.

Probably you dont want to customize this face directly. Better change the base face ciao-face-sectioning-5-face or customize the variable ciao-face-fontify-sectioning.

`ciao-face-sectioning-1-face` (*face*)
: Face for sectioning commands at level 1.

  Probably you dont want to customize this face directly. Better change the base face ciao-face-sectioning-5-face or customize the variable ciao-face-fontify-sectioning.

`ciao-face-sectioning-2-face` (*face*)
: Face for sectioning commands at level 2.

  Probably you dont want to customize this face directly. Better change the base face ciao-face-sectioning-5-face or customize the variable ciao-face-fontify-sectioning.

`ciao-face-sectioning-3-face` (*face*)
: Face for sectioning commands at level 3.

  Probably you dont want to customize this face directly. Better change the base face ciao-face-sectioning-5-face or customize the variable ciao-face-fontify-sectioning.

`ciao-face-sectioning-4-face` (*face*)
: Face for sectioning commands at level 4.

  Probably you dont want to customize this face directly. Better change the base face ciao-face-sectioning-5-face or customize the variable ciao-face-fontify-sectioning.

`ciao-face-sectioning-5-face` (*face*)
: Face for sectioning commands at level 5.

`ciao-face-startup-message` (*face*)
: Face to use for system splash message.

`ciao-face-string` (*face*)
: Face to use for strings.

`ciao-face-test-assrt` (*face*)
: Face to use for test assertions.

`ciao-face-texec-assrt` (*face*)
: Face to use for texec assertions.

`ciao-face-true-assrt` (*face*)
: Face to use for true assertions.

`ciao-face-trust-assrt` (*face*)
: Face to use for trust assertions.

`ciao-face-type-assrt` (*face*)
: Face to use for type definitions.

`ciao-face-user-directive` (*face*)
: Face to use for directives defined by the user (see ciao-user-directives custom variable to add new ones).

`ciao-face-variable` (*face*)
: Face to use for variables.

`ciao-face-warning-mess` (*face*)
: Face to use for warning messages.

`ciao-face-yes-answer` (*face*)
: Face to use for yes answer in top level.

`ciao-menu-error-widget-face` (*face*)
: Face used for menu error representation in graphical interface.

`ciao-menu-note-widget-face` (*face*)
: Face used for menu note representation in graphical interface.

`ciao-mouse-widget-face` (*face*)
    Face used for documentation text.

`ciao-text-widget-face` (*face*)
    Face used for documentation text.

`ciao-title-widget-face` (*face*)
    Face to use for interactive menu title.

19.22 Installation of the Ciao emacs interface

If opening a file ending with `.pl` puts emacs in another mode (such as `perl` mode, which is the –arguably incorrect– default setting in some `emacs` distributions), then either the emacs mode was not installed or the installation settings are being overwritten by other settings in your `.emacs` file or in some library. In any case, you can set things manually so that the Ciao mode is loaded by default in your system. This can be done by including in your `.emacs` file a line such as:

```
(load <INSTALL_CIAOROOT>/ciao-mode-init)
```

This loads the above mentioned file from the Ciao library. If you would like to configure things in a different way, you can also copy the contents of this file to your `.emacs` file and make the appropriate changes. For example, if you do not want `.pl` files to be put automatically in Ciao mode, then comment out (or remove) the line:

```
(setq auto-mode-alist ... )
```

You will then need to switch manually to Ciao mode by typing `M-x ciao-mode` after opening a Ciao file.

If you are able to open the Ciao menu but the Ciao manuals are not found or the `ciao` command (the top-level) is not found when loading `.pl` files, the probable cause is that you do not have the Ciao paths in the `INFOPATH` and `MANPATH` *environment variables* (whether these variables are set automatically or not for users depends on how the Ciao system was installed). See the Ciao installation instructions (Chapter 4 [Installing Ciao from the source distribution], page 39 or ⟨undefined⟩ [Installing Ciao from a Win32 binary distribution], page ⟨undefined⟩) for details.

19.23 Emacs version compatibility

This mode is currently being developed using `GNU emacs` version 26.3. It should also (hopefully) work with 25.XX and older versions.

19.24 Acknowledgments (ciao.el)

This code is derived from the 1993 version of the emacs interface for &-Prolog by Manuel Hermenegildo, itself derived from the original `prolog.el` by *Masanobu Umeda* with changes by *Johan Andersson*, *Peter Olin*, *Mats Carlsson*, and *Johan Bevemyr* of *SICS*, Sweden. Other changes also by Daniel Cabeza, Manuel C. Rodriguez, David Trallero, and Jose Morales. See the changelogs for details.

20 Advanced Ciao Emacs functionalities

Author(s): Miguel Angel Sanchez Ordaz.

This is the documentation for GNU `emacs` advanced functionalities. It explains how to install, set up, and use some of the most popular `emacs` packages with Ciao support. The use of these packages is optional, but may be quite helpful when using Ciao. The current packages are:

flycheck-ciao:
: advanced on-the-fly syntax, test, and assertion checking.

company-ciao:
: advanced auto-completion and auto-help framework.

To install any of these packages, use the `emacs` built-in package manager (*package.el*). Be sure the package list is refreshed (⟨ESC⟩ ⟨^X⟩ package-refresh-contents ◊) and then type:

- ⟨ESC⟩ ⟨^X⟩ package-install-file ◊ **PACKAGE PATH** ◊

21 Flycheck Ciao: on-the-fly syntax and assertion checking, and testing

Author(s): Miguel Angel Sanchez Ordaz.

Flycheck (`https://www.flycheck.org/en/latest/`) is an `emacs` extension for checking syntax on the fly. This package adds Flycheck (`https://www.flycheck.org/en/latest/`) support for Ciao *mode* with various checkers for different purposes:

- `ciaoc` as syntax checker.
- `ciaopp` as syntax and assertions checker.

21.1 Setup

Before installing this package, `emacs` 24 is needed for using Flycheck (`https://www.flycheck.org/en/latest/`). If you do not have installed it yet, this package `flycheck-ciao` will do it automatically from MELPA repository, but as it is not available in `emacs` by default, you should add this code to your `emacs` *init* file.

```
(require 'package)

(add-to-list 'package-archives '("MELPA Stable" . "https://stable.melpa.org/packages/") t)

(package-initialize)
```

There are two options for the setup:

- Insert the next line into your `emacs` init file.

  ```
  (eval-after-load 'flycheck '(add-hook 'flycheck-mode-hook 'flycheck-ciao-setup))
  ```

- If you use use-package (`https://github.com/jwiegley/use-package`), you can insert instead.

  ```
  (use-package flycheck-ciao
    :after flycheck
    :hook
    (flycheck-mode . flycheck-ciao-setup)
    )
  ```

21.2 Using Flycheck

All options when using Flycheck (`https://www.flycheck.org/en/latest/`) are available on the top bar menu *Tools -> Syntax Checking*.

- To enable Flycheck in the current buffer ⟨ESC⟩ ⟨^X⟩ flycheck-mode ◊.
  - If you are interested in having Flycheck enabled by default in all buffers where syntax checking is possible, insert the next line in your `emacs` init file.

    ```
    (add-hook 'after-init-hook 'global-flycheck-mode)
    ```

- To list all errors found by the checker: ⟨C-c⟩ ⟨!⟩ ⟨l⟩ or ⟨ESC⟩ ⟨^X⟩ flycheck-list-errors ◊.
- To select a different syntax checker: ⟨C-c⟩ ⟨!⟩ ⟨s⟩ or ⟨ESC⟩ ⟨^X⟩ flycheck-select-checker ◊.
- To verify setup: ⟨C-c⟩ ⟨!⟩ ⟨v⟩ or ⟨ESC⟩ ⟨^X⟩ flycheck-verify-setup ◊.

21.3 Recommended extensions

In the official Flycheck (`https://www.flycheck.org/en/latest/`) manual some extensions are recommended for better user experience.

- flycheck-colors-modeline (`https://github.com/flycheck/flycheck-color-mode-line`) (official) colors the mode line according to the Flycheck status.
- flycheck-pos-tip (`https://github.com/flycheck/flycheck-pos-tip`) shows Flycheck error messages in a graphical popup.

22 Company Ciao: advanced autocompletion

Author(s): Miguel Angel Sanchez Ordaz.

Company (`https://company-mode.github.io/`) is a text completion framework for Emacs. The name stands for "complete anything".

22.1 Setup

Before installing this package, `emacs` 24 is needed for using Company (`https://company-mode.github.io/`). If you do not have installed it yet, this package `company-ciao` will do it automatically from ELPA repository.

There are two options for the setup:

- Insert the next line into your `emacs` init file.

  ```
  (eval-after-load 'company '(add-hook 'company-mode-hook 'company-ciao-setup))
  ```

- If you use use-package (`https://github.com/jwiegley/use-package`), you can insert instead.

  ```
  (use-package flycheck-ciao
    :after flycheck
    :hook
    (company-mode . company-ciao-setup)
    )
  ```

22.2 Using Company

Once enabled the use of `company-mode` will be automatic.

To enable `company-mode` in all buffers where possible, insert into your `emacs` file the next line:

```
(add-hook 'after-init-hook 'global-company-mode)
```

23 PART II - The Ciao basic language

Author(s): The Ciao Development Team.

This part documents the *Ciao basic language*. It covers the basic language concepts, constructs, and some predefined predicates and declarations that, unless stated otherwise, are available by default every Ciao program as defined in the `prelude` package. However, it is also possible to avoid loading this `prelude` package, on a per module basis, by using some special packages (see `noprelude`, `pure`).

Most of these basic predicates are contained in the `engine` library modules.

The rest of the standard library predicates, including both the packages that provide the ISO-Prolog builtins, the other traditional Prolog builtins, and the many language extensions of Ciao beyond Prolog are documented in subsequent parts.

24 Basic concepts and conventions

Author(s): The Ciao Development Team.

24.1 Syntax terminology and notational conventions

Ciao is a language that has its roots in (and includes) the Prolog language. Thus, this manual assumes some familiarity with logic programming and Prolog. The reader is referred to standard textbooks on these topics (such as, e.g., [SS86, CM81, Apt97, Hog84]) for background. However, we recall herein some concepts for the sake of establishing terminology. Also, we will briefly introduce some of the extensions that Ciao brings to the Prolog language that are instrumental for understanding this part of the manual.

24.1.1 Predicates and their components

In logic programming *procedures* are also called *predicates* and *procedure call sites* are also called *literals*. Predicates are identified in this manual by the keyword 'PREDICATE' in the place where they are documented.

Program instructions are expressions made up of control constructs (Chapter 29 [Control constructs/predicates], page 143) and procedure calls (literals). These expressions are also called *goals*. Literals are also (atomic) goals.

A *predicate definition* is a sequence of clauses. A *clause* has the form "`H :- B.`" (ending in '`.`'), where `H` is syntactically the same as a literal and is called the clause *head*, and `B` is a comma separated list of literals (a conjunction) goal and is called the clause *body*. A clause with no body is written "`H.`" and is called a *fact*. Clauses with body are also called *rules*. A *program* is a sequence of predicate definitions.

24.1.2 Characters and character strings vs. atoms

We adopt the following convention for representing character strings in the text of this manual: when a string is being used as an atom it is written thus: `user` or `'user'`; in all other circumstances double quotes are used (as in `"hello"`).

24.1.3 Predicate specs

Predicates are distinguished by their name *and* their *arity* (number of arguments). We will call `name/arity` a *predicate spec*. Note that there can be predicates that have the same name and different arity. These are *different* predicates (conversely, they may of course have the same arity and different name). The notation `name/arity` is therefore used when it is necessary to refer to a predicate unambiguously. For example, `concatenate/3` specifies the predicate which is named "concatenate" and which takes 3 arguments.

24.1.4 Modes

When documenting a predicate, we will often describe how it is used with a *mode spec* which has the form `predicate(Arg1, ..., ArgN)` where each `Arg` may be preceded by a *mode*. A mode is a functor which is wrapped around an argument (or prepended if defined as an operator). Such a mode allows documenting in a compact way the *instantiation state* on call and exit of the argument to which it is applied. The set of modes which can be used in Ciao is not fixed. Instead, arbitrary modes can be defined in programs by using the `modedef/1` declarations of the Ciao *assertion language* (see Chapter 37 [The Ciao assertion language], page 189 for details). Modes are identified in this manual by the keyword 'MODE'.

Herein, we will use the set of modes defined in the Ciao `isomodes` library, which is essentially a precise definition of those used in the ISO-Prolog standard (Chapter 50 [ISO-Prolog modes], page 267).

24.1.5 Properties and types

Although Ciao is *not* a 'typed language,' it allows writing (and using) types, as well as more general *properties*. These may be properties of states and properties of computations. Properties of state allow expressing characteristics of the program variables at particular points in the computation, like in `sorted(X)` (`X` is a sorted list). Properties of the computation allow expressing characteristics of a whole computation for a predicate, as in `is_det(p(X,Y))` (such calls yield only one solution). Properties are just a special form of predicates (Chapter 39 [Declaring regular types], page 205) and are identified in this manual by the keyword 'PROPERTY'.

In Ciao types are a particular case of property. A frequently used library of types is *regular types* (Chapter 39 [Declaring regular types], page 205). Properties of this kind are identified in this manual by the keyword 'REGTYPE'.

24.1.6 Declarations

A *declaration* provides information to one of the Ciao environment tools. Declarations are interspersed in the code of a program. The target tool can be the compiler (telling it that a predicate is dynamic, or a meta-predicate, etc.), the preprocessor (which understands declarations of properties and types, assertions, etc.), the autodocumenter (which understands the previous declarations and also certain "comment" declarations), or the debugger, among other tools.

A declaration has the form `:- D.` where `D` is syntactically the same as a literal. Declarations are identified in this manual by the keyword 'DECLARATION'.

In Ciao users can define (and document) new declarations. New declarations are typically useful when defining extensions to the language (which in Ciao are called packages). Such extensions are often implemented as expansions (see ⟨undefined⟩ [Extending the syntax], page ⟨undefined⟩). There are many such extensions in Ciao. The `functions` library, which provides fuctional syntax, is an example. The fact that in Ciao expansions are local to modules (as operators, see below) makes it possible to use a certain language extension in a module without affecting other modules.

24.1.7 Operators

An operator is a functor (or predicate name) which has been declared as such, thus allowing its use in a prefix, infix, or suffix fashion, instead of the standard procedure-like fashion. E.g., declaring `+` as an infix operator allows writing `X+Y` instead of `'+'(X,Y)` (which may still, of course, be written).

Operators in Ciao are local to the module/file where they are declared. At the same time, some operators are standard and are declared by default in every program (see Chapter 151 [Defining operators], page 665). This manual documents the operator declarations in each (library) module where they are included. As with expansions, the fact that in Ciao operators are local to modules makes it possible to use a certain language extension in one module without affecting other modules.

25 The module system

Author(s): Daniel Cabeza, The Ciao Development Team.

Modularity is a basic notion in a modern computer language. Modules allow dividing programs into several parts, which have their own independent name spaces. Each module is written in its own file (see `module/2` and `module/3`) and consists of a sequence of directives and predicate definitions.

Modules provide functionality to other modules by *exporting* some of the predicates defined inside the module (and also through multifile predicates). However, a module does not modify the *syntax* that can be used in another module that loads it. This is done instead through the mechanism of packages (see Chapter 27 [Packages and language extension], page 137).

See [CH00a] for a detailed description of the Ciao module system.

25.0.0.1 Visibility rules

The module system in Ciao is, as in most Prolog implementations, *procedure based*. This means that predicate names are local to a module, but functor/atom names in data are shared (at least by default).

The *predicates visible in a module* are the predicates defined in that module, plus the predicates imported from other modules. Only predicates exported by a module can be imported from other modules. The default module of a given predicate name is the local one if the predicate is defined locally, else the *last* module from which the predicate is imported, where explicit imports have priority over implicit ones (that is, a predicate imported through a `use_module/2` declaration is always preferred over a predicate imported through a `use_module/1` declaration). To refer to a predicate from a module which is not the default module for that predicate the name has to be module qualified. A module-qualified predicate name has the form `Module:Predicate` as in the call `debugger:debug_module(M)`. Note that in Ciao this module qualification cannot be used for gaining access to predicates that have not been imported, nor for defining clauses of other modules.

25.0.0.2 Files with no module declaration ('user' files)

All predicates defined in files with no module declaration belong to a special module called `user`, from which they are all implicitly exported. This provides backward compatibility for programs written for Prolog implementations with no module system and allows dividing programs into several files without being aware of the module system at all. Note that this feature is only supported for the above-mentioned backward-compatibility reasons, and the use of `user` files is discouraged. Many attractive compilation features of Ciao cannot be supported for `user` modules.

25.0.0.3 Multifile predicates

The case of multifile predicates (defined with the declaration `multifile/1`) is also special. Multifile predicates can be defined by clauses distributed in several modules, and all modules which define a predicate as multifile can use that predicate. The name space of multifile predicates is independent, as if they belonged to the special module `multifile`.

25.0.0.4 Basic directives

Unlike in other Prolog systems, directives in Ciao are not goals to be *executed* by the compiler or top level. Instead, they are *read* and acted upon by these programs. The advantage of this is that the effect of the directives is consistent for executables, code loaded in the top level, code analyzed by the preprocessor, etc.

As a result, by default only the builtin directives or declarations defined in this section are available in user programs. However, it is possible to define new declarations using the `new_declaration/1` and `new_declaration/2` directives (or using packages including them). Also, packages may define new directives via code translations.

25.0.0.5 Libraries imported by default ('builtins')

While in Ciao there are no 'built-in' predicates (i.e., predicates whose load cannot be avoided or that that cannot be redefined –see below) for convenience every module or `user file` imports implicitly a number of modules called builtin modules (also referred to as default modules). Which exact modules are imported by default is controlled by the third argument of `module/3` declarations, the lack thereof in `module/2` declarations, some rules for user files, etc., as described below. For example, for backward compatibility with traditional Prolog systems, if a `module/2` declaration is used, then the traditional predicates that are built in in most Prolog systems are imported in that module (see Chapter 109 [Classic Prolog], page 493).

Predicates coming from builtin/default modules are imported before all other importations of the module. This allows the redefinition of builtins, i.e., the redefinition of any of the predicates imported by default from builtin/default modules (with the exception of `true/0`) by either defining local versions of these predicates or by importing them from other modules.

Moreover, the implicit importation of the basic modules can be fully disabled by some special packages (for example, omitting all default imports with `noprelude`, or defining pure Prolog modules with the `pure` package).

25.1 Usage and interface

- **Library usage:**
  Modules are an intrinsic feature of Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Regular Types:*
    `modulename/1`.

25.2 Documentation on exports

modulename/1: REGTYPE

A module name is an atom, not containing characters ':' or '$'. Also, `user` and `multifile` are reserved, as well as the module names of all builtin modules (because in an executable all modules must have distinct names).

Usage: `modulename(M)`

`M` is a module name (an atom).

25.3 Documentation on internals

module/3: DECLARATION

Usage: :- `module(Name,Exports,Packages)`.

Declares a module of name `Name` which exports the predicates in `Exports`, and uses the packages in `Packages`. `Name` must match with the name of the file where the module resides, without extension. For each source in `Packages`, a package file is used. If the source is specified with a path alias, this is the file included, if it is an atom, the library paths are searched. See `package/1` for a brief description of package files.

This directive must appear the first in the file.

Also, if the compiler finds an unknown declaration as the first term in a file, the name of the declaration is regarded as a package library to be included, and the arguments of the declaration (if present) are interpreted like the arguments of `module/3`.

– *The following properties should hold at call time:*

`Name` is a module name (an atom). (`modules:modulename/1`)

`Exports` is a list of `prednames`. (`basic_props:list/2`)

`Packages` is a list of `sourcenames`. (`basic_props:list/2`)

module/2: DECLARATION

Usage: :- `module(Name,Exports)`.

Same as directive `module/3`, with an implicit package `default`. This default package provides all the standard features provided by most Prolog systems so that Prolog programs with traditional `module/2` declarations can run without any change. See Chapter 109 [Classic Prolog], page 493.

– *The following properties should hold at call time:*

`Name` is a module name (an atom). (`modules:modulename/1`)

`Exports` is a list of `prednames`. (`basic_props:list/2`)

use_package/1: DECLARATION

:- `use_package(Package)`.

Specifies the use in this file of the packages defined in `Package`. See the description of the third argument of `module/3` for an explanation of package files.

This directive must appear the first in the file, or just after a `module/3` declaration. A file with no module declaration, in the absence of this directive, uses an implicit package `default` (see ⟨undefined⟩ [Other predicates and features defined by default], page ⟨undefined⟩).

Usage 1: :- `use_package(Package)`.

– *The following properties should hold at call time:*

`Package` is a source name. (`stream_basic:sourcename/1`)

Usage 2: :- `use_package(Package)`.

– *The following properties should hold at call time:*

`Package` is a list of `sourcenames`. (`basic_props:list/2`)

use_module/2: DECLARATION

Usage: :- `use_module(Module,Imports)`.

Specifies that this code imports from the module defined in `Module` the predicates in `Imports`. The imported predicates must be exported by the other module.

– *The following properties should hold at call time:*

`Module` is a source name. (`stream_basic:sourcename/1`)

`Imports` is a list of `prednames`. (`basic_props:list/2`)

use_module/1: DECLARATION

Usage: :- `use_module(Module)`.

Specifies that this code imports from the module defined in `Module` all the predicates exported by it. The previous version with the explicit import list is preferred to this as it minimizes the chances to have to recompile this code if the other module changes.

– *The following properties should hold at call time:*

`Module` is a source name. (`stream_basic:sourcename/1`)

import/2: DECLARATION

Usage: :- `import(Module,Imports)`.

Declares that this code imports from the module with name `Module` the predicates in `Imports`.

Important note: this declaration is intended to be used when the current module or the imported module is going to be dynamically loaded, and so the compiler does not include the code of the imported module in the current executable (if only because the compiler cannot know the location of the module file at the time of compilation). For the same reason the predicates imported are not checked to be exported by `Module`. Its use in other cases is strongly discouraged, as it disallows many compiler optimizations.

This is an example of such a case for a dynamically loaded module:

```
:- module(_,_).

:- import(bar,[b/1]).

main(X) :-
     use_module(bar),
     b(X).
```

– *The following properties should hold at call time:*

`Module` is a module name (an atom). (`modules:modulename/1`)

`Imports` is a list of `prednames`. (`basic_props:list/2`)

reexport/2: DECLARATION

Usage: :- `reexport(Module,Preds)`.

Specifies that this code reexports from the module defined in `Module` the predicates in `Preds`. This implies that this module imports from the module defined in `Module` the predicates in `Preds`, an also that this module exports the predicates in `Preds` .

– *The following properties should hold at call time:*

`Module` is a source name. (`stream_basic:sourcename/1`)

`Preds` is a list of `prednames`. (`basic_props:list/2`)

reexport/1: DECLARATION

Usage: :- `reexport(Module)`.

Specifies that this code reexports from the module defined in `Module` all the predicates exported by it. This implies that this module imports from the module defined in `Module` all the predicates exported by it, an also that this module exports all such predicates .

– *The following properties should hold at call time:*

`Module` is a source name. (`stream_basic:sourcename/1`)

ensure_loaded/1: DECLARATION

Usage: :- `ensure_loaded(File)`. ⟨• ISO •⟩

Specifies that the code present in `File` will be included in the executable being prepared, in the `user` module. The file `File` cannot have a module declaration. This directive is intended to be used by programs not divided in modules. Dividing programs into modules is however strongly encouraged, since most of the attractive features of Ciao (such as static debugging and global optimization) are only partially available for `user` modules.

– *The following properties should hold at call time:*

  `File` is a source name. (`stream_basic:sourcename/1`)

include/1: DECLARATION

Usage: :- `include(File)`. ⟨• ISO •⟩

The contents of the file `File` are included in the current program text exactly as if they had been written in place of this directive.

– *The following properties should hold at call time:*

  `File` is a source name. (`stream_basic:sourcename/1`)

export/1: DECLARATION

Usage 1: :- `export(Pred)`.

Adds `Pred` to the set of exported predicates.

– *The following properties should hold at call time:*

  `Pred` is a predicate name. (`basic_props:predname/1`)

Usage 2: :- `export(Exports)`.

Adds `Exports` to the set of exported predicates.

– *The following properties should hold at call time:*

  `Exports` is a list of `prednames`. (`basic_props:list/2`)

multifile/1: DECLARATION

Usage: :- `multifile Predicates`. ⟨• ISO •⟩

Specifies that each predicate in `Predicates` may have clauses in more than one file. Each file that contains clauses for a multifile predicate must contain a directive multifile for the predicate. The directive should precede all clauses of the affected predicates, and also dynamic/data declarations for the predicate. This directive is defined as a prefix operator in the compiler.

– *The following properties should hold at call time:*

  `Predicates` is a sequence or list of `prednames`. (`basic_props:sequence_or_list/2`)

meta_predicate/1: DECLARATION

Usage: :- `meta_predicate MetaSpecs`.

Specifies that the predicates in `MetaSpecs` have arguments which have to be module expanded (predicates, goals, etc). `meta_predicate/1` directives are only mandatory for exported predicates (in modules). This directive is defined as a prefix operator in the compiler.

- *The following properties should hold at call time:*
  `MetaSpecs` is a sequence of `metaspec`s. (`basic_props:sequence/2`)

redefining/1: DECLARATION

Usage: :- `redefining(Predicate)`.

Specifies that this module redefines predicate `Predicate`, also imported from other module, or imports it from more than one module. This prevents the compiler giving warnings about redefinitions of that predicate. `Predicate` can be partially (or totally) uninstantiated, to allow disabling those warnings for several (or all) predicates at once.

- *The following properties should hold at call time:*
  `predname` is *compatible* with `Predicate`. (`basic_props:compat/2`)

discontiguous/1: DECLARATION

Usage: :- `discontiguous Predicates`. ⟨• ISO •⟩

Specifies that each predicate in `Predicates` may be defined in this file by clauses which are not in consecutive order. Otherwise, a warning is signaled by the compiler when clauses of a predicate are not consecutive (this behavior is controllable by the prolog flag *discontiguous_warnings*). The directive should precede all clauses of the affected predicates. This directive is defined as a prefix operator in the compiler.

- *The following properties should hold at call time:*
  `Predicates` is a sequence or list of `predname`s. (`basic_props:sequence_or_list/2`)

impl_defined/1: DECLARATION

Usage: :- `impl_defined(Predicates)`.

Specifies that each predicate in `Predicates` is *impl*icitly *defined* in the current prolog source, either because it is a builtin predicate or because it is defined in a C file. Otherwise, a warning is signaled by the compiler when an exported predicate is not defined in the module or imported from other module.

- *The following properties should hold at call time:*
  `Predicates` is a sequence or list of `predname`s. (`basic_props:sequence_or_list/2`)

metaspec/1: REGTYPE

A meta-predicate specification for a predicate is the functor of that predicate applied to terms which represent the kind of module expansion that should be applied to each argument. Possible contents are represented as:

`?,+,-,_` These values denote that this argument is not module expanded.

`goal` This argument will be a term denoting a goal (either a simple or complex one) which will be called. For commpatibility reasons it can be named as `:` as well.

`clause` This argument will be a term denoting a clause.

`fact` This argument should be instantiated to a term denoting a fact (head-only clause).

`spec` This argument should be instantiated to a predicate name, as Functor/Arity.

`pred(`*N*`)` This argument should be instantiated to a predicate construct to be called by means of a `call/`*N* predicate call (see `call/2`).

`list(`*Meta*`)`
This argument should be instantiated to a list of terms as described by *Meta* (e.g. `list(goal)`).

`addterm(Meta)`
This argument should be instantiated to the meta-data specified by *Meta*, and an argument added after this one will carry the original data without module expansion. Not intended to be used by normal users.

`addmodule(Meta)`
This argument should be instantiated to the meta-data specified by *Meta*, and in an argument added after this one will be passed the calling module, for example to allow handling more involved meta-data by using conversion builtins. `addmodule` is an alias of `addmodule(?)`. Not intended to be used by normal users.

Usage: `metaspec(M)`

`M` is a meta-predicate specification.

initialization/1: DECLARATION

Usage: :- `initialization(Goal)`. ⟨ • ISO • ⟩

`Goal` will be executed at the start of the execution of any program containing the current code. The initialization of a module/file never runs before the initializations of the modules from which the module/file imports (excluding circular dependencies).

– *The following properties should hold at call time:*

  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

on_abort/1: DECLARATION

Usage: :- `on_abort(Goal)`.

`Goal` will be executed after an abort of the execution of any program containing the current code.

– *The following properties should hold at call time:*

  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

25.4 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

26 Bundles and workspaces

Author(s): Jose F. Morales, The Ciao Development Team.

In the same way that modules can be seen as collections of predicates, Ciao bundles define collections of modules. The visibility of bundles is determined by workspaces. A workspace is a directory that may contain bundles, each of them in a separate directory with the same name declared on its manifest. See Chapter 12 [Bundle management], page 77 for details about creating and managing bundles.

26.1 Bundles

Bundles are usually organized as follows:

```
Manifest/        Metadata
cmds/            Modules implementing main/{0,1} (for CLI tools)
{src,lib,...}/   Source code (modules, packages, included files, etc.)
```

Bundles include some *metadata* describing versions, dependencies, (relative) path aliases, entry points for manuals and commands, additional pre-build, build, and installation rules, etc. This metadata is separated from the actual code into a single `Manifest.pl` file at the root of the bundle, or inside the `Manifest/` directory. A typical `Manifest.pl` file has the following form:

```
:- bundle(<Name>).
version(<Version>).
depends([Dep1, ..., DepN]).
alias_paths([
    <Alias1> = <RelPath1>,
    ...
    <AliasN> = <RelPathN>
]).
lib(<RelPath>).
cmd(<CmdName>, [main=<RelPathMainMod>, ...]).
manual(<ManName>, [main=<RelPathDocCfg>, ...]).
```

where:

- `version/1` specifies the bundle version.
- `depends/1` declares dependencies with other bundles.
- `alias_paths/1` declares path aliases.
- (multiple) `lib/1` declares directories containing compilable modules.
- (multiple) `cmd/2` declares entry points for commands (executables).
- (multiple) `manual/2` declares entry points for manuals.

Versioning: Versions are specified as atoms (see `version_strings`) of the form `'X.Y.Z'`, where `X`, `Y`, and `Z` are non-negative integers, and an optional `-Prerelease` suffix. Omitted version numbers are assumed to be 0. Dependencies may include version constraints, e.g., `core-[version>='1.15']` (see `version_compare/3` for the definition of the comparison).

Currently the system supports only one version of a bundle at the same time. If two versions need to coexist at the same time, the bundles must be renamed (e.g. `bndv1`, `bndv2`).

Additional build rules: Although the Ciao compiler is incremental and performs the compilation on demand, some components may require configuration steps and pre-building. For example, this is often the case for bindings to foreign third-party libraries. Currently these rules must be written in a *lower-level* module Manifest/<bundle>.hooks.pl.

26.2 Workspaces

A workspace directory looks like:

```
build/          (Generated automatically)
<bundle1>/
...
<bundleN>/
```

Each workspace directory contains a `build/` directory with intermediate and final results of compilation (`build/bin/`) and documentation generation (`build/doc/`).

Bundle catalogs: workspaces may contain directories holding bundles. Such directories must be marked with an empty `BUNDLE_CATALOG` file. In that case, bundles are visible only if they are marked with an empty `ACTIVE` file. This mecanism is useful to implement catalogs of bundles.

27 Packages and language extension

Author(s): Daniel Cabeza, The Ciao Development Team.

This chapter documents package files and the main builtin directives that Ciao provides for implementing syntactic and semantic language extensions.

These directives allow the definition of new operators, new declarations, code translations, etc. Although these directives can be used directly in any source file, it is highly recommended to group them into packages, where each package contains a set of operators and transformations that together provide a particular syntactic and semantic extension to the modules that load it. Most Ciao extensions, such as functional syntax, constraint solving, or breadth-first search are implemented as packages. The structure and code of these Ciao library packages can serve as useful examples of how to build packages in general.

See also [CH00a] for a detailed description of the Ciao module system and, in particular, packages and the language extension primitives.

27.1 Usage and interface

- **Library usage:**
  These directives are builtin in Ciao, so nothing special has to be done to use them.

27.2 Documentation on internals

package/1: DECLARATION

Usage: :- `package(Name)`.

Declares the current file to be a package of name `Name`. Like in modules, `Name` must match the name of the file where the package resides, without extension. This directive must appear first in the file.

- *The following properties should hold at call time:*
  `Name` is a module name (an atom). (`modules:modulename/1`)

op/3: DECLARATION

Usage: :- `op(Priority,Op_spec,Operator)`. ⟨• ISO •⟩

Updates the operator table for reading the terms in the rest of the current text, in the same way as the builtin `op/3` does. Its scope is local to the current text. Usually included in package files.

- *The following properties should hold at call time:*
  `Priority` is an integer. (`basic_props:int/1`)
  `Op_spec` specifies the type and associativity of an operator. (`basic_props:operator_specifier/1`)
  `Operator` is an atom or a list of atoms. (`basic_props:atm_or_atm_list/1`)

new_declaration/1: DECLARATION

Usage: :- `new_declaration(Predicate)`.

Declares `Predicate` to be a valid declaration in the rest of the current text. Such declarations are simply ignored by the compiler or top level, but can be used by other code processing programs such as an automatic documentator. Also, they can easily translated into standard code (a set of facts and/or rules) by defining a suitable expansion (e.g., by `add_sentence_trans/1`, etc.). This is tipically done in package files.

Equivalent to `new_declaration(Predicate, off)`.

– *The following properties should hold at call time:*

`Predicate` is a predicate name. (`basic_props:predname/1`)

new_declaration/2: DECLARATION

Usage: :- `new_declaration(Predicate,In_Itf)`.

Declares `Predicate` to be a valid declaration in the rest of the current text. Such declarations will be included in the interface file for this file if `In_Itf` is 'on', not if it is 'off'. Including such declarations in interface files makes them visible while processing other modules which make use of this one.

– *The following properties should hold at call time:*

`Predicate` is a predicate name. (`basic_props:predname/1`)

`In_Itf` is 'on' or 'off' (`packages:switch/1`)

load_compilation_module/1: DECLARATION

Usage: :- `load_compilation_module(File)`.

Loads code defined in `File` into the compiler, usually including predicates which define translations of clauses, sentences, and terms, for use with the declarations `add_sentence_trans/2` and similar ones. The application order of translations is determined by ascending *priority* numbers. Normally included in package files.

– *The following properties should hold at call time:*

`File` is a source name. (`stream_basic:sourcename/1`)

add_sentence_trans/2: DECLARATION

Usage: :- `add_sentence_trans(Predicate,Priority)`.

Starts a translation, defined by `Predicate`, of the terms read by the compiler in the rest of the current text. For each subsequent term read by the compiler, the translation predicate is called to obtain a new term which will be used by the compiler as if it where the term present in the file. If the call fails, the term is used as such. A list may be returned also, to translate a single term into several terms. Before calling the translation predicate with actual program terms, it is called with an input of `0` to give an opportunity of making initializations for the module, discarding the result (note that normally a 0 could not be there). `Predicate` must be exported by a module previously loaded with a `load_compilation_module/1` declaration. Normally included in package files.

– *The following properties should hold at call time:*

`Predicate` is a translation predicate spec (has arity 2 or 3). (`packages:translation_predname/1`)

`Priority` is an integer. (`basic_props:int/1`)

add_term_trans/2: DECLARATION

Usage: :- `add_term_trans(P,Priority)`.

Starts a translation, defined by `Predicate`, of the terms and sub-terms read by the compiler in the rest of the current text. This translation is performed after all translations defined by `add_sentence_trans/1` are done. For each subsequent term read by the compiler, and recursively any subterm included, the translation predicate is called to possibly obtain a new term to replace the old one. Care must be taken of not introducing an endless loop of translations. `Predicate` must be exported by a module previously loaded with a `load_compilation_module/1` declaration. Normally included in package files.

– *The following properties should hold at call time:*

  `P` is a translation predicate spec (has arity 2 or 3). (`packages:translation_predname/1`)

  `Priority` is an integer. (`basic_props:int/1`)

add_goal_trans/2: DECLARATION

Usage: :- `add_goal_trans(Predicate,Priority)`.

Declares a translation, defined by `Predicate`, of the goals present in the clauses of the current text. This translation is performed after all translations defined by `add_sentence_trans/1` and `add_term_trans/1` are done. For each clause read by the compiler, the translation predicate is called with each goal present in the clause to possibly obtain other goal to substitute the original one, and the translation is subsequently applied to the resulting goal. Care must be taken of not introducing an endless loop of translations. `Predicate` must be exported by a module previously loaded with a `load_compilation_module/1` declaration. Bear in mind that this type of translation noticeably slows down compilation. Normally included in package files.

– *The following properties should hold at call time:*

  `Predicate` is a translation predicate spec (has arity 2 or 3). (`packages:translation_predname/1`)

  `Priority` is an integer. (`basic_props:int/1`)

add_clause_trans/2: DECLARATION

Usage: :- `add_clause_trans(Predicate,Priority)`.

Declares a translation, defined by `Predicate`, of the clauses of the current text. The translation is performed before `add_goal_trans/1` translations but after `add_sentence_trans/1` and `add_term_trans/1` translations. The usefulness of this translation is that information of the interface of related modules is available when it is performed. For each clause read by the compiler, the translation predicate is called with the first argument instantiated to a structure `clause(Head,Body)`, and the predicate must return in the second argument a similar structure, without changing the functor in `Head` (or fail, in which case the clause is used as is). Before executing the translation predicate with actual clauses it is called with an input of `clause(0,0)`, discarding the result.

– *The following properties should hold at call time:*

  `Predicate` is a translation predicate spec (has arity 2 or 3). (`packages:translation_predname/1`)

  `Priority` is an integer. (`basic_props:int/1`)

translation_predname/1: REGTYPE

A translation predicate is a predicate of arity 2 or 3 used to make compile-time translations. The compiler invokes a translation predicate instantiating its first argument with the item to be translated, and if the predicate is of arity 3 its third argument with the name of the module where the translation is done. If the call is successful, the second argument is used as if that item were in the place of the original, else the original item is used.

Usage: `translation_predname(P)`

`P` is a translation predicate spec (has arity 2 or 3).

27.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

28 Conditional compilation

Author(s): Jose F. Morales.

This package defines a series of directives for conditional compilation that allow the inclusion or exclusion of code blocks (which may contain nested conditional directives) based on the truth value at compile time of special goals called *conditions*. The syntax for conditional directives is:

```
:- if(Cond1).
  <<Block1>>
:- elif(Cond2).
  <<Block2>>
:- else.
  <<BlockN>>
:- endif.
```

where `elif(_)` can appear zero or more times and the `else` part is optional. The sentences in `Block1` are included if the condition in `Cond1` is satisfied, else `Block2` is included if `Cond2` is satisfied (and so on for each `elif`), and `BlockN` if no previous condition is satisfied.

28.1 Conditions

The valid conditions are restricted to a subset of goals that can be safely evaluated at compile time. At this moment, only the following ones are accepted:

- Compile-time values of prolog flags (`current_prolog_flag/2`).
- Conjunctions, disjunctions, or negations of conditions.
- Calls to facts previously defined with `:- compilation_fact(Fact)`. This is an experimental feature that may change in the future.
- `defined(F/N)` (or `defined(F)`, equivalent to `defined(F/0)`), which succeeds only if there is a definition for the compilation fact `F/N`.

28.2 Usage and interface

- **Library usage:**
  This module is imported by default (unless the `noprelude` or `pure` packages are used).
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `assertions`, `assertions/assertions_basic`.

29 Control constructs/predicates

Author(s): Daniel Cabeza, Manuel Hermenegildo.

This module contains the set of basic control predicates, except the predicates dealing with exceptions, which are in ⟨undefined⟩ [Exception and Signal handling], page ⟨undefined⟩.

29.1 Usage and interface

- **Library usage:**
  This module is imported by default (unless the `noprelude` or `pure` packages are used).
- **Exports:**
  - *Predicates:*
    `,/2`, `;/2`, `->/2`, `!/0`, `\+/1`, `if/3`, `true/0`, `fail/0`, `repeat/0`, `false/0`, `otherwise/0`.

29.2 Documentation on exports

,/2: PREDICATE

`P,Q`

Conjunction (`P` *and* `Q`).

Usage: ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Q` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `goal,goal`.

;/2: PREDICATE

`P;Q`

Disjunction (`P` *or* `Q`). Note that in Ciao `|/2` is not equivalent to `;/2`.

Usage: ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Q` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `goal;goal`.

->/2: PREDICATE

`P->Q`

If `P` then `Q` else fail, using first solution of `P` only. Also, `(P -> Q ; R)`, if `P` then `Q` else `R`, using first solution of `P` only. No cuts are allowed in `P`.

Usage: ⟨• ISO •⟩

– *The following properties should hold at call time:*

`P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

`Q` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `goal->goal`.

!/0: PREDICATE

Usage: ⟨• ISO •⟩

Commit to any choices taken in the current predicate.

– *The following properties hold globally:*

All calls of the form `!` are deterministic. (`native_props:is_det/1`)

All the calls of the form `!` do not fail. (`native_props:not_fails/1`)

Goal `!` produces `1` solutions. (`native_props:relations/2`)

\+/1: PREDICATE

`\+P`

Goal `P` is not provable (negation by failure). Fails if `P` has a solution, and succeeds otherwise. No cuts are allowed in `P`.

Usage: ⟨• ISO •⟩

– *The following properties should hold at call time:*

`P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP as `not(X)`. (`basic_props:native/2`)

All calls of the form `\+P` are deterministic. (`native_props:is_det/1`)

Meta-predicate with arguments: `\+goal`.

if/3: PREDICATE

`if(P,Q,R)`

If `P` then `Q` else `R`, exploring all solutions of `P`. No cuts are allowed in `P`.

Usage: `if(A,B,C)`

– *The following properties should hold at call time:*

`A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`C` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

A is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

B is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

C is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

– *The following properties hold upon exit:*

A is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

B is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

C is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

Meta-predicate with arguments: `if(goal,goal,goal)`.

true/0: PREDICATE

Usage: ⟨• ISO •⟩

Succeed (noop).

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (basic_props:native/1)

`true` is side-effect `free`. (basic_props:sideff/2)

All calls of the form `true` are deterministic. (native_props:is_det/1)

All the calls of the form `true` do not fail. (native_props:not_fails/1)

Goal `true` produces `1` solutions. (native_props:relations/2)

Other properties:

– *The following properties hold globally:*

`true` is evaluable at compile-time. (basic_props:eval/1)

fail/0: PREDICATE

Usage: ⟨• ISO •⟩

Fail, backtrack immediately.

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (basic_props:native/1)

`fail` is side-effect `free`. (basic_props:sideff/2)

All calls of the form `fail` are deterministic. (native_props:is_det/1)

Calls of the form `fail` fail. (native_props:fails/1)

Goal `fail` produces `0` solutions. (native_props:relations/2)

Other properties:

– *The following properties hold globally:*

`fail` is evaluable at compile-time. (basic_props:eval/1)

– *The following properties hold globally:*

`fail` is equivalent to `fail`. (basic_props:equiv/2)

repeat/0: PREDICATE

Usage: ⟨ • ISO • ⟩

Generates an infinite sequence of backtracking choices.

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

`repeat` is side-effect `free`. (`basic_props:sideff/2`)

false/0: PREDICATE

– *The following properties hold globally:*

`false` is side-effect `free`. (`basic_props:sideff/2`)

`false` is evaluable at compile-time. (`basic_props:eval/1`)

otherwise/0: PREDICATE

– *The following properties hold globally:*

`otherwise` is side-effect `free`. (`basic_props:sideff/2`)

`otherwise` is evaluable at compile-time. (`basic_props:eval/1`)

29.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`native_props`, `debugger`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`, `nativeprops`.

30 Exception and signal handling

Author(s): The Ciao Development Team, Jose F. Morales (global vars version).

This module includes predicates related to exceptions and signals, which alter the normal flow of execution.

30.1 Usage and interface

- **Library usage:**
  This module is imported by default (unless the `noprelude` or `pure` packages are used).
- **Exports:**
  - *Predicates:*
    `catch/3`, `intercept/3`, `throw/1`, `send_signal/1`, `send_signal/2`, `halt/0`, `halt/1`, `abort/0`.

30.2 Documentation on exports

catch/3: PREDICATE

`catch(Goal,Error,Handler)`

Executes `Goal`. If an exception is raised during its execution, `Error` is unified with the exception, and if the unification succeeds, the entire execution derived from `Goal` is aborted, and `Handler` is executed. The execution resumes with the continuation of the `catch/3` call. For example, given the code

```
p(X) :- throw(error), display('---').
p(X) :- display(X).
```

the execution of "`catch(p(0), E, display(E)), display(.), fail.`" results in the output "`error.`".

Usage: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Error` is any term. (`basic_props:term/1`)
  `Handler` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties should hold at call time:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties hold upon exit:*
  `Error` is any term. (`basic_props:term/1`)
  `Handler` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate with arguments: `catch(goal,?,goal)`.

intercept/3: PREDICATE

`intercept(Goal,Signal,Handler)`

Executes `Goal`. If a signal is sent during its execution, `Signal` is unified with the exception, and if the unification succeeds, `Handler` is executed and then the execution resumes after the point where the exception was thrown. To avoid infinite loops if `Handler` raises an exception which unifies with `Error`, the exception handler is deactivated before executing `Handler`. Note the difference with builtin `catch/3`, given the code

```
p(X) :- send_signal(error), display('---').
p(X) :- display(X).
```

the execution of "`intercept(p(0), E, display(E)), display(.), fail.`" results in the output "`error---.0.`".

Usage:

- *Call and exit should be compatible with:*
  `Signal` is any term. (`basic_props:term/1`)
- *The following properties should hold at call time:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `Handler` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*
  `Signal` is any term. (`basic_props:term/1`)

Meta-predicate with arguments: `intercept(goal,?,goal)`.

throw/1: PREDICATE

`throw(Ball)`

Raises an error, throwing the exception `Ball`, to be caught by an ancestor `catch/3`. The closest matching ancestor is chosen. In addition to calls to `throw/2` in user code, exceptions are also thrown by many library predicates in cases of error.

Usage: `throw(Term)` ⟨• ISO •⟩

- *The following properties should hold at call time:*
  `Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
- *The following properties hold globally:*
  `throw(Term)` is equivalent to `fail`. (`basic_props:equiv/2`)

send_signal/1: PREDICATE

`send_signal(Signal)`

Emits a signal, to be intercepted by an ancestor `intercept/3`. The closest matching ancestor is chosen. If the signal is not intercepted, the following error is thrown: `error(unintercepted_signal(Signal), send_signal/1-1)`.

Usage: `send_signal(Term)`

- *The following properties should hold at call time:*
  `Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

send_signal/2: PREDICATE

`send_signal(Signal,Intercepted)`

Emits a signal as `send_signal/1`, `Intercepted=false` if the signal is not intercepted (i.e. just suceeds) or `true` otherwise.

Usage: `send_signal(Term,Intercepted)`

– *The following properties should hold at call time:*

`Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

halt/0: PREDICATE

`halt`

Halt the system, exiting to the invoking shell.

Usage: ⟨• ISO •⟩

– *The following properties should hold globally:*

This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

halt/1: PREDICATE

`halt(Code)`

Halt the system, exiting to the invoking shell, returning exit code `Code`.

Usage: ⟨• ISO •⟩

– *The following properties should hold at call time:*

`Code` is an integer. (`basic_props:int/1`)

– *The following properties should hold upon exit:*

`Code` is an integer. (`basic_props:int/1`)

– *The following properties should hold globally:*

`halt(Code)` is equivalent to `fail`. (`basic_props:equiv/2`)

abort/0: PREDICATE

`abort`

Abort the current execution.

30.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`datafacts_rt`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`, `datafacts`.

31 Extra-logical properties for typing

Author(s): Daniel Cabeza, Manuel Hermenegildo.

This library contains traditional Prolog predicates for testing types. They depend on the state of instantiation of their arguments, thus being of extra-logical nature.

31.1 Usage and interface

- **Library usage:**
  This module is imported by default (unless the `noprelude` or `pure` packages are used).
- **Exports:**
  - *Properties:*
    `var/1`, `nonvar/1`, `atom/1`, `integer/1`, `float/1`, `number/1`, `atomic/1`, `ground/1`, `type/2`.

31.2 Documentation on exports

var/1: PROPERTY

Usage: `var(X)`

`X` is a free variable.

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP as `free(X)`. (`basic_props:native/2`)

Other properties:

- *The following properties hold globally:*
  All calls of the form `var(Arg1)` are deterministic. (`native_props:is_det/1`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

`var(X)`

- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  `var(X)` is side-effect `free`. (`basic_props:sideff/2`)

`var(X)`

- *If the following properties hold at call time:*
  `X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `var(X)` is evaluable at compile-time. (`basic_props:eval/1`)

`var(X)`

- *If the following properties hold at call time:*
  `X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `var(X)` is equivalent to `fail`. (`basic_props:equiv/2`)

`var(X)`
- *If the following properties hold at call time:*
  `X` is a free variable. (`term_typing:var/1`)
  then the following properties hold globally:
  `var(X)` is equivalent to `true`. (`basic_props:equiv/2`)

nonvar/1: PROPERTY

Usage: `nonvar(X)`

`X` is currently a term which is not a free variable.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP as `not_free(X)`. (`basic_props:native/2`)

Other properties:
- *The following properties hold globally:*
  All calls of the form `nonvar(Arg1)` are deterministic. (`native_props:is_det/1`)

`nonvar(X)`
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)
  `nonvar(X)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

`nonvar(X)`
- *If the following properties hold at call time:*
  `X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `nonvar(X)` is evaluable at compile-time. (`basic_props:eval/1`)

`nonvar(T)`
- *If the following properties hold at call time:*
  `T` is a free variable. (`term_typing:var/1`)
  then the following properties hold globally:
  `nonvar(T)` is equivalent to `fail`. (`basic_props:equiv/2`)

`nonvar(T)`
- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `nonvar(T)` is equivalent to `true`. (`basic_props:equiv/2`)

atom/1: PROPERTY

Usage: `atom(X)`

`X` is currently instantiated to an atom.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`atom(X)`

- *The following properties hold upon exit:*
  `X` is an atom. (`basic_props:atm/1`)
- *The following properties hold globally:*
  All calls of the form `atom(Arg1)` are deterministic. (`native_props:is_det/1`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

`atom(X)`

- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)
  `atom(X)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

`atom(X)`

- *If the following properties hold at call time:*
  `X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `atom(X)` is evaluable at compile-time. (`basic_props:eval/1`)

`atom(T)`

- *If the following properties hold at call time:*
  `T` is a free variable. (`term_typing:var/1`)
  then the following properties hold globally:
  `atom(T)` is equivalent to `fail`. (`basic_props:equiv/2`)

integer/1: PROPERTY

Usage: `integer(X)`

`X` is currently instantiated to an integer.

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`integer(X)`

- *The following properties hold upon exit:*
  `X` is an integer. (`basic_props:int/1`)
- *The following properties hold globally:*
  All calls of the form `integer(Arg1)` are deterministic. (`native_props:is_det/1`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

`integer(X)`

- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)
  `integer(X)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

`integer(X)`
- *If the following properties hold at call time:*
  `X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `integer(X)` is evaluable at compile-time. (`basic_props:eval/1`)

`integer(T)`
- *If the following properties hold at call time:*
  `T` is a free variable. (`term_typing:var/1`)
  then the following properties hold globally:
  `integer(T)` is equivalent to `fail`. (`basic_props:equiv/2`)

float/1: PROPERTY

Usage: `float(X)`

`X` is currently instantiated to a float.
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`float(X)`
- *The following properties hold upon exit:*
  `X` is a float. (`basic_props:flt/1`)
- *The following properties hold globally:*
  All calls of the form `float(Arg1)` are deterministic. (`native_props:is_det/1`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

`float(X)`
- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)
  `float(X)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

`float(X)`
- *If the following properties hold at call time:*
  `X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `float(X)` is evaluable at compile-time. (`basic_props:eval/1`)

`float(T)`
- *If the following properties hold at call time:*
  `T` is a free variable. (`term_typing:var/1`)
  then the following properties hold globally:
  `float(T)` is equivalent to `fail`. (`basic_props:equiv/2`)

number/1: PROPERTY

Usage: `number(X)`

`X` is currently instantiated to a number.

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`number(X)`

- *The following properties hold upon exit:*
  `X` is a number. (`basic_props:num/1`)
- *The following properties hold globally:*
  All calls of the form `number(Arg1)` are deterministic. (`native_props:is_det/1`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

`number(X)`

- *The following properties hold globally:*
  `X` is not further instantiated. (`basic_props:not_further_inst/2`)
  `number(X)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

`number(X)`

- *If the following properties hold at call time:*
  `X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `number(X)` is evaluable at compile-time. (`basic_props:eval/1`)

`number(T)`

- *If the following properties hold at call time:*
  `T` is a free variable. (`term_typing:var/1`)
  then the following properties hold globally:
  `number(T)` is equivalent to `fail`. (`basic_props:equiv/2`)

atomic/1: PROPERTY

Usage: `atomic(X)`

`X` is currently instantiated to an atom or a number.

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`atomic(T)`

- *The following properties hold upon exit:*
  `T` is an atomic term (an atom or a number). (`basic_props:constant/1`)
- *The following properties hold globally:*
  All calls of the form `atomic(Arg1)` are deterministic. (`native_props:is_det/1`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

`atomic(X)`
– *The following properties hold globally:*
`X` is not further instantiated. (`basic_props:not_further_inst/2`)
`atomic(X)` is side-effect `free`. (`basic_props:sideff/2`)
This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

`atomic(X)`
– *If the following properties hold at call time:*
`X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
then the following properties hold globally:
`atomic(X)` is evaluable at compile-time. (`basic_props:eval/1`)

`atomic(T)`
– *If the following properties hold at call time:*
`T` is a free variable. (`term_typing:var/1`)
then the following properties hold globally:
`atomic(T)` is equivalent to `fail`. (`basic_props:equiv/2`)

ground/1: PROPERTY

Usage: `ground(X)`

`X` is currently ground (it contains no variables).
– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`ground(X)`
– *The following properties hold upon exit:*
`X` is ground. (`basic_props:gnd/1`)
– *The following properties hold globally:*
All calls of the form `ground(Arg1)` are deterministic. (`native_props:is_det/1`)
Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

`ground(X)`
– *The following properties hold globally:*
`X` is not further instantiated. (`basic_props:not_further_inst/2`)
`ground(X)` is side-effect `free`. (`basic_props:sideff/2`)
This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

`ground(X)`
– *If the following properties hold at call time:*
`X` is currently ground (it contains no variables). (`term_typing:ground/1`)
then the following properties hold globally:
`ground(X)` is evaluable at compile-time. (`basic_props:eval/1`)

`ground(X)`

- *If the following properties hold at call time:*
  X is a free variable. (term_typing:var/1)
  then the following properties hold globally:
  ground(X) is equivalent to fail. (basic_props:equiv/2)

ground(X)
- *If the following properties hold at call time:*
  X is currently ground (it contains no variables). (term_typing:ground/1)
  then the following properties hold globally:
  ground(X) is equivalent to true. (basic_props:equiv/2)

type/2: PROPERTY

Usage: type(X,Y)

X is internally of type Y (var, attv, float, integer, structure, atom or list).
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (basic_props:native/1)

Other properties:

type(X,Y)
- *The following properties hold upon exit:*
  Y is an atom. (basic_props:atm/1)
- *The following properties hold globally:*
  All calls of the form type(Arg1,Arg2) are deterministic. (native_props:is_det/1)
- *The following properties hold globally:*
  type(Arg1,Arg2) is side-effect free. (basic_props:sideff/2)
  This predicate is understood natively by CiaoPP. (basic_props:native/1)

type(X,Y)
- *If the following properties hold at call time:*
  X is currently a term which is not a free variable. (term_typing:nonvar/1)
  then the following properties hold globally:
  type(X,Y) is evaluable at compile-time. (basic_props:eval/1)

31.3 Documentation on imports

This module has the following direct dependencies:
- *System library modules:*
  native_props.
- *Packages:*
  prelude, initial, condcomp, assertions, assertions/assertions_basic, nortchecks, nativeprops, isomodes.

32 Basic term manipulation

Author(s): Daniel Cabeza, Manuel Hermenegildo.

This module provides basic term manipulation.

32.1 Usage and interface

- **Library usage:**
  This module is imported by default (unless the `noprelude` or `pure` packages are used).
- **Exports:**
  - *Predicates:*
    `\=/2`, `arg/3`, `functor/3`, `=../2`, `copy_term/2`, `copy_term_nat/2`, `C/3`, `cyclic_term/1`.
  - *Properties:*
    `=/2`, `const_head/1`.
  - *Regular Types:*
    `list_functor/1`, `non_empty_list/1`.

32.2 Documentation on exports

=/2: PREDICATE

`X=Y`

`X` and `Y` unify.

- *The following properties hold globally:*
  `X=Y` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  `X=Y` is evaluable at compile-time. (`basic_props:eval/1`)
  All calls of the form `X=Y` are deterministic. (`native_props:is_det/1`)
  Goal `X=Y` produces `inf` solutions. (`native_props:relations/2`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

\=/2: PREDICATE

Usage: `X\=Y` ⟨• ISO •⟩

`X` and `Y` are not unifiable.

- *The following properties hold globally:*
  `X\=Y` is side-effect `free`. (`basic_props:sideff/2`)
  `X\=Y` is binding insensitive. (`basic_props:bind_ins/1`)
  All calls of the form `X\=Y` are deterministic. (`native_props:is_det/1`)

Other properties:

`X\=Y`

- *If the following properties hold at call time:*
  X is currently ground (it contains no variables). (term_typing:ground/1)
  Y is currently ground (it contains no variables). (term_typing:ground/1)
  then the following properties hold globally:
  X\=Y is evaluable at compile-time. (basic_props:eval/1)

X\=Y

- *The following properties hold globally:*
  X\=Y is side-effect free. (basic_props:sideff/2)

arg/3: PREDICATE

Usage 1: arg(ArgNo,Term,Arg)

- *The following properties should hold at call time:*
  ArgNo is a number. (basic_props:num/1)
- *The following properties hold globally:*
  All calls of the form arg(ArgNo,Term,Arg) are deterministic. (native_props:is_det/1)
  Goal arg(ArgNo,Term,Arg) produces inf solutions. (native_props:relations/2)

Usage 2: arg(ArgNo,Term,Arg)

- *The following properties should hold at call time:*
  ArgNo is a number. (basic_props:num/1)
  Term is ground. (basic_props:gnd/1)
- *The following properties hold upon exit:*
  Arg is ground. (basic_props:gnd/1)

Usage 3: arg(ArgNo,Term,Arg) ⟨• ISO •⟩

Argument ArgNo of the term Term is Arg.

- *The following properties should hold at call time:*
  ArgNo is a non-negative integer. (basic_props:nnegint/1)
  Term is a compound term. (basic_props:struct/1)
- *The following properties hold globally:*
  arg(ArgNo,Term,Arg) is side-effect free. (basic_props:sideff/2)
  This predicate is understood natively by CiaoPP. (basic_props:native/1)
  arg(ArgNo,Term,Arg) is binding insensitive. (basic_props:bind_ins/1)

Usage 4: arg(ArgNo,Term,Arg)

- *The following properties should hold at call time:*
  ArgNo is a non-negative integer. (basic_props:nnegint/1)
  Term is ground. (basic_props:gnd/1)
- *The following properties hold upon exit:*
  Arg is ground. (basic_props:gnd/1)

functor/3: PREDICATE

Usage 1: `functor(Term,Name,Arity)` ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
- *The following properties hold upon exit:*

  `Name` is an atom. (`basic_props:atm/1`)

  `Arity` is a non-negative integer. (`basic_props:nnegint/1`)
- *The following properties hold globally:*

  `Term` is not further instantiated. (`basic_props:not_further_inst/2`)

  `functor(Term,Name,Arity)` is side-effect `free`. (`basic_props:sideff/2`)

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  `functor(Term,Name,Arity)` is binding insensitive. (`basic_props:bind_ins/1`)

  `functor(Term,Name,Arity)` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 2: `functor(Term,Name,Arity)` ⟨• ISO •⟩

The principal functor of the term `Term` has name `Name` and arity `Arity`.

- *The following properties should hold at call time:*

  `Name` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Arity` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
- *The following properties hold upon exit:*

  `Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Name` is an atom. (`basic_props:atm/1`)

  `Arity` is a non-negative integer. (`basic_props:nnegint/1`)
- *The following properties hold globally:*

  `functor(Term,Name,Arity)` is side-effect `free`. (`basic_props:sideff/2`)

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  `functor(Term,Name,Arity)` is binding insensitive. (`basic_props:bind_ins/1`)

  `functor(Term,Name,Arity)` is evaluable at compile-time. (`basic_props:eval/1`)

  All the calls of the form `functor(Term,Name,Arity)` do not fail. (`native_props:not_fails/1`)

Usage 3: `functor(Term,Name,Arity)`

- *The following properties should hold at call time:*

  `Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Name` is a free variable. (`term_typing:var/1`)

  `Arity` is a free variable. (`term_typing:var/1`)

  `Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
- *The following properties hold upon exit:*

  `1` is the size of argument `Name`, for any approximation. (`native_props:size/2`)

  `arity(Term)` is the size of argument `Arity`, for any approximation. (`native_props:size/2`)
- *The following properties hold globally:*

  `arity` is the measure used to determine the size of the terms that `Term` is bound to, for any type of approximation. (`native_props:size_metric/3`)

Other properties:

- *The following properties hold globally:*
  `functor(Arg1,Arg2,Arg3)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  All calls of the form `functor(Arg1,Arg2,Arg3)` are deterministic. (`native_props:is_det/1`)

=../2: PREDICATE

Usage: `Term=..List` ⟨• ISO •⟩

The functor and arguments of the term `Term` comprise the list `List`.

- *The following properties hold upon exit:*
  `List` is a list. (`basic_props:list/1`)
- *The following properties hold globally:*
  `Term=..List` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

- *If the following properties hold at call time:*
  `Arg1` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `Arg1=..Arg2` is evaluable at compile-time. (`basic_props:eval/1`)

`Arg1=..List`

- *If the following properties hold at call time:*
  `List` is a list. (`basic_props:list/1`)
  `term_basic:const_head(List)` (`term_basic:const_head/1`)
  then the following properties hold globally:
  `Arg1=..List` is evaluable at compile-time. (`basic_props:eval/1`)

const_head/1: PREDICATE

No further documentation available for this predicate.

copy_term/2: PREDICATE

Usage: `copy_term(Term,Copy)` ⟨• ISO •⟩

`Copy` is a renaming of `Term`, such that brand new variables have been substituted for all variables in `Term`. If any of the variables of `Term` have attributes, the copied variables will have copies of the attributes as well. It behaves as if defined by:

```
:- data 'copy of'/1.

copy_term(X, Y) :-
    asserta_fact('copy of'(X)),
    retract_fact('copy of'(Y)).
```

- *The following properties hold globally:*
  `copy_term(Term,Copy)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`copy_term(Term,Copy)`

- *If the following properties hold at call time:*
  `Term` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `copy_term(Term,Copy)` is evaluable at compile-time. (`basic_props:eval/1`)

copy_term_nat/2: PREDICATE

Usage: `copy_term_nat(Term,Copy)`

Same as `copy_term/2`, except that attributes of variables are not copied.

- *The following properties hold globally:*
  `copy_term_nat(Term,Copy)` is side-effect `free`. (`basic_props:sideff/2`)

C/3: PREDICATE

Usage 1: `C(S1,Terminal,S2)`

- *The following properties hold upon exit:*
  `term_basic:list_functor(S1)` (`term_basic:list_functor/1`)

Usage 2: `C(S1,Terminal,S2)`

`S1` is connected by the terminal `Terminal` to `S2`. Internally used in *DCG grammar rules*. Defined as if by the single clause: `'C'([X|S], X, S).`

- *The following properties hold globally:*
  `C(S1,Terminal,S2)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`C(S1,Terminal,S2)`

- *If the following properties hold at call time:*
  A list that is not the empty list []. (`term_basic:non_empty_list/1`)
  then the following properties hold upon exit:
  `S2` is a list. (`basic_props:list/1`)
- *If the following properties hold at call time:*
  `Arg1` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `C(Arg1,Arg2,Arg3)` is evaluable at compile-time. (`basic_props:eval/1`)

list_functor/1: REGTYPE

A regular type, defined as follows:

```
list_functor([A|B]) :-
    term(A),
    term(B).
```

non_empty_list/1: REGTYPE

Usage: `non_empty_list(A)`

A list that is not the empty list [].

cyclic_term/1: PREDICATE

Usage: `cyclic_term(T)`

True if `T` is cyclic (infinite).

32.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `native_props`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nativeprops`, `isomodes`, `nortchecks`.

33 Comparing terms

Author(s): Daniel Cabeza, Manuel Hermenegildo.

These built-in predicates are extra-logical. They treat uninstantiated variables as objects with values which may be compared, and they never instantiate those variables. They should *not* be used when what you really want is arithmetic comparison or unification.

The predicates make reference to a *standard total ordering* of terms, which is as follows:

- Variables, by age (roughly, oldest first – the order is *not* related to the names of variables).
- Floats, in numeric order (e.g. -1.0 is put before 1.0).
- Integers, in numeric order (e.g. -1 is put before 1).
- Atoms, in alphabetical (i.e. character code) order.
- Compound terms, ordered first by arity, then by the name of the principal functor, then by the arguments in left-to-right order. Recall that lists are equivalent to compound terms with principal functor `'.'/2`.

For example, here is a list of terms in standard order:

```
[ X, -1.0, -9, 1, bar, foo, [1], X = Y, foo(0,2), bar(1,1,1) ]
```

33.1 Usage and interface

- **Library usage:**
  This module is imported by default (unless the `noprelude` or `pure` packages are used).
- **Exports:**
  - *Predicates:*
    `\==/2`, `@</2`, `@=</2`, `@>/2`, `@>=/2`, `compare/3`.
  - *Properties:*
    `==/2`.
  - *Regular Types:*
    `comparator/1`.

33.2 Documentation on exports

==/2: PROPERTY

Usage: `Term1==Term2`

The terms `Term1` and `Term2` are strictly identical.

Other properties:

`Term1==Term2`

- *The following properties hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term1==Term2` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

`Term1==Term2`

- *If the following properties hold at call time:*
  `Term1` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `Term2` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `Term1==Term2` is evaluable at compile-time. (`basic_props:eval/1`)

`Term1==Term2`

- *The following properties hold globally:*
  All calls of the form `Term1==Term2` are deterministic. (`native_props:is_det/1`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

\==/2: PREDICATE

Usage: `Term1\==Term2`

The terms `Term1` and `Term2` are not strictly identical.

- *The following properties hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term1\==Term2` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`Term1\==Term2`

- *If the following properties hold at call time:*
  `Term1` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `Term2` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `Term1\==Term2` is evaluable at compile-time. (`basic_props:eval/1`)

`Term1\==Term2`

- *The following properties hold globally:*
  All calls of the form `Term1\==Term2` are deterministic. (`native_props:is_det/1`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

@</2: PREDICATE

Usage: `Term1@<Term2`

The term `Term1` precedes the term `Term2` in the standard order.

- *The following properties hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term1@<Term2` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`Term1@<Term2`

- *If the following properties hold at call time:*
  `Term1` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `Term2` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `Term1@<Term2` is evaluable at compile-time. (`basic_props:eval/1`)

`@=</2`: PREDICATE

Usage: `Term1@=<Term2`

The term `Term1` precedes or is identical to the term `Term2` in the standard order.

- *The following properties hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term1@=<Term2` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`Term1@=<Term2`

- *If the following properties hold at call time:*
  `Term1` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `Term2` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `Term1@=<Term2` is evaluable at compile-time. (`basic_props:eval/1`)

`@>/2`: PREDICATE

Usage: `Term1@>Term2`

The term `Term1` follows the term `Term2` in the standard order.

- *The following properties hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term1@>Term2` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`Term1@>Term2`

- *If the following properties hold at call time:*
  `Term1` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `Term2` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `Term1@>Term2` is evaluable at compile-time. (`basic_props:eval/1`)

@>=/2: PREDICATE

Usage: `Term1@>=Term2`

The term `Term1` follows or is identical to the term `Term2` in the standard order.

- *The following properties hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term1@>=Term2` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`Term1@>=Term2`

- *If the following properties hold at call time:*
  `Term1` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `Term2` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `Term1@>=Term2` is evaluable at compile-time. (`basic_props:eval/1`)

compare/3: PREDICATE

`compare(Op,Term1,Term2)`

`Op` is the result of comparing the terms `Term1` and `Term2`.

Usage:

- *Calls should, and exit will be compatible with:*
  `Op` is an atom. (`basic_props:atm/1`)
- *The following properties should hold at call time:*
  `Term1` is any term. (`basic_props:term/1`)
  `Term2` is any term. (`basic_props:term/1`)
- *The following properties hold upon exit:*
  `Op` is an atom. (`basic_props:atm/1`)
  `Term1` is any term. (`basic_props:term/1`)
  `Term2` is any term. (`basic_props:term/1`)
  `term_compare:comparator(Op)` (`term_compare:comparator/1`)
  `Term1` is any term. (`basic_props:term/1`)
  `Term2` is any term. (`basic_props:term/1`)
- *The following properties hold globally:*
  `Term1` is not further instantiated. (`basic_props:not_further_inst/2`)
  `Term2` is not further instantiated. (`basic_props:not_further_inst/2`)
  `compare(Op,Term1,Term2)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`compare(Op,Term1,Term2)`

- *If the following properties hold at call time:*
  `Term1` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `Term2` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `compare(Op,Term1,Term2)` is evaluable at compile-time. (`basic_props:eval/1`)

comparator/1: REGTYPE

A regular type, defined as follows:

```
comparator(=).
comparator(>).
comparator(<).
```

33.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `native_props`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `nativeprops`, `isomodes`.

34 Conversion between constants and strings

Author(s): The Ciao Development Team.

The Ciao system provides builtin predicates which allow conversions between constants (atomic terms, i.e., atoms or numbers) and their string representation (character code lists). Note that sometimes strings are more suitable to handle sequences of characters.

34.1 Usage and interface

- **Library usage:**
  This module is imported by default (unless the `noprelude` or `pure` packages are used).
- **Exports:**
  - *Predicates:*
    `name/2`, `atom_codes/2`, `number_codes/2`, `number_codes/3`, `atom_number/2`, `atom_number/3`, `atom_length/2`, `atom_concat/3`, `sub_atom/4`.

34.2 Documentation on exports

name/2: PREDICATE

`name(Const,String)`

`String` is the list of the ASCII codes of the characters comprising the name of `Const`. Note that if `Const` is an atom whose name can be interpreted as a number (e.g. '96'), the predicate is not reversible, as that atom will not be constructed when `Const` is uninstantiated. Thus it is recommended that new programs use the ISO-compliant predicates `atom_codes/2` or `number_codes/2`, as these predicates do not have this inconsistency.

Usage 1:

- *Calls should, and exit will be compatible with:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Const` is an atomic term (an atom or a number). (`basic_props:constant/1`)
- *The following properties hold upon exit:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties hold globally:*
  `name(Const,String)` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 2:

If `String` can be interpreted as a number, `Const` is unified with that number, otherwise with the atom whose name is `String`.

- *The following properties should hold at call time:*
  `Const` is a free variable. (`term_typing:var/1`)
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties hold upon exit:*
  `Const` is an atomic term (an atom or a number). (`basic_props:constant/1`)
- *The following properties hold globally:*
  `name(Const,String)` is evaluable at compile-time. (`basic_props:eval/1`)

Other properties:

- *The following properties hold globally:*
  `name(Const,String)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

atom_codes/2: PREDICATE

`atom_codes(Atom,String)` ⟨• ISO •⟩

`String` is the list of the ASCII codes of the characters comprising the name of `Atom`.

Usage 1:

- *Calls should, and exit will be compatible with:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Atom` is an atom. (`basic_props:atm/1`)
- *The following properties hold upon exit:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties hold globally:*
  `atom_codes(Atom,String)` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 2:

- *Calls should, and exit will be compatible with:*
  `Atom` is an atom. (`basic_props:atm/1`)
- *The following properties should hold at call time:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties hold upon exit:*
  `Atom` is an atom. (`basic_props:atm/1`)
- *The following properties hold globally:*
  `atom_codes(Atom,String)` is evaluable at compile-time. (`basic_props:eval/1`)

Other properties:

- *The following properties hold globally:*
  `atom_codes(Atom,String)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  All calls of the form `atom_codes(Atom,String)` are deterministic. (`native_props:is_det/1`)

number_codes/2: PREDICATE

`number_codes(Number,String)` ⟨• ISO •⟩

`String` is the list of the ASCII codes of the characters comprising a representation of `Number`.

Usage 1:

- *Calls should, and exit will be compatible with:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Number` is a number. (`basic_props:num/1`)

- *The following properties hold upon exit:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties hold globally:*
  `number_codes(Number,String)` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 2:

- *The following properties should hold at call time:*
  `Number` is a free variable. (`term_typing:var/1`)
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties hold upon exit:*
  `Number` is a number. (`basic_props:num/1`)
- *The following properties hold globally:*
  `number_codes(Number,String)` is evaluable at compile-time. (`basic_props:eval/1`)

Other properties:

- *The following properties hold globally:*
  `number_codes(Number,String)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

`number_codes(A,B)`

- *If the following properties hold at call time:*
  `A` is an integer. (`basic_props:int/1`)
  then the following properties hold upon exit:
  `B` is a string (a list of character codes). (`basic_props:string/1`)

number_codes/3: PREDICATE

`number_codes(Number,Base,String)`

`String` is the list of the ASCII codes of the characters comprising a representation of `Number` in base `Base`.

Usage 1:

- *Calls should, and exit will be compatible with:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Number` is a number. (`basic_props:num/1`)
  `Base` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties hold globally:*
  `number_codes(Number,Base,String)` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 2:

- *The following properties should hold at call time:*
  `Number` is a free variable. (`term_typing:var/1`)
  `Base` is an integer. (`basic_props:int/1`)
  `String` is a string (a list of character codes). (`basic_props:string/1`)

- *The following properties hold upon exit:*
  `Number` is a number. (`basic_props:num/1`)
- *The following properties hold globally:*
  `number_codes(Number,Base,String)` is evaluable at compile-time. (`basic_props:eval/1`)

Other properties:

- *The following properties hold globally:*
  `number_codes(Number,Base,String)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

atom_number/2: PREDICATE

`atom_number(Atom,Number)`

`Atom` can be read as a representation of `Number`.

Usage 1:

- *Call and exit should be compatible with:*
  `Number` is a number. (`basic_props:num/1`)
- *The following properties should hold at call time:*
  `Atom` is an atom. (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Number` is a number. (`basic_props:num/1`)
  `Atom` is an atom. (`basic_props:atm/1`)
  `Number` is a number. (`basic_props:num/1`)
- *The following properties should hold globally:*
  `atom_number(Atom,Number)` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 2:

- *The following properties should hold at call time:*
  `Atom` is a free variable. (`term_typing:var/1`)
  `Number` is a number. (`basic_props:num/1`)
- *The following properties should hold upon exit:*
  `Atom` is an atom. (`basic_props:atm/1`)
  `Atom` is an atom. (`basic_props:atm/1`)
  `Number` is a number. (`basic_props:num/1`)
- *The following properties should hold globally:*
  `atom_number(Atom,Number)` is evaluable at compile-time. (`basic_props:eval/1`)

Other properties:

- *The following properties hold globally:*
  `atom_number(Atom,Number)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  All calls of the form `atom_number(Atom,Number)` are deterministic. (`native_props:is_det/1`)

atom_number/3: PREDICATE

`atom_number(Atom,Base,Number)`

`Atom` can be read as a representation of `Number` in base `Base`.

Usage 1:

- *Call and exit should be compatible with:*

  `Base` is a number. (`basic_props:num/1`)
- *The following properties should hold at call time:*

  `Atom` is an atom. (`basic_props:atm/1`)

  `Number` is a number. (`basic_props:num/1`)
- *The following properties should hold upon exit:*

  `Base` is a number. (`basic_props:num/1`)

Usage 2:

- *The following properties should hold at call time:*

  `Atom` is a free variable. (`term_typing:var/1`)

  `Base` is a number. (`basic_props:num/1`)

  `Number` is a number. (`basic_props:num/1`)
- *The following properties should hold upon exit:*

  `Atom` is an atom. (`basic_props:atm/1`)

Other properties:

- *The following properties hold globally:*

  `atom_number(Atom,Base,Number)` is side-effect `free`. (`basic_props:sideff/2`)

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  All calls of the form `atom_number(Atom,Base,Number)` are deterministic. (`native_props:is_det/1`)

atom_length/2: PREDICATE

`atom_length(Atom,Length)` ⟨• ISO •⟩

`Length` is the number of characters forming the name of `Atom`.

Usage:

- *Calls should, and exit will be compatible with:*

  `Length` is an integer. (`basic_props:int/1`)
- *The following properties should hold at call time:*

  `Atom` is an atom. (`basic_props:atm/1`)
- *The following properties hold upon exit:*

  `Length` is an integer. (`basic_props:int/1`)
- *The following properties hold globally:*

  `atom_length(Atom,Length)` is evaluable at compile-time. (`basic_props:eval/1`)

Other properties:

- *The following properties hold globally:*

  `atom_length(Atom,Length)` is side-effect `free`. (`basic_props:sideff/2`)

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  All calls of the form `atom_length(Atom,Length)` are deterministic. (`native_props:is_det/1`)

atom_concat/3: PREDICATE

`atom_concat(Atom_1,Atom_2,Atom_12)` ⟨• ISO •⟩

`Atom_12` is the result of concatenating `Atom_1` followed by `Atom_2`.

Usage 1:

Concatenate two atoms.

- *Calls should, and exit will be compatible with:*
  `Atom_12` is an atom. (`basic_props:atm/1`)
- *The following properties should hold at call time:*
  `Atom_1` is an atom. (`basic_props:atm/1`)
  `Atom_2` is an atom. (`basic_props:atm/1`)
- *The following properties hold upon exit:*
  `Atom_12` is an atom. (`basic_props:atm/1`)
  `Atom_1` is an atom. (`basic_props:atm/1`)
  `Atom_2` is an atom. (`basic_props:atm/1`)
  `Atom_12` is an atom. (`basic_props:atm/1`)
- *The following properties hold globally:*
  `atom_concat(Atom_1,Atom_2,Atom_12)` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 2:

Non-deterministically split an atom.

- *The following properties should hold at call time:*
  `Atom_1` is a free variable. (`term_typing:var/1`)
  `Atom_2` is a free variable. (`term_typing:var/1`)
  `Atom_12` is an atom. (`basic_props:atm/1`)
- *The following properties hold upon exit:*
  `Atom_1` is an atom. (`basic_props:atm/1`)
  `Atom_2` is an atom. (`basic_props:atm/1`)
  `Atom_1` is an atom. (`basic_props:atm/1`)
  `Atom_2` is an atom. (`basic_props:atm/1`)
  `Atom_12` is an atom. (`basic_props:atm/1`)
- *The following properties hold globally:*
  `atom_concat(Atom_1,Atom_2,Atom_12)` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 3:

Take out of an atom a certain suffix (or fail if it cannot be done).

- *The following properties should hold at call time:*
  `Atom_1` is a free variable. (`term_typing:var/1`)
  `Atom_2` is an atom. (`basic_props:atm/1`)
  `Atom_12` is an atom. (`basic_props:atm/1`)
- *The following properties hold upon exit:*
  `Atom_1` is an atom. (`basic_props:atm/1`)
  `Atom_1` is an atom. (`basic_props:atm/1`)
  `Atom_2` is an atom. (`basic_props:atm/1`)
  `Atom_12` is an atom. (`basic_props:atm/1`)

– *The following properties hold globally:*
  `atom_concat(Atom_1,Atom_2,Atom_12)` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 4:

Take out of an atom a certain prefix (or fail if it cannot be done).

– *The following properties should hold at call time:*
  `Atom_1` is an atom. (`basic_props:atm/1`)
  `Atom_2` is a free variable. (`term_typing:var/1`)
  `Atom_12` is an atom. (`basic_props:atm/1`)
– *The following properties hold upon exit:*
  `Atom_2` is an atom. (`basic_props:atm/1`)
  `Atom_1` is an atom. (`basic_props:atm/1`)
  `Atom_2` is an atom. (`basic_props:atm/1`)
  `Atom_12` is an atom. (`basic_props:atm/1`)
– *The following properties hold globally:*
  `atom_concat(Atom_1,Atom_2,Atom_12)` is evaluable at compile-time. (`basic_props:eval/1`)

Other properties:

– *The following properties hold globally:*
  `atom_concat(Atom_1,Atom_2,Atom_12)` is side-effect `free`. (`basic_props:sideff/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  All calls of the form `atom_concat(Atom_1,Atom_2,Atom_12)` are deterministic. (`native_props:is_det/1`)

sub_atom/4: PREDICATE

`sub_atom(Atom,Before,Length,Sub_atom)`

`Sub_atom` is formed with `Length` consecutive characters of `Atom` after the `Before` character. For example, the goal `sub_atom(summer,1,4,umme)` succeeds.

Usage:

– *Calls should, and exit will be compatible with:*
  `Sub_atom` is an atom. (`basic_props:atm/1`)
– *The following properties should hold at call time:*
  `Atom` is an atom. (`basic_props:atm/1`)
  `Before` is an integer. (`basic_props:int/1`)
  `Length` is an integer. (`basic_props:int/1`)
– *The following properties hold upon exit:*
  `Sub_atom` is an atom. (`basic_props:atm/1`)
– *The following properties hold globally:*
  `sub_atom(Atom,Before,Length,Sub_atom)` is evaluable at compile-time. (`basic_props:eval/1`)

Other properties:

- *The following properties hold globally:*

  `sub_atom(Atom,Before,Length,Sub_atom)` is side-effect `free`. (`basic_props:sideff/2`)

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

34.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `native_props`.
- *Packages:*

  `noprelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`, `nativeprops`.

35 Arithmetic

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Arithmetic is performed by built-in predicates which take as arguments arithmetic expressions (see `arithexpression/1`) and evaluate them. Terms representing arithmetic expressions can be created dynamically, but at the time of evaluation, each variable in an arithmetic expression must be bound to a non-variable expression (the term must be ground). For example, given the code in the first line a possible shell interaction follows:

```
evaluate(Expression, Answer) :- Answer is Expression.

?- _X=24*9, evaluate(_X+6, Ans).

Ans = 222 ?

yes
```

35.1 Usage and interface

- **Library usage:**
  This module is imported by default (unless the `noprelude` or `pure` packages are used).
- **Exports:**
  - *Predicates:*
    `is/2`, `</2`, `=</2`, `>/2`, `>=/2`, `=:=/2`, `=\=/2`.
  - *Regular Types:*
    `arithexpression/1`, `intexpression/1`.
  - *Multifiles:*
    `$internal_error_where_term/4`.

35.2 Documentation on exports

is/2: PREDICATE

`Val is Exp` ⟨• ISO •⟩

The arithmetic expression `Exp` is evaluated and the result is unified with `Val`

Usage 1: `A is B`

- *The following properties should hold at call time:*
  `A` is a free variable. (`term_typing:var/1`)
  `B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `A` is a free variable. (`term_typing:var/1`)
  `B` is an arithmetic expression. (`arithmetic:arithexpression/1`)
- *The following properties hold upon exit:*
  `A` is a number. (`basic_props:num/1`)
  `B` is an arithmetic expression. (`arithmetic:arithexpression/1`)
  `int(B)` is the size of argument `A`, for any approximation. (`native_props:size/2`)

- *The following properties hold globally:*
  All the calls of the form `A is B` do not fail. (`native_props:not_fails/1`)
  `A is B` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 2: `A is B`

- *The following properties should hold at call time:*
  `A` is a free variable. (`term_typing:var/1`)
  `B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `A` is a free variable. (`term_typing:var/1`)
  `B` is an integer expression. (`arithmetic:intexpression/1`)
- *The following properties hold upon exit:*
  `A` is an integer. (`basic_props:int/1`)
  `B` is an integer expression. (`arithmetic:intexpression/1`)
  `int(B)` is the size of argument `A`, for any approximation. (`native_props:size/2`)
- *The following properties hold globally:*
  All the calls of the form `A is B` do not fail. (`native_props:not_fails/1`)
  `A is B` is evaluable at compile-time. (`basic_props:eval/1`)

Usage 3: `A is B`

- *The following properties should hold at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `A` is a number. (`basic_props:num/1`)
  `B` is an arithmetic expression. (`arithmetic:arithexpression/1`)
- *The following properties hold globally:*
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

Usage 4: `A is B`

- *The following properties should hold at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `A` is an integer. (`basic_props:int/1`)
  `B` is an integer expression. (`arithmetic:intexpression/1`)
- *The following properties hold globally:*
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

Other properties:

`A is B`

- *The following properties hold globally:*
  `int` is the measure used to determine the size of the terms that `A` is bound to, for any type of approximation. (`native_props:size_metric/3`)
  `int` is the measure used to determine the size of the terms that `B` is bound to, for any type of approximation. (`native_props:size_metric/3`)

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  `Val is Exp` is side-effect `free`. (`basic_props:sideff/2`)
  `Val is Exp` is binding insensitive. (`basic_props:bind_ins/1`)
  All calls of the form `Val is Exp` are deterministic. (`native_props:is_det/1`)
  Goal `Val is Exp` produces `inf` solutions. (`native_props:relations/2`)

</2: PREDICATE

`Exp1<Exp2` ⟨• ISO •⟩

The numeric value of `Exp1` is less than the numeric value of `Exp2` when both are evaluated as arithmetic expressions.

Usage: `A<B`

- *The following properties should hold at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `A` is an arithmetic expression. (`arithmetic:arithexpression/1`)
  `B` is an arithmetic expression. (`arithmetic:arithexpression/1`)
- *The following properties hold globally:*
  `A<B` is evaluable at compile-time. (`basic_props:eval/1`)
  `int` is the measure used to determine the size of the terms that `A` is bound to, for any type of approximation. (`native_props:size_metric/3`)
  `int` is the measure used to determine the size of the terms that `B` is bound to, for any type of approximation. (`native_props:size_metric/3`)

Other properties:

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  `Exp1<Exp2` is side-effect `free`. (`basic_props:sideff/2`)
  `Exp1<Exp2` is binding insensitive. (`basic_props:bind_ins/1`)
  All calls of the form `Exp1<Exp2` are deterministic. (`native_props:is_det/1`)
  Goal `Exp1<Exp2` produces `inf` solutions. (`native_props:relations/2`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

=</2: PREDICATE

`Exp1=<Exp2` ⟨• ISO •⟩

The numeric value of `Exp1` is less than or equal to the numeric value of `Exp2` when both are evaluated as arithmetic expressions.

Usage: `A=<B`

- *The following properties should hold at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `A` is an arithmetic expression. (`arithmetic:arithexpression/1`)
  `B` is an arithmetic expression. (`arithmetic:arithexpression/1`)

- *The following properties hold globally:*

  `A=<B` is evaluable at compile-time. (`basic_props:eval/1`)

  `int` is the measure used to determine the size of the terms that `A` is bound to, for any type of approximation. (`native_props:size_metric/3`)

  `int` is the measure used to determine the size of the terms that `B` is bound to, for any type of approximation. (`native_props:size_metric/3`)

Other properties:

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  `Exp1=<Exp2` is side-effect `free`. (`basic_props:sideff/2`)

  `Exp1=<Exp2` is binding insensitive. (`basic_props:bind_ins/1`)

  All calls of the form `Exp1=<Exp2` are deterministic. (`native_props:is_det/1`)

  Goal `Exp1=<Exp2` produces `inf` solutions. (`native_props:relations/2`)

  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

>/2: PREDICATE

`Exp1>Exp2` ⟨• ISO •⟩

The numeric value of `Exp1` is greater than the numeric value of `Exp2` when both are evaluated as arithmetic expressions.

Usage: `A>B`

- *The following properties should hold at call time:*

  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `A` is an arithmetic expression. (`arithmetic:arithexpression/1`)

  `B` is an arithmetic expression. (`arithmetic:arithexpression/1`)

- *The following properties hold globally:*

  `A>B` is evaluable at compile-time. (`basic_props:eval/1`)

  `int` is the measure used to determine the size of the terms that `A` is bound to, for any type of approximation. (`native_props:size_metric/3`)

  `int` is the measure used to determine the size of the terms that `B` is bound to, for any type of approximation. (`native_props:size_metric/3`)

Other properties:

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  `Exp1>Exp2` is side-effect `free`. (`basic_props:sideff/2`)

  `Exp1>Exp2` is binding insensitive. (`basic_props:bind_ins/1`)

  All calls of the form `Exp1>Exp2` are deterministic. (`native_props:is_det/1`)

  Goal `Exp1>Exp2` produces `inf` solutions. (`native_props:relations/2`)

  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

>=/2: PREDICATE

`Exp1>=Exp2` ⟨• ISO •⟩

The numeric value of `Exp1` is greater than or equal to the numeric value of `Exp2` when both are evaluated as arithmetic expressions.

Usage: `A>=B`

- *The following properties should hold at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `A` is an arithmetic expression. (`arithmetic:arithexpression/1`)
  `B` is an arithmetic expression. (`arithmetic:arithexpression/1`)
- *The following properties hold globally:*
  `A>=B` is evaluable at compile-time. (`basic_props:eval/1`)
  `int` is the measure used to determine the size of the terms that `A` is bound to, for any type of approximation. (`native_props:size_metric/3`)
  `int` is the measure used to determine the size of the terms that `B` is bound to, for any type of approximation. (`native_props:size_metric/3`)

Other properties:

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  `Exp1>=Exp2` is side-effect `free`. (`basic_props:sideff/2`)
  `Exp1>=Exp2` is binding insensitive. (`basic_props:bind_ins/1`)
  All calls of the form `Exp1>=Exp2` are deterministic. (`native_props:is_det/1`)
  Goal `Exp1>=Exp2` produces `inf` solutions. (`native_props:relations/2`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

=:=/2: PREDICATE

`Exp1=:=Exp2` ⟨• ISO •⟩

The numeric values of `Exp1` and `Exp2` are equal when both are evaluated as arithmetic expressions.

Usage: `A=:=B`

- *The following properties should hold at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `A` is an arithmetic expression. (`arithmetic:arithexpression/1`)
  `B` is an arithmetic expression. (`arithmetic:arithexpression/1`)
- *The following properties hold globally:*
  `A=:=B` is evaluable at compile-time. (`basic_props:eval/1`)
  `int` is the measure used to determine the size of the terms that `A` is bound to, for any type of approximation. (`native_props:size_metric/3`)
  `int` is the measure used to determine the size of the terms that `B` is bound to, for any type of approximation. (`native_props:size_metric/3`)

Other properties:

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  `Exp1=:=Exp2` is side-effect `free`. (`basic_props:sideff/2`)

  `Exp1=:=Exp2` is binding insensitive. (`basic_props:bind_ins/1`)

  All calls of the form `Exp1=:=Exp2` are deterministic. (`native_props:is_det/1`)

  Goal `Exp1=:=Exp2` produces `inf` solutions. (`native_props:relations/2`)

  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

=\=/2: PREDICATE

`Exp1=\=Exp2` ⟨• ISO •⟩

The numeric values of `Exp1` and `Exp2` are not equal when both are evaluated as arithmetic expressions.

Usage: `A=\=B`

- *The following properties should hold at call time:*

  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `B` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `A` is an arithmetic expression. (`arithmetic:arithexpression/1`)

  `B` is an arithmetic expression. (`arithmetic:arithexpression/1`)

- *The following properties hold globally:*

  `A=\=B` is evaluable at compile-time. (`basic_props:eval/1`)

  `int` is the measure used to determine the size of the terms that `A` is bound to, for any type of approximation. (`native_props:size_metric/3`)

  `int` is the measure used to determine the size of the terms that `B` is bound to, for any type of approximation. (`native_props:size_metric/3`)

Other properties:

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  `Exp1=\=Exp2` is side-effect `free`. (`basic_props:sideff/2`)

  `Exp1=\=Exp2` is binding insensitive. (`basic_props:bind_ins/1`)

  All calls of the form `Exp1=\=Exp2` are deterministic. (`native_props:is_det/1`)

  Goal `Exp1=\=Exp2` produces `inf` solutions. (`native_props:relations/2`)

  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

arithexpression/1: REGTYPE

An arithmetic expression is a term built from numbers and evaluable functors that represent arithmetic functions. An arithmetic expression evaluates to a number, which may be an integer (`int/1`) or a float (`flt/1`). The evaluable functors allowed in an arithmetic expression are listed below, together with an indication of the functions they represent. All evaluable functors defined in ISO-Prolog are implemented, as well as some other useful or traditional. Unless stated otherwise, an expression evaluates to a float if any of its arguments is a float, otherwise to an integer.

- `- /1`: sign reversal. ⟨• ISO •⟩

- `+ /1`: identity.
- `-- /1`: decrement by one.
- `++ /1`: increment by one.
- `+ /2`: addition. ⟨• ISO •⟩
- `- /2`: subtraction. ⟨• ISO •⟩
- `* /2`: multiplication. ⟨• ISO •⟩
- `// /2`: integer division. Float arguments are truncated to integers, result always integer. ⟨• ISO •⟩
- `/ /2`: division. Result always float. ⟨• ISO •⟩
- `rem/2`: integer remainder. The result is always an integer, its sign is the sign of the first argument. ⟨• ISO •⟩
- `mod/2`: modulo. The result is always a positive integer. ⟨• ISO •⟩
- `abs/1`: absolute value. ⟨• ISO •⟩
- `sign/1`: sign of. ⟨• ISO •⟩
- `float_integer_part/1`: float integer part. Result always float. ⟨• ISO •⟩
- `float_fractional_part/1`: float fractional part. Result always float. ⟨• ISO •⟩
- `truncate/1`: The result is the integer equal to the integer part of the argument. ⟨• ISO •⟩
- `integer/1`: same as `truncate/1`.
- `float/1`: conversion to float. ⟨• ISO •⟩
- `floor/1`: largest integer not greater than. ⟨• ISO •⟩
- `round/1`: integer nearest to. ⟨• ISO •⟩
- `ceiling/1`: smallest integer not smaller than. ⟨• ISO •⟩
- `** /2`: exponentiation. Result always float. ⟨• ISO •⟩
- `>> /2`: integer bitwise right shift. ⟨• ISO •⟩
- `<< /2`: integer bitwise left shift. ⟨• ISO •⟩
- `/\ /2`: integer bitwise and. ⟨• ISO •⟩
- `\/ /2`: integer bitwise or. ⟨• ISO •⟩
- `\ /1`: integer bitwise complement. ⟨• ISO •⟩
- `# /2`: integer bitwise exclusive or (xor).
- `exp/1`: exponential (e to the power of). Result always float. ⟨• ISO •⟩
- `log/1`: natural logarithm (base e). Result always float. ⟨• ISO •⟩
- `sqrt/1`: square root. Result always float. ⟨• ISO •⟩
- `sin/1`: sine. Result always float. ⟨• ISO •⟩
- `cos/1`: cosine. Result always float. ⟨• ISO •⟩
- `atan/1`: arc tangent. Result always float. ⟨• ISO •⟩
- `gcd/2`: Greatest common divisor. Arguments must evaluate to integers, result always integer.

In addition to these functors, a list of just a number evaluates to this number. Since a quoted string is just a list of integers, this allows a quoted character to be used in place of its ASCII code; e.g. `"A"` behaves within arithmetic expressions as the integer 65. Note that this is not ISO-compliant, and that can be achieved by using the ISO notation `0'A`.

Arithmetic expressions, as described above, are just data structures. If you want one evaluated you must pass it as an argument to one of the arithmetic predicates defined in this library.

Usage: `arithexpression(E)`

`E` is an arithmetic expression.

Other properties:

- *The following properties hold globally:*
  `arithexpression(Arg1)` is side-effect `free`. (`basic_props:sideff/2`)
- *If the following properties hold at call time:*
  `Arg1` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  All calls of the form `arithexpression(Arg1)` are deterministic. (`native_props:is_det/1`)
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

intexpression/1: REGTYPE

Usage: `intexpression(E)`

`E` is an integer expression.

Other properties:

- *The following properties hold globally:*
  `intexpression(Arg1)` is side-effect `free`. (`basic_props:sideff/2`)

35.3 Documentation on multifiles

$internal_error_where_term/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

35.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `native_props`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `nativeprops`, `isomodes`.

36 PART III - Assertions and auto-documentation

Author(s): The Ciao Development Team.

Ciao allows *annotating* the program code with *assertions*. Such assertions include type and instantiation mode declarations, but also more general properties as well as comments for *autodocumentation* in the *literate programming* style. These assertions document predicates (and modules and whole applications) and can be used by the Ciao tools while debugging, testing, and optimizing the program or library, and by the Ciao documenter to build program or library reference manuals.

37 The Ciao assertion language

Author(s): Manuel Hermenegildo, Francisco Bueno, German Puebla.

The `assertions` package adds a number of new declaration definitions and new operator definitions which allow including program assertions in user programs. Such assertions can be used to describe predicates, properties, modules, applications, etc. These descriptions can contain formal specifications (such as sets of preconditions, post-conditions, or descriptions of computations) as well as machine-readable textual comments.

This module is part of the `assertions` library. It defines the basic code-related assertions, i.e., those intended to be used mainly by compilation-related tools, such as the static analyzer or the run-time test generator.

Here we document mainly the use of assertions for providing specifications for predicates and other program elements, as well as the locations within assertions where machine-readable documentation strings can be placed. The commands that can be used in the documentation strings and other directives that can be used to provide additional machine-readable comments are described in the autodocumenter (`lpdoc` [Knu84, Her99]) manual.

There are two kinds of assertions: predicate assertions and program point assertions. Predicate assertions are placed as directives in the source code, i.e., preceded by "`:-`". Program point assertions are placed as literals in clause bodies. Additional documentation on the syntax and fields of predicate assertions can be found in the Chapter 38 [Types and properties related to assertions], page 199 module.

37.1 Getting more information

This documentation is intended to provide information at a "reference manual" level. For more tutorial introductions to the assertion language and more examples please see [BCHP96, HPB99, PBH00, HPBLG05, HBC12] and the `ciaopp` tutorial. % The assertion language as implemented in this library essentially follows these documents, although, due to its evolution, it may differ in some details. The purpose of this manual is to document precisely what the implementation of the library supports at any given point in time.

37.2 Some attention points

- **Formatting commands within text strings:** many of the predicates defined in these modules include arguments intended for providing textual information. This includes titles, descriptions, comments, etc. The type of this argument is a character string. In order for the automatic generation of documentation to work correctly, this character string should adhere to certain conventions. See the description of the `docstring/1` type/grammar for details.
- **Referring to variables:** In order for the automatic documentation system to work correctly, variable names (for example, when referring to arguments in the head patterns of *pred* declarations) must be surrounded by an `@var` command. For example, `@var{VariableName}` should be used for referring to the variable "VariableName", which will appear then formatted as follows: `VariableName`. See the description of the `docstring/1` type/grammar for details.

37.3 Usage and interface

- **Library usage:**

  `:- use_package(assertions).` or `:- module(...,...,[assertions]).`. Use the `assertions_basic` package to disable the implicit use of the `basic_props` module.
- **Exports:**
  - *Predicates:*

    `check/1`, `trust/1`, `true/1`, `false/1`.
- **New operators defined:**

  `=>/2` [975,xfx], `::/2` [978,xfx], `decl/1` [1150,fx], `decl/2` [1150,xfx], `pred/1` [1150,fx], `pred/2` [1150,xfx], `prop/1` [1150,fx], `prop/2` [1150,xfx], `modedef/1` [1150,fx], `calls/1` [1150,fx], `calls/2` [1150,xfx], `success/1` [1150,fx], `success/2` [1150,xfx], `comp/1` [1150,fx], `comp/2` [1150,xfx], `entry/1` [1150,fx], `exit/1` [1150,fx], `exit/2` [1150,xfx], `test/1` [1150,fx], `test/2` [1150,xfx], `texec/1` [1150,fx], `texec/2` [1150,xfx].
- **New declarations defined:**

  `pred/1`, `pred/2`, `calls/1`, `calls/2`, `success/1`, `success/2`, `comp/1`, `comp/2`, `prop/1`, `prop/2`, `test/1`, `test/2`, `texec/1`, `texec/2`, `entry/1`, `exit/1`, `exit/2`, `modedef/1`, `decl/1`, `decl/2`, `doc/2`, `comment/2`.
- **Implicit imports:**
  - *System library modules:*

    `assertions_props`.
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions/assertions_basic`.

37.4 Documentation on new declarations

pred/1: DECLARATION

`pred` assertions are the most general type of assertion, and they are used to provide information on the set of admissible calls to a predicate, the set of successes, global properties, and documentation. The body of a `pred` assertion (its only argument) contains properties or comments in different fields following the formats specified by `assrt_body/1`.

There can be more than one of these assertions per predicate, in which case each one represents a possible " mode" of use (usage) of the predicate. The exact scope of the usage is defined by the properties given for calls in the body of each assertion (which should thus distinguish the different usages intended). Predicates are typically specified using a *set* of `pred` assertions, so that together they cover all ways in which the predicate is intended be used (all usages). Each `pred` assertion is translated internally to a `calls` assertion covering all the calling modes and a `success` assertion covering the successes for those calls.

For example, the following assertions would describe all intended modes (and the only modes) of use of a predicate `length/2` (see `lists`):

```
:- pred length(L,N) : list * var => list * integer
    # "Computes the length of L.".
:- pred length(L,N) : var * integer => list * integer
    # "Outputs L of length N.".
:- pred length(L,N) : list * integer => list * integer
```

```
        # "Checks that L is of length N.".
```

Usage: :- `pred(AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

pred/2: DECLARATION

A `pred` assertion (see `pred/1`) can be preceded (as most other assertions) by an assertion status (`assrt_status/1`). If no assertion status is present (i.e., in `pred/1` assertions) the status is assumed to be `check`.

For example, the following assertion:

```
:- pred length(L,N) : list * var => list * integer.
```

is equivalent to:

```
:- check pred length(L,N) : list * var => list * integer.
```

Usage: :- `pred(AssertionStatus,AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

calls/1: DECLARATION

`calls` assertions are similar to `pred/1` assertions but the only provide information about the calls to a predicate. The set of calls assertions for a predicate describe *all* possible calls to the predicate.

For example, the following assertion describes all possible calls to predicate `is/2` (see `arithmetic`):

```
:- calls is(term,arithexpression).
```

Usage: :- `calls(AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionBody` is a call assertion body. (`assertions_props:c_assrt_body/1`)

calls/2: DECLARATION

A `calls` assertion can be preceded (as most other assertions) by an assertion status (`assrt_status/1`). If no assertion status is present (i.e., in `calls/1` assertions) the status is assumed to be `check`.

Usage: :- `calls(AssertionStatus,AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)
  `AssertionBody` is a call assertion body. (`assertions_props:c_assrt_body/1`)

success/1: DECLARATION

`success` assertions specify properties of the answers of a predicate, similarly to the corresponding `success` field of a `pred` assertion. The assertion can be limited to apply to a particular way of calling the predicate if a `calls` field is present. However, unlike `pred` or `calls` assertions, the predicate is not forced to be called only that way.

For example, the following assertion specifies the answers of the `length/2` predicate *if* it is called as in the first mode of usage above (note that the previous pred assertion already conveys such information, however it also restricts the set of admissible `calls`, while the success assertion does not):

```
:- success length(L,N) : list * var => list * integer.
```

Usage: :- `success(AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionBody` is a predicate assertion body. (`assertions_props:s_assrt_body/1`)

success/2: DECLARATION

A `success` assertion can be preceded (as most other assertions) by an assertion status (`assrt_status/1`). If no assertion status is present (i.e., in `success/1` assertions) the status is assumed to be `check`.

Usage: :- `success(AssertionStatus,AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

`AssertionBody` is a predicate assertion body. (`assertions_props:s_assrt_body/1`)

comp/1: DECLARATION

`comp` assertions specify global properties of the execution of a predicate, similarly to the corresponding `comp` field of a `pred` assertion. The assertion can be limited to apply to a particular way of calling the predicate if a `calls` field is present. However, unlike `pred` or `calls` assertions, the predicate is not forced to be called only that way.

For example, the following assertion specifies that the computation of `append/3` (see `lists`) will not fail *if* it is called as described (but does not force the predicate to be called only that way):

```
:- comp append(Xs,Ys,Zs) : var * var * var + not_fail.
```

Usage: :- `comp(AssertionBody)`.

– *The following properties should hold at call time:*

`AssertionBody` is a comp assertion body. (`assertions_props:g_assrt_body/1`)

comp/2: DECLARATION

A `comp` assertion can be preceded (as most other assertions) by an assertion status (`assrt_status/1`). If no assertion status is present (i.e., in `comp/1` assertions) the status is assumed to be `check`.

Usage: :- `comp(AssertionStatus,AssertionBody)`.

- *The following properties should hold at call time:*

  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

  `AssertionBody` is a comp assertion body. (`assertions_props:g_assrt_body/1`)

prop/1: DECLARATION

`prop` assertions are similar to a `pred/1` assertions but they flag that the predicate being documented is also a " property."

Properties are standard predicates, but which are *guaranteed to terminate for any possible instantiation state of their argument(s)*, do not perform side-effects which may interfere with the program behaviour, and do not further instantiate their arguments or add new constraints.

Provided the above holds, properties can thus be safely used as run-time checks. The program transformation used in `ciaopp` for run-time checking guarantees the third requirement. It also performs some basic checks on properties which in most cases are enough for the second requirement. However, it is the user's responsibility to guarantee termination of the properties defined. (See also Chapter 39 [Declaring regular types], page 205 for some considerations applicable to writing properties.)

The set of properties is thus a strict subset of the set of predicates. Note that properties, in addition to being used to describe characteristics of arguments in assertions, they can also be executed (called) as any other predicates.

Usage: :- `prop(AssertionBody)`.

- *The following properties should hold at call time:*

  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

prop/2: DECLARATION

This assertion is similar to a `prop/1` assertion but it is explicitely qualified. Non-qualified `prop/1` assertions are assumed the qualifier `check`.

Usage: :- `prop(AssertionStatus,AssertionBody)`.

- *The following properties should hold at call time:*

  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

test/1: DECLARATION

This assertion is similar to a success assertion but it specifies a concrete test case to be run in order verify (partially) that the predicate is working as expected. For example, the following test will verify that the length predicate works well for the particular list given:

```
:- test length(L,N) : ( L = [1,2,5,2] ) => ( N = 4 ).
```

Usage: :- `test(AssertionBody)`.

- *The following properties should hold at call time:*

  `AssertionBody` is a predicate assertion body. (`assertions_props:s_assrt_body/1`)

test/2: DECLARATION

This assertion is similar to a `test/1` assertion but it is explicitely qualified with an assertion status. Non-qualified `test/1` assertions are assumed to have `check` status. In this context, check means that the test should be executed when the developer runs the test battery.

Usage: :- `test(AssertionStatus,AssertionBody)`.

- *The following properties should hold at call time:*

  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

  `AssertionBody` is a predicate assertion body. (`assertions_props:s_assrt_body/1`)

texec/1: DECLARATION

This assertion is similar to a `calls/1` assertion but it is used to provide input data and execution commands for run-time testing.

Usage: :- `texec(AssertionBody)`.

- *The following properties should hold at call time:*

  `AssertionBody` is a call assertion body. (`assertions_props:c_assrt_body/1`)

texec/2: DECLARATION

This assertion is similar to a `texec/1` assertion but it is explicitely qualified with an assertion status. Non-qualified `texec/1` assertions are assumed to have `check` status.

Usage: :- `texec(AssertionStatus,AssertionBody)`.

- *The following properties should hold at call time:*

  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

  `AssertionBody` is a call assertion body. (`assertions_props:c_assrt_body/1`)

entry/1: DECLARATION

`entry` assertions provide information about the *external* calls to a predicate. They are identical syntactically to a `calls/1` assertion. However, they describe only the external calls, i.e., the calls to the exported predicates of a module from outside the module, or calls to the predicates in a non-modular file from other files (or the user).

These assertions are *trusted* by the compiler, i.e., they are similar to writing a `trust calls` assertion (except for referring only to the external calls). As a result, if they are erroneous they can introduce bugs in programs. Thus, `entry` assertions should be written with care.

An important use of these assertions is in providing information to the compiler which it may not be able to infer from the program. The main use is in providing information on the ways in which exported predicates of a module will be called from outside the module. This will greatly improve the precision of the analyzer, which otherwise has to assume that the arguments that exported predicates receive are any arbitrary term.

The distinction between external and internal calls is not always relevant and in those cases the use of `trust calls` assertions is preferred. Because of this, `entry` assertions may be deprecated in the future, since the distinction between external and internal calls can also be achieved by means of a bridge predicate.

Usage: :- `entry(AssertionBody)`.

– *The following properties should hold at call time:*
  `AssertionBody` is a call assertion body. (`assertions_props:c_assrt_body/1`)

exit/1: DECLARATION

This type of assertion provides information about the answers that an (exported) predicate provides for *external* calls. It is identical syntactically to a `success/1` assertion. However, it describes only external answers, i.e., answers to the exported predicates of a module from outside the module, or answers to the predicates in a non-modular file from other files (or the user). The described answers may be conditioned to a particular way of calling the predicate. E.g.:

```
:- exit length(L,N) : list * var => list * integer.
```

These assertions are *trusted* by the compiler, i.e., they are similar to writing a `trust success` assertion (except for referring only to the external calls). As a result, if they are erroneous they can introduce bugs in programs. Thus, `exit` assertions should be written with care.

The distinction between external and internal calls is not always relevant and in those cases the use of `trust success` assertions is preferred. Because of this, `entry` assertions may be deprecated in the future, since the distinction between external and internal calls can also be achieved by means of a bridge predicate.

Usage: :- `exit(AssertionBody)`.

– *The following properties should hold at call time:*
  `AssertionBody` is a predicate assertion body. (`assertions_props:s_assrt_body/1`)

exit/2: DECLARATION

exit assertion This assertion is similar to an `exit/1` assertion but it is explicitely qualified with an assertion status. Non-qualified `exit/1` assertions are assumed the qualifier `check`.

Usage: :- `exit(AssertionStatus,AssertionBody)`.

– *The following properties should hold at call time:*
  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)
  `AssertionBody` is a predicate assertion body. (`assertions_props:s_assrt_body/1`)

modedef/1: DECLARATION

This assertion is used to define modes. A mode defines in a compact way a set of call and success properties. Once defined, modes can be applied to predicate arguments in assertions. The meaning of this application is that the call and success properties defined by the mode hold for the argument to which the mode is applied. Thus, a mode is conceptually a "property macro."

The syntax of mode definitions is similar to that of `pred` declarations. For example, the following set of assertions:

```
:- modedef +A : nonvar(A) # "A is bound upon predicate entry.".

:- pred p(+A,B) : integer(A) =>  ground(B).
```

is equivalent to:

```
:- pred p(A,B) : (nonvar(A),integer(A)) =>  ground(B)
   # "A is bound upon predicate entry.".
```

Usage: :- `modedef(AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

decl/1: DECLARATION

This assertion is similar to a `pred/1` assertion but it is used to describe declarations instead of predicates.

Usage: :- `decl(AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

decl/2: DECLARATION

This assertion is similar to a `decl/1` assertion but it also has a status. Non-qualified `decl/1` assertions are assumed to have status `check`.

Usage: :- `decl(AssertionStatus,AssertionBody)`.

- *The following properties should hold at call time:*
  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)
  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

doc/2: DECLARATION

Usage: :- `doc(Pred,Comment)`.

Documentation . This assertion provides a text `Comment` for a given predicate `Pred`, as well as other directives for the documenter.

- *The following properties should hold at call time:*
  `Pred` is a head pattern. (`assertions_props:head_pattern/1`)
  `Comment` is a text comment with admissible documentation commands. The usual formatting commands that are applicable in comment strings are defined by `stringcommand/1`. See the `lpdoc` manual for documentation on comments. (`assertions_props:docstring/1`)

comment/2: DECLARATION

Usage: :- `comment(Pred,Comment)`.

An alias for `doc/2` (deprecated, for compatibility with older versions).

- *The following properties should hold at call time:*
  `Pred` is a head pattern. (`assertions_props:head_pattern/1`)
  `Comment` is a text comment with admissible documentation commands. The usual formatting commands that are applicable in comment strings are defined by `stringcommand/1`. See the `lpdoc` manual for documentation on comments. (`assertions_props:docstring/1`)
- *The following properties should hold globally:*
  DEPRECATED. (`basic_props:deprecated/1`)

37.5 Documentation on exports

check/1: PREDICATE

Usage: `check(PropertyConjunction)`

This assertion provides information on a clause program point (position in the body of a clause). Calls to a `check/1` assertion can appear in the body of a clause in any place where a literal can normally appear. The property defined by `PropertyConjunction` should hold in all the run-time stores corresponding to that program point. See also ⟨undefined⟩ [Run-time checking of assertions], page ⟨undefined⟩.

– *The following properties should hold at call time:*

  `PropertyConjunction` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (`assertions_props:property_conjunction/1`)

trust/1: PREDICATE

Usage: `trust(PropertyConjunction)`

This assertion also provides information on a clause program point. It is identical syntactically to a `check/1` assertion. However, the properties stated are not taken as something to be checked but are instead *trusted* by the compiler. While the compiler may in some cases detect an inconsistency between a `trust/1` assertion and the program, in all other cases the information given in the assertion will be taken to be true. As a result, if these assertions are erroneous they can introduce bugs in programs. Thus, `trust/1` assertions should be written with care.

An important use of these assertions is in providing information to the compiler which it may not be able to infer from the program (either because the information is not present or because the analyzer being used is not precise enough). In particular, providing information on external predicates which may not be accessible at the time of compiling the module can greatly improve the precision of the analyzer. This can be easily done with trust assertion.

– *The following properties should hold at call time:*

  `PropertyConjunction` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (`assertions_props:property_conjunction/1`)

true/1: PREDICATE

Usage: `true(PropertyConjunction)`

This assertion is identical syntactically to a `check/1` assertion. However, the properties stated have been proved to hold by the analyzer. Thus, these assertions often represent the analyzer output.

– *The following properties should hold at call time:*

  `PropertyConjunction` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (`assertions_props:property_conjunction/1`)

false/1: PREDICATE

Usage: `false(PropertyConjunction)`

This assertion is identical syntactically to a `check/1` assertion. However, the properties stated have been proved not to hold by the analyzer. Thus, these assertions often represent the analyzer output.

- *The following properties should hold at call time:*

  `PropertyConjunction` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument. (`assertions_props:property_conjunction/1`)

38 Types and properties related to assertions

Author(s): Manuel Hermenegildo.

This module is part of the `assertions` library. It provides the formal definition of the syntax of several forms of assertions and describes their meaning. It does so by defining types and properties related to the assertions themselves. The text describes, for example, the overall fields which are admissible in the bodies of assertions, where properties can be used inside these bodies, how to combine properties for a given predicate argument (e.g., conjunctions) , etc. and provides some examples.

38.1 Usage and interface

- **Library usage:**

  `:- use_module(library(assertions/assertions_props)).`
- **Exports:**
  - *Properties:*

    `head_pattern/1`, `nabody/1`, `docstring/1`.
  - *Regular Types:*

    `assrt_body/1`, `complex_arg_property/1`, `property_conjunction/1`, `property_starterm/1`, `complex_goal_property/1`, `dictionary/1`, `c_assrt_body/1`, `s_assrt_body/1`, `g_assrt_body/1`, `assrt_status/1`, `assrt_type/1`, `predfunctor/1`, `propfunctor/1`.

38.2 Documentation on exports

assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `pred/1`, `decl/1`, etc. assertions. Such a body is of the form:

```
Pr [:: DP] [: CP] [=> AP] [+ GP] [# CO]
```

where (fields between [...] are optional):

- `Pr` is a head pattern (`head_pattern/1`) which describes the predicate or property and possibly gives some implicit call/answer information.
- `DP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which expresses properties which are compatible with the predicate, i.e., instantiations made by the predicate are *compatible* with the properties in the sense that applying the property at any point would not make it fail.
- `CP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- `AP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *answers* to the predicate (if the predicate succeeds). These only apply if the (possibly empty) properties given for calls in the assertion hold.
- `GP` is a (possibly empty) complex goal property (`complex_goal_property/1`) which applies to the *whole execution* of a call to the predicate. These only apply if the (possibly empty) properties given for calls in the assertion hold.

- `CO` is a comment string (`docstring/1`). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

  See the `lpdoc` manual for documentation on assertion comments.

Usage: `assrt_body(X)`

`X` is an assertion body.

head_pattern/1: PROPERTY

A head pattern can be a predicate name (functor/arity) (`predname/1`) or a term. Thus, both `p/3` and `p(A,B,C)` are valid head patterns. In the case in which the head pattern is a term, each argument of such a term can be:

- A variable. This is useful in order to be able to refer to the corresponding argument positions by name within properties and in comments. Thus, `p(Input,Parameter,Output)` is a valid head pattern.
- A variable, as above, but preceded by a " mode." This mode determines in a compact way certain call or answer properties. For example, the head pattern `p(Input,+Parameter,Output)` is valid, as long as `+/1` is declared as a mode.

  Some useful mode definitions can be found in the `basicmodes` and `isomodes` libraries. Users can define modes using the `modedef/1` declaration.
- Any term. In this case this term determines the instantiation state of the corresponding argument position of the predicate calls to which the assertion applies.
- A ground term preceded by a " mode." The ground term determines a property of the corresponding argument. The mode determines if it applies to the calls and/or the successes. The actual property referred to is that given by the term but with one more argument added at the beginning, which is a new variable which, in a rewriting of the head pattern, appears at the argument position occupied by the term. For example, the head pattern `p(Input,+list(int),Output)` is valid for mode `+/1` defined in `library(isomodes)`, and equivalent in this case to having the head pattern `p(Input,A,Output)` and stating that the property `list(int,A)` holds for the calls of the predicate.
- Any term preceded by a " mode." In this case, only one variable is admitted, it has to be the first argument of the mode, and it represents the argument position. I.e., it plays the role of the new variable mentioned above. Thus, no rewriting of the head pattern is performed in this case. For example, the head pattern `p(Input,+(Parameter,list(int)),Output)` is valid for mode `+/2` defined in `library(isomodes)`, and equivalent in this case to having the head pattern `p(Input,Parameter,Output)` and stating that the property `list(int,Parameter)` holds for the calls of the predicate.

Usage: `head_pattern(Pr)`

`Pr` is a head pattern.

complex_arg_property/1: REGTYPE

`complex_arg_property(Props)`

`Props` is a (possibly empty) complex argument property. Such properties can appear in two formats, which are defined by `property_conjunction/1` and `property_starterm/1` respectively. The two formats can be mixed provided they are not in the same field of an assertion. I.e., the following is a valid assertion:

```
:- pred foo(X,Y) : nonvar * var => (ground(X),ground(Y)).
```

Usage: `complex_arg_property(Props)`

`Props` is a (possibly empty) complex argument property

property_conjunction/1: REGTYPE

This type defines the first, unabridged format in which properties can be expressed in the bodies of assertions. It is essentially a conjunction of properties which refer to variables. The following is an example of a complex property in this format:

- `(integer(X),list(integer,Y))`: `X` has the property `integer/1` and `Y` has the property `list/2`, with second argument `integer`.

Usage: `property_conjunction(Props)`

`Props` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. The first argument of each such term is a variable which appears as a head argument.

property_starterm/1: REGTYPE

This type defines a second, compact format in which properties can be expressed in the bodies of assertions. A `property_starterm/1` is a term whose main functor is `*/2` and, when it appears in an assertion, the number of terms joined by `*/2` is exactly the arity of the predicate it refers to. A similar series of properties as in `property_conjunction/1` appears, but the arity of each property is one less: the argument position to which they refer (first argument) is left out and determined by the position of the property in the `property_starterm/1`. The idea is that each element of the `*/2` term corresponds to a head argument position. Several properties can be assigned to each argument position by grouping them in curly brackets. The following is an example of a complex property in this format:

- `integer * list(integer)`: the first argument of the procedure (or function, or ...) has the property `integer/1` and the second one has the property `list/2`, with second argument `integer`.
- `{integer,var} * list(integer)`: the first argument of the procedure (or function, or ...) has the properties `integer/1` and `var/1` and the second one has the property `list/2`, with second argument `integer`.

Usage: `property_starterm(Props)`

`Props` is either a term or several terms separated by `*/2`. The main functor of each of those terms corresponds to that of the definition of a property, and the arity should be one less than in the definition of such property. All arguments of each such term are ground.

complex_goal_property/1: REGTYPE

`complex_goal_property(Props)`

`Props` is a (possibly empty) complex goal property. Such properties can be either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. Such properties apply to all executions of all goals of the predicate which comply with the assertion in which the `Props` appear.

The arguments of the terms in `Props` are implicitely augmented with a first argument which corresponds to a goal of the predicate of the assertion in which the `Props` appear. For example, the assertion

```
:- comp var(A) + not_further_inst(A).
```

has property `not_further_inst/1` as goal property, and establishes that in all executions of `var(A)` it should hold that `not_further_inst(var(A),A)`.

Usage: `complex_goal_property(Props)`

`Props` is either a term or a *conjunction* of terms. The main functor and arity of each of those terms corresponds to the definition of a property. A first implicit argument in such terms identifies goals to which the properties apply.

nabody/1: PROPERTY

Usage: `nabody(ABody)`

`ABody` is a normalized assertion body.

dictionary/1: REGTYPE

Usage: `dictionary(D)`

`D` is a dictionary of variable names.

c_assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `call/1`, `entry/1`, etc. assertions. The following are admissible:

```
Pr : CP [# CO]
```

where (fields between [...] are optional):

- `CP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- `CO` is a comment string (`docstring/1`). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `n_assrt_body/5` and its auxiliary types.

Usage: `c_assrt_body(X)`

`X` is a call assertion body.

s_assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `pred/1`, `func/1`, etc. assertions. The following are admissible:

```
Pr : CP => AP # CO
Pr : CP => AP
Pr => AP # CO
Pr => AP
```

where:

- `Pr` is a head pattern (`head_pattern/1`) which describes the predicate or property and possibly gives some implicit call/answer information.
- `CP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- `AP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *answers* to the predicate (if the predicate succeeds). These only apply if the (possibly empty) properties given for calls in the assertion hold.
- `CO` is a comment string (`docstring/1`). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `n_assrt_body/5` and its auxiliary types.

Usage: `s_assrt_body(X)`

`X` is a predicate assertion body.

g_assrt_body/1: REGTYPE

This predicate defines the different types of syntax admissible in the bodies of `comp/1` assertions. The following are admissible:

```
Pr : CP + GP # CO
Pr : CP + GP
Pr + GP # CO
Pr + GP
```

where:

- `Pr` is a head pattern (`head_pattern/1`) which describes the predicate or property and possibly gives some implicit call/answer information.
- `CP` is a (possibly empty) complex argument property (`complex_arg_property/1`) which applies to the *calls* to the predicate.
- `GP` contains (possibly empty) complex goal property (`complex_goal_property/1`) which applies to the *whole execution* of a call to the predicate. These only apply if the (possibly empty) properties given for calls in the assertion hold.
- `CO` is a comment string (`docstring/1`). This comment only applies if the (possibly empty) properties given for calls in the assertion hold. The usual formatting commands that are applicable in comment strings can be used (see `stringcommand/1`).

The format of the different parts of the assertion body are given by `n_assrt_body/5` and its auxiliary types.

Usage: `g_assrt_body(X)`

`X` is a comp assertion body.

assrt_status/1: REGTYPE

The types of assertion status. They have the same meaning as the program-point assertions, and are as follows:

```
assrt_status(true).
assrt_status(false).
assrt_status(check).
assrt_status(checked).
```

```
assrt_status(trust).
```

Usage: `assrt_status(X)`

`X` is an acceptable status for an assertion.

assrt_type/1: REGTYPE

The admissible kinds of assertions:

```
assrt_type(pred).
assrt_type(prop).
assrt_type(decl).
assrt_type(func).
assrt_type(calls).
assrt_type(success).
assrt_type(comp).
assrt_type(entry).
assrt_type(exit).
assrt_type(test).
assrt_type(texec).
assrt_type(modedef).
```

Usage: `assrt_type(X)`

`X` is an admissible kind of assertion.

predfunctor/1: REGTYPE

Usage: `predfunctor(X)`

`X` is a type of assertion which defines a predicate.

propfunctor/1: REGTYPE

Usage: `propfunctor(X)`

`X` is a type of assertion which defines a *property*.

docstring/1: PROPERTY

Usage: `docstring(String)`

`String` is a text comment with admissible documentation commands. The usual formatting commands that are applicable in comment strings are defined by `stringcommand/1`. See the `lpdoc` manual for documentation on comments.

38.3 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

39 Declaring regular types

Author(s): Manuel Hermenegildo, Pedro López, Francisco Bueno.

This library package adds declarations and new operator definitions which provide simple syntactic sugar to write regular type definitions in source code. Regular types are just properties which have the additional characteristic of being regular types (`basic_props:regtype/1`), defined below.

For example, this library package allows writing:

```
:- regtype tree(X) # "X is a tree.".
```

instead of the more cumbersome:

```
:- prop tree(X) + regtype # "X is a tree.".
```

Regular types can be used as properties to describe predicates and play an essential role in program debugging (see the Ciao Prolog preprocessor (`ciaopp`) manual).

In this chapter we explain some general considerations worth taking into account when writing properties in general, not just regular types.

39.1 Defining properties

Given the classes of assertions in the Ciao assertion language, there are two fundamental classes of properties. Properties used in assertions which refer to execution states (i.e., `calls/1`, `success/1`, and the like) are called *properties of execution states*. Properties used in assertions related to computations (i.e., `comp/1`) are called *properties of computations*. Different considerations apply when writing a property of the former or of the latter kind.

Consider a definition of the predicate `string_concat/3` which concatenates two character strings (represented as lists of ASCII codes):

```
string_concat([],L,L).
string_concat([X|Xs],L,[X|NL]):- string_concat(Xs,L,NL).
```

Assume that we would like to state in an assertion that each argument "is a list of integers." However, we must decide which one of the following two possibilities we mean exactly: "the argument is *instantiated* to a list of integers" (let us call this property `instantiated_to_intlist/1`), or "if any part of the argument is instantiated, this instantiation must be compatible with the argument being a list of integers" (we will call this property `compatible_with_intlist/1`). For example, `instantiated_to_intlist/1` should be true for the terms `[]` and `[1,2]`, but should not for `X`, `[a,2]`, and `[X,2]`. In turn, `compatible_with_intlist/1` should be true for `[]`, `X`, `[1,2]`, and `[X,2]`, but should not be for `[X|1]`, `[a,2]`, and `1`. We refer to properties such as `instantiated_to_intlist/1` above as *instantiation properties* and to those such as `compatible_with_intlist/1` as *compatibility properties* (corresponding to the traditional notions of "instantiation types" and "compatibility types").

It turns out that both of these notions are quite useful in practice. In the example above, we probably would like to use `compatible_with_intlist/1` to state that on success of `string_concat/3` all three argument must be compatible with lists of integers in an assertion like:

```
:- success string_concat(A,B,C) => ( compatible_with_intlist(A),
                                     compatible_with_intlist(B),
                                     compatible_with_intlist(C) ).
```

With this assertion, no error will be flagged for a call to `string_concat/3` such as `string_concat([20],L,R)`, which on success produces the resulting atom `string_concat([20],L,[20|L])`, but a call `string_concat([],a,R)` would indeed flag an error.

On the other hand, and assuming that we are running on a Prolog system, we would probably like to use `instantiated_to_intlist/1` for `sumlist/2` as follows:

```
:- calls sumlist(L,N) : instantiated_to_intlist(L).

sumlist([],0).
sumlist([X|R],S) :- sumlist(R,PS), S is PS+X.
```

to describe the type of calls for which the program has been designed, i.e., those in which the first argument of `sumlist/2` is indeed a list of integers.

The property `instantiated_to_intlist/1` might be written as in the following (Prolog) definition:

```
:- prop instantiated_to_intlist/1.

instantiated_to_intlist(X) :-
       nonvar(X), instantiated_to_intlist_aux(X).

instantiated_to_intlist_aux([]).
instantiated_to_intlist_aux([X|T]) :-
       integer(X), instantiated_to_intlist(T).
```

(Recall that the Prolog builtin `integer/1` itself implements an instantiation check, failing if called with a variable as the argument.)

The property `compatible_with_intlist/1` might in turn be written as follows (also in Prolog):

```
:- prop compatible_with_intlist/1.

compatible_with_intlist(X) :- var(X).
compatible_with_intlist(X) :-
       nonvar(X), compatible_with_intlist_aux(X).

compatible_with_intlist_aux([]).
compatible_with_intlist_aux([X|T]) :-
       int_compat(X), compatible_with_intlist(T).

int_compat(X) :- var(X).
int_compat(X) :- nonvar(X), integer(X).
```

Note that these predicates meet the criteria for being properties and thus the `prop/1` declaration is correct.

Ensuring that a property meets the criteria for "not affecting the computation" can sometimes make its coding somewhat tedious. In some ways, one would like to be able to write simply:

```
intlist([]).
intlist([X|R]) :- int(X), intlist(R).
```

(Incidentally, note that the above definition, provided that it suits the requirements for being a property and that `int/1` is a regular type, meets the criteria for being a regular type. Thus, it could be declared `:- regtype intlist/1`.)

But note that (independently of the definition of `int/1`) the definition above is not the correct instantiation check, since it would succeed for a call such as `intlist(X)`. In fact, it is not strictly correct as a compatibility property either, because, while it would fail or succeed as expected, it would perform instantiations (e.g., if called with `intlist(X)` it would bind `X` to `[]`). In practice, it is convenient to provide some run-time support to aid in this task.

The run-time support of the Ciao system (see ⟨undefined⟩ [Run-time checking of assertions], page ⟨undefined⟩) ensures that the execution of properties is performed in such a way that properties written as above can be used directly as instantiation checks. Thus, writing:

```
:- calls sumlist(L,N) : intlist(L).
```

has the desired effect. Also, the same properties can often be used as compatibility checks by writing them in the assertions as `compat(Property)` (`basic_props:compat/1`). Thus, writing:

```
:- success string_concat(A,B,C) => ( compat(intlist(A)),
                                     compat(intlist(B)),
                                     compat(intlist(C)) ).
```

also has the desired effect.

As a general rule, the properties that can be used directly for checking for compatibility should be *downwards closed*, i.e., once they hold they will keep on holding in every state accessible in forwards execution. There are certain predicates which are inherently *instantiation* checks and should not be used as *compatibility* properties nor appear in the definition of a property that is to be used with `compat`. Examples of such predicates (for Prolog) are `==`, `ground`, `nonvar`, `integer`, `atom`, `>`, etc. as they require a certain instantiation degree of their arguments in order to succeed.

In contrast with properties of execution states, *properties of computations* refer to the entire execution of the call(s) that the assertion relates to. One such property is, for example, `not_fails/1` (note that although it has been used as in `:- comp append(Xs,Ys,Zs) + not_fails`, it is in fact read as `not_fails(append(Xs,Ys,Zs))`; see `assertions_props:complex_goal_property/1`). For this property, which should be interpreted as "execution of the predicate either succeeds at least once or loops," we can use the following predicate `not_fails/1` for run-time checking:

```
not_fails(Goal):-
      if( call(Goal),
          true,            %% then
          warning(Goal) ). %% else
```

where the `warning/1` (library) predicate simply prints a warning message.

In this simple case, implementation of the predicate is not very difficult using the (non-standard) `if/3` builtin predicate present in many Prolog systems.

However, it is not so easy to code predicates which check other properties of the computation and we may in general need to program a meta-interpreter for this purpose.

39.2 Usage and interface

- **Library usage:**
  `:- use_package(regtypes).`
  or
  `:- module(...,...,[regtypes]).`
- **New operators defined:**
  `regtype/1` [1150,fx], `regtype/2` [1150,xfx].
- **New declarations defined:**
  `regtype/1`, `regtype/2`.
- **Implicit imports:**
  - *System library modules:*
    `assertions_props`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

39.3 Documentation on new declarations

regtype/1: DECLARATION

This assertion is similar to a prop assertion but it flags that the property being documented is also a " regular type." Regular types are properties whose definitions are *regular programs* (see below). This allows for example checking whether it is in the class of types supported by the regular type checking and inference modules.

A regular program is defined by a set of clauses, each of the form:

```
p(v_1, ..., v_n, x)  :- body_1, ..., body_k.
```

where:

1. `x` is a term whose variables (which are called *term variables*) are unique, i.e., it is not allowed to introduce equality constraints between the variables of `x`.

   For example, `p(f(X, Y)) :- ...` is valid, but `p(f(X, X)) :- ...` is not.
2. in all clauses defining `p/n+1` the terms `x` do not unify except maybe for one single clause in which `x` is a variable.
3. `n` >= 0 and `p/n` is a *parametric type functor* (whereas the predicate defined by the clauses is `p/n+1`).
4. `v_1`, ..., `v_n` are unique variables, which are called *parametric variables*.
5. Each `body_i` is of the form:
   1. `t(z)` where `z` is one of the *term variables* and `t` is a *regular type expression*;
   2. `q(t_1, ..., t_m, y)` where `m` >= 0, `q/m` is a *parametric type functor*, not in the set of functors `=/2`, `^/2`, `./3`.

      `t_1, ..., t_m` are *regular type expressions*, and `y` is a *term variable*.
6. Each term variable occurs at most once in the clause's body (and should be as the last argument of a literal).

A *regular type expression* is either a parametric variable or a parametric type functor applied to some of the parametric variables.

A parametric type functor is a regular type, defined by a regular program, or a basic type. Basic types are defined in Chapter 40 [Basic data types and properties], page 209.

The set of regular types is thus a well defined subset of the set of properties. Note that types can be used to describe characteristics of arguments in assertions and they can also be executed (called) as any other predicates.

Usage: :- `regtype(AssertionBody)`.

- *The following properties should hold at call time:*

  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

regtype/2: DECLARATION

This assertion is similar to a `regtype/1` assertion but it is explicitely qualified. Non-qualified `regtype/1` assertions are assumed the qualifier `check`. Note that checking regular type definitions should be done with the `ciaopp` preprocessor.

Usage: :- `regtype(AssertionStatus,AssertionBody)`.

- *The following properties should hold at call time:*

  `AssertionStatus` is an acceptable status for an assertion. (`assertions_props:assrt_status/1`)

  `AssertionBody` is an assertion body. (`assertions_props:assrt_body/1`)

40 Basic data types and properties

Author(s): Daniel Cabeza, Manuel Hermenegildo.

This library contains the set of basic properties used by the builtin predicates, and which constitute the basic data types and properties of the language. They are intented to be used as properties in assertions (for both runtime and compile time checking).

Note that most of those properties can be used directly from goals as type testing builtins, and they should be correct when terms are sufficiently instantiated. Calling with uninstantiated terms have undefined behaviour (failure or non termination, depending on the type). Please use the run-time checking facilities (`inst/2` and `compat/2`) to ensure well-defined behaviour. For low-level instantiation checks we encourage the use `term_typing` builtins.

40.1 Usage and interface

- **Library usage:**

  This module is automatically imported by the `assertions` package. It can be imported explicitly with the `:- use_module(engine(basic_props))` directive.
- **Exports:**
  - *Predicates:*

    `check/1`, `trust/1`, `true/1`, `false/1`.
  - *Properties:*

    `callable/1`, `member/2`, `compat/2`, `inst/2`, `iso/1`, `deprecated/1`, `srcloc/4`, `example/1`, `not_further_inst/2`, `sideff/2`, `regtype/1`, `native/1`, `native/2`, `rtcheck/1`, `rtcheck/2`, `no_rtcheck/1`, `eval/1`, `equiv/2`, `bind_ins/1`, `error_free/1`, `memo/1`, `filter/2`, `pe_type/1`.
  - *Regular Types:*

    `term/1`, `int/1`, `nnegint/1`, `flt/1`, `num/1`, `atm/1`, `struct/1`, `gnd/1`, `gndstr/1`, `constant/1`, `cgoal/1`, `internal_module_id/1`, `operator_specifier/1`, `list/1`, `list/2`, `nlist/2`, `sequence/2`, `sequence_or_list/2`, `character_code/1`, `string/1`, `bytelist/1`, `predname/1`, `atm_or_atm_list/1`, `flag_values/1`.

40.2 Documentation on exports

term/1: REGTYPE

The most general type (includes all possible terms).

Usage: `term(X)`

`X` is any term.

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`term(X)`

- *The following properties hold globally:*

  `term(X)` is side-effect `free`. (`basic_props:sideff/2`)

`term(X)`

– *The following properties hold globally:*
  `term(X)` is evaluable at compile-time. (`basic_props:eval/1`)

`term(X)`
– *The following properties hold globally:*
  `term(X)` is equivalent to `true`. (`basic_props:equiv/2`)

int/1: REGTYPE

The type of integers. The range of integers is `[-2^2147483616, 2^2147483616)`. Thus for all practical purposes, the range of integers can be considered infinite.

Usage: `int(T)`

`T` is an integer.
– *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`int(T)`
– *The following properties hold globally:*
  `int(T)` is side-effect `free`. (`basic_props:sideff/2`)

`int(T)`
– *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `int(T)` is evaluable at compile-time. (`basic_props:eval/1`)
  All calls of the form `int(T)` are deterministic. (`native_props:is_det/1`)

`int(T)`
– *The following properties hold upon exit:*
  `T` is an integer. (`basic_props:int/1`)
– *The following properties hold globally:*
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

nnegint/1: REGTYPE

The type of non-negative integers, i.e., natural numbers.

Usage: `nnegint(T)`

`T` is a non-negative integer.
– *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`nnegint(T)`
– *The following properties hold globally:*
  `nnegint(T)` is side-effect `free`. (`basic_props:sideff/2`)

`nnegint(T)`

- *If the following properties hold at call time:*

  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  then the following properties hold globally:

  `nnegint(T)` is evaluable at compile-time. (`basic_props:eval/1`)

`nnegint(T)`

- *The following properties hold upon exit:*

  `T` is a non-negative integer. (`basic_props:nnegint/1`)
- *The following properties hold globally:*

  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

flt/1: REGTYPE

The type of floating-point numbers. The range of floats is the one provided by the C `double` type, typically `[4.9e-324, 1.8e+308]` (plus or minus). There are also three special values: Infinity, either positive or negative, represented as `1.0e1000` and `-1.0e1000`; and Not-a-number, which arises as the result of indeterminate operations, represented as `0.Nan`

Usage: `flt(T)`

`T` is a float.

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`flt(T)`

- *The following properties hold globally:*

  `flt(T)` is side-effect `free`. (`basic_props:sideff/2`)

`flt(T)`

- *If the following properties hold at call time:*

  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  then the following properties hold globally:

  `flt(T)` is evaluable at compile-time. (`basic_props:eval/1`)

  All calls of the form `flt(T)` are deterministic. (`native_props:is_det/1`)

`flt(T)`

- *The following properties hold upon exit:*

  `T` is a float. (`basic_props:flt/1`)
- *The following properties hold globally:*

  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

num/1: REGTYPE

The type of numbers, that is, integer or floating-point.

Usage: `num(T)`

`T` is a number.

– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`num(T)`

– *The following properties hold globally:*
`num(T)` is side-effect `free`. (`basic_props:sideff/2`)
`num(T)` is binding insensitive. (`basic_props:bind_ins/1`)

`num(T)`

– *If the following properties hold at call time:*
`T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
then the following properties hold globally:
`num(T)` is evaluable at compile-time. (`basic_props:eval/1`)
All calls of the form `num(T)` are deterministic. (`native_props:is_det/1`)

`num(T)`

– *The following properties hold upon exit:*
`T` is a number. (`basic_props:num/1`)

– *The following properties hold globally:*
Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

atm/1: REGTYPE

The type of atoms, or non-numeric constants. The size of atoms is unbound.

Usage: `atm(T)`

`T` is an atom.

– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`atm(T)`

– *The following properties hold globally:*
`atm(T)` is side-effect `free`. (`basic_props:sideff/2`)

`atm(T)`

– *If the following properties hold at call time:*
`T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
then the following properties hold globally:
`atm(T)` is evaluable at compile-time. (`basic_props:eval/1`)
All calls of the form `atm(T)` are deterministic. (`native_props:is_det/1`)

`atm(T)`

– *The following properties hold upon exit:*
`T` is an atom. (`basic_props:atm/1`)

– *The following properties hold globally:*
Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

struct/1: REGTYPE

The type of compound terms, or terms with non-zeroary functors. By now there is a limit of 255 arguments.

Usage: `struct(T)`

`T` is a compound term.

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`struct(T)`

- *The following properties hold globally:*
  `struct(T)` is side-effect `free`. (`basic_props:sideff/2`)

`struct(T)`

- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `struct(T)` is evaluable at compile-time. (`basic_props:eval/1`)

`struct(T)`

- *The following properties hold upon exit:*
  `T` is a compound term. (`basic_props:struct/1`)

gnd/1: REGTYPE

The type of all terms without variables.

Usage: `gnd(T)`

`T` is ground.

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`gnd(T)`

- *The following properties hold globally:*
  `gnd(T)` is side-effect `free`. (`basic_props:sideff/2`)

`gnd(T)`

- *If the following properties hold at call time:*
  `T` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `gnd(T)` is evaluable at compile-time. (`basic_props:eval/1`)
  All calls of the form `gnd(T)` are deterministic. (`native_props:is_det/1`)

`gnd(T)`

- *The following properties hold upon exit:*
  `T` is ground. (`basic_props:gnd/1`)
- *The following properties hold globally:*
  Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`native_props:test_type/2`)

gndstr/1: REGTYPE

Usage: `gndstr(T)`

`T` is a ground compound term.

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`gndstr(T)`

- *The following properties hold globally:*
  `gndstr(T)` is side-effect `free`. (`basic_props:sideff/2`)

`gndstr(T)`

- *If the following properties hold at call time:*
  `T` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `gndstr(T)` is evaluable at compile-time. (`basic_props:eval/1`)
  All calls of the form `gndstr(T)` are deterministic. (`native_props:is_det/1`)

`gndstr(T)`

- *The following properties hold upon exit:*
  `T` is a ground compound term. (`basic_props:gndstr/1`)

constant/1: REGTYPE

Usage: `constant(T)`

`T` is an atomic term (an atom or a number).

Other properties:

`constant(T)`

- *The following properties hold globally:*
  `constant(T)` is side-effect `free`. (`basic_props:sideff/2`)

`constant(T)`

- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `constant(T)` is evaluable at compile-time. (`basic_props:eval/1`)
  All calls of the form `constant(T)` are deterministic. (`native_props:is_det/1`)

`constant(T)`

- *The following properties hold upon exit:*
  `T` is an atomic term (an atom or a number). (`basic_props:constant/1`)

cgoal/1: REGTYPE

Usage: `cgoal(T)`

`T` is a term which represents a goal, i.e., an atom or a structure.

Other properties:

`cgoal(T)`

- *The following properties hold globally:*
  `cgoal(T)` is side-effect `free`. (`basic_props:sideff/2`)

`cgoal(T)`
- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `cgoal(T)` is evaluable at compile-time. (`basic_props:eval/1`)
  All calls of the form `cgoal(T)` are deterministic. (`native_props:is_det/1`)

`cgoal(T)`
- *The following properties hold upon exit:*
  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

callable/1: PROPERTY

Usage: `callable(T)`

`T` is instantiated with an atom or a structure (also known as callable).

Other properties:

`callable(T)`
- *The following properties hold globally:*
  `callable(T)` is side-effect `free`. (`basic_props:sideff/2`)

`callable(T)`
- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `callable(T)` is evaluable at compile-time. (`basic_props:eval/1`)
  All calls of the form `callable(T)` are deterministic. (`native_props:is_det/1`)

`callable(T)`
- *The following properties hold upon exit:*
  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

internal_module_id/1: REGTYPE

For a user file it is a term user/1 with an argument different for each user file, for other modules is just the name of the module (as an atom).

Usage: `internal_module_id(M)`

`M` is an internal module identifier

operator_specifier/1: REGTYPE

The type and associativity of an operator is described by the following mnemonic atoms:

`xfx` Infix, non-associative: it is a requirement that both of the two subexpressions which are the arguments of the operator must be of *lower* precedence than the operator itself.

`xfy` Infix, right-associative: only the first (left-hand) subexpression must be of lower precedence; the right-hand subexpression can be of the *same* precedence as the main operator.

`yfx` Infix, left-associative: same as above, but the other way around.

`fx` Prefix, non-associative: the subexpression must be of *lower* precedence than the operator.

`fy` Prefix, associative: the subexpression can be of the *same* precedence as the operator.

`xf` Postfix, non-associative: the subexpression must be of *lower* precedence than the operator.

`yf` Postfix, associative: the subexpression can be of the *same* precedence as the operator.

Usage: `operator_specifier(X)`

`X` specifies the type and associativity of an operator.

Other properties:

`operator_specifier(X)`

- *The following properties hold globally:*

  `operator_specifier(X)` is side-effect `free`. (`basic_props:sideff/2`)

`operator_specifier(X)`

- *If the following properties hold at call time:*

  `X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  then the following properties hold globally:

  `operator_specifier(X)` is evaluable at compile-time. (`basic_props:eval/1`)

  All calls of the form `operator_specifier(X)` are deterministic. (`native_props:is_det/1`)

  Goal `operator_specifier(X)` produces `7` solutions. (`native_props:relations/2`)

`operator_specifier(T)`

- *The following properties hold upon exit:*

  `T` specifies the type and associativity of an operator. (`basic_props:operator_specifier/1`)

list/1: REGTYPE

A list is formed with successive applications of the functor `'.'/2`, and its end is the atom `[]`. Defined as

```
list([]).
list([_1|L]) :-
    list(L).
```

Usage: `list(L)`

`L` is a list.

Other properties:

`list(L)`

- *The following properties hold globally:*

  `list(L)` is side-effect `free`. (`basic_props:sideff/2`) `list(L)`

– *If the following properties hold at call time:*
`L` is currently ground (it contains no variables). (`term_typing:ground/1`)
then the following properties hold globally:
`list(L)` is evaluable at compile-time. (`basic_props:eval/1`)
All calls of the form `list(L)` are deterministic. (`native_props:is_det/1`)

`list(T)`

– *The following properties hold upon exit:*
`T` is a list. (`basic_props:list/1`)

list/2: REGTYPE

`list(T,L)`

`L` is a list, and for all its elements, `T` holds.

Usage: `list(T,L)`

`L` is a list of `T`s.

Meta-predicate with arguments: `list(pred(1),?)`.

Other properties:

`list(T,L)`

– *The following properties hold globally:*
`list(T,L)` is side-effect `free`. (`basic_props:sideff/2`)

`list(T,L)`

– *If the following properties hold at call time:*
`T` is currently ground (it contains no variables). (`term_typing:ground/1`)
`L` is currently ground (it contains no variables). (`term_typing:ground/1`)
then the following properties hold globally:
`list(T,L)` is evaluable at compile-time. (`basic_props:eval/1`)

`list(T,X)`

– *The following properties hold upon exit:*
`X` is a list. (`basic_props:list/1`)

nlist/2: REGTYPE

Usage: `nlist(T,L)`

`L` is `T` or a nested list of `T`s. Note that if `T` is term, this type is equivalent to term, this fact explains why we do not have an `nlist/1` type.

Meta-predicate with arguments: `nlist(pred(1),?)`.

Other properties:

`nlist(T,L)`

– *The following properties hold globally:*
`nlist(T,L)` is side-effect `free`. (`basic_props:sideff/2`)

`nlist(T,L)`

- *If the following properties hold at call time:*
  T is currently ground (it contains no variables). (term_typing:ground/1)
  L is currently ground (it contains no variables). (term_typing:ground/1)
  then the following properties hold globally:
  nlist(T,L) is evaluable at compile-time. (basic_props:eval/1)

nlist(T,X)
- *The following properties hold upon exit:*
  X is any term. (basic_props:term/1)

member/2: PROPERTY

Usage: member(X,L)

X is an element of L.

Other properties:

member(X,L)
- *The following properties hold globally:*
  member(X,L) is side-effect free. (basic_props:sideff/2)
  member(X,L) is binding insensitive. (basic_props:bind_ins/1)

member(X,L)
- *If the following properties hold at call time:*
  L is a list. (basic_props:list/1)
  then the following properties hold globally:
  member(X,L) is evaluable at compile-time. (basic_props:eval/1)

member(_X,L)
- *The following properties hold upon exit:*
  L is a list. (basic_props:list/1)

member(X,L)
- *If the following properties hold at call time:*
  L is currently ground (it contains no variables). (term_typing:ground/1)
  then the following properties hold upon exit:
  X is currently ground (it contains no variables). (term_typing:ground/1)

sequence/2: REGTYPE

A sequence is formed with zero, one, or more occurrences of the operator ','/2. For example, a, b, c is a sequence of three atoms, a is a sequence of one atom.

Usage: sequence(T,S)

S is a sequence of Ts.

Meta-predicate with arguments: sequence(pred(1),?).

Other properties:

sequence(T,S)
- *The following properties hold globally:*
  sequence(T,S) is side-effect free. (basic_props:sideff/2)

`sequence(T,S)`
- *If the following properties hold at call time:*
  `T` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `S` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `sequence(T,S)` is evaluable at compile-time. (`basic_props:eval/1`)

`sequence(T,E)`
- *The following properties hold upon exit:*
  `T` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `E` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

sequence_or_list/2: REGTYPE

Usage: `sequence_or_list(T,S)`

`S` is a sequence or list of `T`s.

Meta-predicate with arguments: `sequence_or_list(pred(1),?)`.

Other properties:

`sequence_or_list(T,S)`
- *The following properties hold globally:*
  `sequence_or_list(T,S)` is side-effect `free`. (`basic_props:sideff/2`)

`sequence_or_list(T,S)`
- *If the following properties hold at call time:*
  `T` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `S` is currently ground (it contains no variables). (`term_typing:ground/1`)
  then the following properties hold globally:
  `sequence_or_list(T,S)` is evaluable at compile-time. (`basic_props:eval/1`)

`sequence_or_list(T,E)`
- *The following properties hold upon exit:*
  `T` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `E` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

character_code/1: REGTYPE

Usage: `character_code(T)`

`T` is an integer which is a character code.

Other properties:

`character_code(T)`
- *The following properties hold globally:*
  `character_code(T)` is side-effect `free`. (`basic_props:sideff/2`)

`character_code(T)`
- *If the following properties hold at call time:*
  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `character_code(T)` is evaluable at compile-time. (`basic_props:eval/1`)

`character_code(I)`

– *The following properties hold upon exit:*

`I` is an integer which is a character code. (`basic_props:character_code/1`)

string/1: REGTYPE

A string is a list of character codes. The usual syntax for strings `"string"` is allowed, which is equivalent to `[0's,0't,0'r,0'i,0'n,0'g]` or `[115,116,114,105,110,103]`. There is also a special Ciao syntax when the list is not complete: `"st"||R` is equivalent to `[0's,0't|R]`.

Usage: `string(T)`

`T` is a string (a list of character codes).

Other properties:

`string(T)`

– *The following properties hold globally:*

`string(T)` is side-effect `free`. (`basic_props:sideff/2`)

`string(T)`

– *If the following properties hold at call time:*

`T` is currently ground (it contains no variables). (`term_typing:ground/1`)

then the following properties hold globally:

`string(T)` is evaluable at compile-time. (`basic_props:eval/1`)

`string(T)`

– *The following properties hold upon exit:*

`T` is a string (a list of character codes). (`basic_props:string/1`)

bytelist/1: REGTYPE

Usage: `bytelist(T)`

`T` is list of bytes.

Other properties:

`bytelist(T)`

– *The following properties hold globally:*

`bytelist(T)` is side-effect `free`. (`basic_props:sideff/2`)

`bytelist(T)`

– *If the following properties hold at call time:*

`T` is currently ground (it contains no variables). (`term_typing:ground/1`)

then the following properties hold globally:

`bytelist(T)` is evaluable at compile-time. (`basic_props:eval/1`)

`bytelist(T)`

– *The following properties hold upon exit:*

`T` is list of bytes. (`basic_props:bytelist/1`)

predname/1: REGTYPE

`predname(P)`

`P` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
    atm(P),
    nnegint(A).
```

Usage: `predname(P)`

`P` is a predicate name.

Other properties:

`predname(P)`

– *The following properties hold globally:*
`predname(P)` is side-effect `free`. (`basic_props:sideff/2`)

`predname(P)`

– *If the following properties hold at call time:*
`P` is currently ground (it contains no variables). (`term_typing:ground/1`)
then the following properties hold globally:
`predname(P)` is evaluable at compile-time. (`basic_props:eval/1`)

`predname(P)`

– *The following properties hold upon exit:*
`P` is a predicate name. (`basic_props:predname/1`)

atm_or_atm_list/1: REGTYPE

Usage: `atm_or_atm_list(T)`

`T` is an atom or a list of atoms.

Other properties:

`atm_or_atm_list(T)`

– *The following properties hold globally:*
`atm_or_atm_list(T)` is side-effect `free`. (`basic_props:sideff/2`)

`atm_or_atm_list(T)`

– *If the following properties hold at call time:*
`T` is currently ground (it contains no variables). (`term_typing:ground/1`)
then the following properties hold globally:
`atm_or_atm_list(T)` is evaluable at compile-time. (`basic_props:eval/1`)

`atm_or_atm_list(T)`

– *The following properties hold upon exit:*
`T` is an atom or a list of atoms. (`basic_props:atm_or_atm_list/1`)

compat/2: PROPERTY

This property captures the notion of type or property compatibility. The instantiation or constraint state of the term is compatible with the given property, in the sense that assuming that imposing that property on the term does not render the store inconsistent.

For example, terms `X` (i.e., a free variable), `[Y|Z]`, and `[Y,Z]` are all compatible with the regular type `list/1`, whereas the terms `f(a)` and `[1|2]` are not.

Usage: `compat(Term,Prop)`

`Term` is *compatible* with `Prop`.

Meta-predicate with arguments: `compat(?,pred(1))`.

Other properties:

`compat(Term,Prop)`

– *If the following properties hold at call time:*

`Term` is currently ground (it contains no variables). (`term_typing:ground/1`)

`Prop` is currently ground (it contains no variables). (`term_typing:ground/1`)

then the following properties hold globally:

`compat(Term,Prop)` is evaluable at compile-time. (`basic_props:eval/1`)

inst/2: PROPERTY

Usage: `inst(Term,Prop)`

`Term` is instantiated enough to satisfy `Prop`.

Meta-predicate with arguments: `inst(?,pred(1))`.

Other properties:

`inst(Term,Prop)`

– *The following properties hold globally:*

`inst(Term,Prop)` is side-effect `free`. (`basic_props:sideff/2`)

`inst(Term,Prop)`

– *If the following properties hold at call time:*

`Term` is currently ground (it contains no variables). (`term_typing:ground/1`)

`Prop` is currently ground (it contains no variables). (`term_typing:ground/1`)

then the following properties hold globally:

`inst(Term,Prop)` is evaluable at compile-time. (`basic_props:eval/1`)

iso/1: PROPERTY

Specifies that the predicate and usage marked with this global property complies with the ISO-Prolog standard.

Usage: `iso(G)`

Complies with the ISO-Prolog standard.

Meta-predicate with arguments: `iso(goal)`.

Other properties:

`iso(G)`

– *The following properties hold globally:*

`iso(G)` is side-effect `free`. (`basic_props:sideff/2`)

deprecated/1: PROPERTY

Specifies that the predicate marked with this global property has been deprecated, i.e., its use is not recommended any more since it will be deleted at a future date. Typically this is done because its functionality has been superseded by another predicate.

Usage: `deprecated(G)`

DEPRECATED.

Meta-predicate with arguments: `deprecated(goal)`.

Other properties:

`deprecated(G)`

– *The following properties hold globally:*

`deprecated(G)` is side-effect `free`. (`basic_props:sideff/2`)

srcloc/4: PROPERTY

This annotation (pseudo-property) is used to mark that the element (normally assertion) in which it appears was originally between lines `LB`-`LE` of source file `Src`.

Usage: `srcloc(Goal,Src,LB,LE)`

– *The following properties should hold globally:*

`srcloc(Goal,Src,LB,LE)` is side-effect `free`. (`basic_props:sideff/2`)

Meta-predicate with arguments: `srcloc(?,?,?,?)`.

example/1: PROPERTY

This pseudo-property can be used as a qualifier for test assertions. Specifies that the test assertion in which it appears is actually an example and should be included as such in the manual by the auto-documeter.

Usage: `example(G)`

EXAMPLE.

Meta-predicate with arguments: `example(goal)`.

Other properties:

`example(G)`

– *The following properties hold globally:*

`example(G)` is side-effect `free`. (`basic_props:sideff/2`)

not_further_inst/2: PROPERTY

Usage: `not_further_inst(G,V)`

`V` is not further instantiated.

Meta-predicate with arguments: `not_further_inst(goal,?)`.

Other properties:

`not_further_inst(G,V)`

– *The following properties hold globally:*

`not_further_inst(G,V)` is side-effect `free`. (`basic_props:sideff/2`)

`not_further_inst(G,V)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

sideff/2: PROPERTY

`sideff(G,X)`

Declares that `G` is side-effect free (if its execution has no observable result other than its success, its failure, or its abortion), soft (if its execution may have other observable results which, however, do not affect subsequent execution, e.g., input/output), or hard (e.g., assert/retract).

Usage: `sideff(G,X)`

`G` is side-effect `X`.

- *If the following properties should hold at call time:*

  `G` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `X` is an element of `[free,soft,hard]`. (`basic_props:member/2`)

Meta-predicate with arguments: `sideff(goal,?)`.

Other properties:

`sideff(G,X)`

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  `sideff(G,X)` is side-effect `free`. (`basic_props:sideff/2`)

  `sideff(G,X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

regtype/1: PROPERTY

Usage: `regtype(G)`

Defines a regular type.

Meta-predicate with arguments: `regtype(goal)`.

Other properties:

`regtype(G)`

- *The following properties hold globally:*

  `regtype(G)` is side-effect `free`. (`basic_props:sideff/2`)

native/1: PROPERTY

Usage: `native(Pred)`

This predicate is understood natively by CiaoPP.

Meta-predicate with arguments: `native(goal)`.

Other properties:

`native(P)`

- *The following properties hold globally:*

  `native(P)` is side-effect `free`. (`basic_props:sideff/2`)

native/2: PROPERTY

Usage: `native(Pred,Key)`

This predicate is understood natively by CiaoPP as `Key`.

Meta-predicate with arguments: `native(goal,?)`.

Other properties:

`native(P,K)`

– *The following properties hold globally:*
  `native(P,K)` is side-effect `free`. (`basic_props:sideff/2`)

rtcheck/1: PROPERTY

Usage: `rtcheck(G)`

Equivalent to rtcheck(G, complete).

– *If the following properties should hold at call time:*
  `G` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `rtcheck(goal)`.

Other properties:

`rtcheck(G)`

– *The following properties hold globally:*
  `rtcheck(G)` is side-effect `free`. (`basic_props:sideff/2`)

rtcheck/2: PROPERTY

Usage: `rtcheck(G,Status)`

The runtime check of this property is `Status`.

– *If the following properties should hold at call time:*
  `G` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `Status` is the status of the runtime-check implementation for a given property. Valid values are:
  - complete: The implementation of the run-time check for this property is complete. i.e., an error is reported always if the property is violated. Default.
  - incomplete: The current run-time check is incomplete, i.e., it is possible that no error is reported even if the property is violated.
  - unimplemented: No run-time checker has been implemented (yet) for the property.
  - unknown: It has not been determined yet whether the current implementation of the run-time checker is complete or not.
  - impossible: The property cannot be checked at run time (for theoretical or practical reasons).

  (`basic_props:rtc_status/1`)

Meta-predicate with arguments: `rtcheck(goal,?)`.

Other properties:

`rtcheck(G,Status)`

– *The following properties hold globally:*
  `rtcheck(G,Status)` is side-effect `free`. (`basic_props:sideff/2`)

no_rtcheck/1: PROPERTY

This comp pseudo-property is used to declare that the assertion in which it appears should not be checked at run-time. Equivalent to `rtcheck(G, S)` with `S` unimplemented, impossible, etc.

Usage: `no_rtcheck(G)`

`G` is not checked during run-time checking.

- *If the following properties should hold at call time:*

  `G` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `no_rtcheck(goal)`.

Other properties:

`no_rtcheck(G)`

- *The following properties hold globally:*

  `no_rtcheck(G)` is side-effect `free`. (`basic_props:sideff/2`)

eval/1: PROPERTY

Usage: `eval(Goal)`

`Goal` is evaluable at compile-time.

Meta-predicate with arguments: `eval(goal)`.

equiv/2: PROPERTY

Usage: `equiv(Goal1,Goal2)`

`Goal1` is equivalent to `Goal2`.

Meta-predicate with arguments: `equiv(goal,goal)`.

bind_ins/1: PROPERTY

Usage: `bind_ins(Goal)`

`Goal` is binding insensitive.

Meta-predicate with arguments: `bind_ins(goal)`.

error_free/1: PROPERTY

Usage: `error_free(Goal)`

`Goal` is error free.

Meta-predicate with arguments: `error_free(goal)`.

memo/1: PROPERTY

Usage: `memo(Goal)`

`Goal` should be memoized (not unfolded).

Meta-predicate with arguments: `memo(goal)`.

filter/2: PROPERTY

Usage: `filter(Vars,Goal)`

`Vars` should be filtered during global control).

flag_values/1: REGTYPE

Usage: `flag_values(X)`

Define the valid flag values

pe_type/1: PROPERTY

Usage: `pe_type(Goal)`

`Goal` will be filtered in partial evaluation time according to the PE types defined in the assertion.

Meta-predicate with arguments: `pe_type(goal)`.

check/1: PREDICATE

No further documentation available for this predicate.

trust/1: PREDICATE

No further documentation available for this predicate.

true/1: PREDICATE

No further documentation available for this predicate.

false/1: PREDICATE

No further documentation available for this predicate.

40.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `native_props`, `terms_check`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `nativeprops`.

41 Properties which are native to analyzers

Author(s): Francisco Bueno, Manuel Hermenegildo, Pedro López, Edison Mera, Amadeo Casas, Jose F. Morales.

This library contains a set of properties which are natively understood by the different program analyzers of `ciaopp`. They are used by `ciaopp` on output and they can also be used as properties in assertions.

Some of the properties can also be used as runtime-checks. See `native_props_rtc` for the runtime-check implementation of such properties.

42 Properties related to sharing/aliasing, groundness

These properties are part of the `native_props` library.

42.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assertions/native_props))`
  or also as a package `:- use_package(nativeprops)`.
  Note the slightly different names of the library and the package.
- **Exports:**
  - *Properties:*
    `mshare/1`, `mshare/2`, `indep/2`, `indep/1`, `covered/2`, `linear/1`, `ivar/1`, `nonground/1`, `clique/1`, `clique_1/1`.

42.2 Documentation on exports

mshare/1: PROPERTY

`mshare(X)`

`X` contains all *sharing sets* [JL88, MH89] which specify the possible variable occurrences in the terms to which the variables involved in the clause may be bound. Sharing sets are a compact way of representing groundness of variables and dependencies between variables. This representation is however generally difficult to read for humans. For this reason, this information is often translated to `ground/1`, `indep/1` and `indep/2` properties, which are easier to read.

Usage: `mshare(X)`

The sharing pattern for the variables in the clause is `X`.

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP as `sharing(X)`. (`basic_props:native/2`)
  `mshare(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

mshare/2: PROPERTY

Usage: `mshare(Xs,Xss)`

The sharing pattern for the variables `Xs` is `Xss`.

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP as `sharing(Xs,Xss)`. (`basic_props:native/2`)
  `mshare(Xs,Xss)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

indep/2: PROPERTY

Usage: `indep(X,Y)`

`X` and `Y` do not have variables in common.

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP as `indep([[X,Y]])`. (`basic_props:native/2`)

indep/1: PROPERTY

Usage: `indep(X)`

The variables in the the pairs in `X` are pairwise independent.

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP as `indep(X)`. (`basic_props:native/2`)

covered/2: PROPERTY

`covered(X,Y)`

All variables occuring in `X` occur also in `Y`. Used by the non-strict independence-based annotators.

Usage: `covered(X,Y)`

`X` is covered by `Y`.

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

linear/1: PROPERTY

`linear(X)`

`X` is bound to a term which is linear, i.e., if it contains any variables, such variables appear only once in the term. For example, `[1,2,3]` and `f(A,B)` are linear terms, while `f(A,A)` is not.

Usage: `linear(X)`

`X` is instantiated to a linear term.

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

ivar/1: PROPERTY

`ivar(X)`

`X` is a free variable independent of the rest of the variables appearing in the head of the predicate.

For a predicate `p(X0, ..., Xn)`, `ivar(X0)` conceptually expands to `(var(X0), indep([[X0,X1], ..., [X0,Xn]]))`.

Usage: `ivar(X)`

`X` is a free independent variable.

nonground/1: PROPERTY

Usage: `nonground(X)`

`X` is not ground.

- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP as `not_ground(X)`. (`basic_props:native/2`)

clique/1: PROPERTY

`clique(X)`

`X` is a set of variables of interest, much the same as a sharing group but `X` represents all the sharing groups in the powerset of those variables. Similar to a sharing group, a clique is often translated to `ground/1`, `indep/1`, and `indep/2` properties.

Usage: `clique(X)`

The clique sharing pattern is `X`.

- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP as `clique(X)`. (`basic_props:native/2`)

  `clique(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

clique_1/1: PROPERTY

`clique_1(X)`

`X` is a set of variables of interest, much the same as a sharing group but `X` represents all the sharing groups in the powerset of those variables but disregarding the singletons. Similar to a sharing group, a clique_1 is often translated to `ground/1`, `indep/1`, and `indep/2` properties.

Usage: `clique_1(X)`

The 1-clique sharing pattern is `X`.

- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP as `clique_1(X)`. (`basic_props:native/2`)

  `clique_1(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

42.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

43 Properties related to determinacy, failure, choice-points

These properties are part of the `native_props` library.

43.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assertions/native_props))`
  or also as a package `:- use_package(nativeprops)`.
  Note the slightly different names of the library and the package.
- **Exports:**
  - *Properties:*
    `det/1`, `fails/1`, `semidet/1`, `multi/1`, `nondet/1`, `mut_exclusive/1`, `not_mut_exclusive/1`, `possibly_not_mut_exclusive/1`, `covered/1`, `not_covered/1`, `possibly_not_covered/1`, `no_choicepoints/1`, `leaves_choicepoints/1`, `is_det/1`, `non_det/1`, `possibly_nondet/1`, `not_fails/1`, `possibly_fails/1`.

43.2 Documentation on exports

det/1: PROPERTY

`det(X)`

Calls of the form `X` are deterministic, i.e., produce exactly one solution (or do not terminate).

Note that it can still leave choice points after its execution, but when backtracking into these, it can only fail or go into an infinite loop.

These properties are inferred and checked natively by CiaoPP using the domains and techniques of [LGBH05, LGBH10, DLGH97, BLGH04].

Usage: `det(X)`

Calls of the form `X` are deterministic.

Meta-predicate with arguments: `det(goal)`.

fails/1: PROPERTY

`fails(X)`

Calls of the form `X` fail.

Usage: `fails(X)`

Calls of the form `X` fail.

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate with arguments: `fails(goal)`.

semidet/1: PROPERTY

`semidet(X)`

Calls of the form `X` are semi-deterministic, i.e., produce at most one solution (or do not terminate).

Note that it can still leave choice points after its execution, but when backtracking into these, it can only fail or go into an infinite loop.

These properties are inferred and checked natively by CiaoPP using the domains and techniques of [LGBH05, LGBH10, DLGH97, BLGH04].

Usage: `semidet(X)`

Calls of the form `X` are semi-deterministic.

Meta-predicate with arguments: `semidet(goal)`.

multi/1: PROPERTY

`multi(X)`

Calls of the form `X` are multi-deterministic, i.e., they produce one or more solutions and do not fail.

Usage: `multi(X)`

Calls of the form `X` are multi-deterministic.

Meta-predicate with arguments: `multi(goal)`.

nondet/1: PROPERTY

`nondet(X)`

Nothing is ensured about failure and determinacy of calls to `X`. This is the default when no information is given for a predicate, so this property does not need to be stated explicitly.

Usage: `nondet(X)`

Calls of the form `X` are non-deterministic.

- *The following properties should hold globally:*

  `nondet(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `nondet(goal)`.

mut_exclusive/1: PROPERTY

`mut_exclusive(X)`

For any call of the form `X` there is at most one clause whose test (guard) succeeds, i.e., clause tests are pairwise exclusive. Note that determinacy is the transitive closure (to all called predicates) of this property. This property is inferred and checked natively by CiaoPP using the domains and techniques of [LGBH05, LGBH10].

Usage: `mut_exclusive(X)`

For any call of the form `X` at most one clause test succeeds.

- *The following properties should hold globally:*

  The runtime check of this property is `unimplemented`. (`basic_props:rtcheck/2`)

Meta-predicate with arguments: `mut_exclusive(goal)`.

not_mut_exclusive/1: PROPERTY

`not_mut_exclusive(X)`

For calls of the form `X` more than one clause test may succeed. I.e., clause tests are not disjoint for some call.

Usage: `not_mut_exclusive(X)`

For some calls of the form `X` more than one clause test may succeed.

– *The following properties should hold globally:*

  The runtime check of this property is `unimplemented`. (`basic_props:rtcheck/2`)

Meta-predicate with arguments: `not_mut_exclusive(goal)`.

possibly_not_mut_exclusive/1: PROPERTY

`possibly_not_mut_exclusive(X)`

Mutual exclusion of the clause tests for calls of the form `X` cannot be ensured. This is the default when no information is given for a predicate, so this property does not need to be stated explicitly.

Usage: `possibly_not_mut_exclusive(X)`

Mutual exclusion is not ensured for calls of the form `X`.

– *The following properties should hold globally:*

  `possibly_not_mut_exclusive(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `possibly_not_mut_exclusive(goal)`.

covered/1: PROPERTY

`covered(X)`

For any call of the form `X` there is at least one clause whose test (guard) succeeds (i.e., all the calls of the form `X` are covered). Note that nonfailure is the transitive closure (to all called predicates) of this property. [DLGH97, BLGH04].

Usage: `covered(X)`

All the calls of the form `X` are covered.

– *The following properties should hold globally:*

  The runtime check of this property is `unimplemented`. (`basic_props:rtcheck/2`)

Meta-predicate with arguments: `covered(goal)`.

not_covered/1: PROPERTY

`not_covered(X)`

There is some call of the form `X` for which there is no clause whose test succeeds [DLGH97].

Usage: `not_covered(X)`

Not all of the calls of the form `X` are covered.

– *The following properties should hold globally:*

  The runtime check of this property is `unimplemented`. (`basic_props:rtcheck/2`)

Meta-predicate with arguments: `not_covered(goal)`.

possibly_not_covered/1: PROPERTY

`possibly_not_covered(X)`

Covering is not ensured for any call of the form `X`. In other words, nothing can be ensured about covering of such calls.

Usage: `possibly_not_covered(X)`

Covering is not ensured for calls of the form `X`.

- *The following properties should hold globally:*
  `possibly_not_covered(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `possibly_not_covered(goal)`.

no_choicepoints/1: PROPERTY

Usage: `no_choicepoints(X)`

A call to `X` does not leave new choicepoints.

Meta-predicate with arguments: `no_choicepoints(goal)`.

leaves_choicepoints/1: PROPERTY

Usage: `leaves_choicepoints(X)`

A call to `X` leaves new choicepoints.

Meta-predicate with arguments: `leaves_choicepoints(goal)`.

is_det/1: PROPERTY

`is_det(X)`

All calls of the form `X` are deterministic, i.e., produce at most one solution (or do not terminate). In other words, if `X` succeeds, it can only succeed once. It can still leave choice points after its execution, but when backtracking into these, it can only fail or go into an infinite loop. This property is inferred and checked natively by CiaoPP using the domains and techniques of [LGBH05, LGBH10].

Usage: `is_det(X)`

All calls of the form `X` are deterministic.

Meta-predicate with arguments: `is_det(goal)`.

non_det/1: PROPERTY

`non_det(X)`

All calls of the form `X` are non-deterministic, i.e., they always produce more than one solution.

Usage: `non_det(X)`

All calls of the form `X` are non-deterministic.

Meta-predicate with arguments: `non_det(goal)`.

possibly_nondet/1: PROPERTY

`possibly_nondet(X)`

Non-determinism is not ensured for calls of the form `X`. In other words, nothing can be ensured about determinacy of such calls. This is the default when no information is given for a predicate, so this property does not need to be stated explicitly.

Usage: `possibly_nondet(X)`

Non-determinism is not ensured for calls of the form `X`.

- *The following properties should hold globally:*

  `possibly_nondet(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `possibly_nondet(goal)`.

not_fails/1: PROPERTY

`not_fails(X)`

Calls of the form `X` produce at least one solution (succeed), or do not terminate. This property is inferred and checked natively by CiaoPP using the domains and techniques of [DLGH97, BLGH04].

Usage: `not_fails(X)`

All the calls of the form `X` do not fail.

- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate with arguments: `not_fails(goal)`.

possibly_fails/1: PROPERTY

`possibly_fails(X)`

Non-failure is not ensured for any call of the form `X`. In other words, nothing can be ensured about non-failure nor termination of such calls.

Usage: `possibly_fails(X)`

Non-failure is not ensured for calls of the form `X`.

- *The following properties should hold globally:*

  `possibly_fails(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `possibly_fails(goal)`.

43.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

44 Properties related to cardinality and exact solutions

These properties are part of the `native_props` library.

44.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assertions/native_props))`
  or also as a package `:- use_package(nativeprops)`.
  Note the slightly different names of the library and the package.
- **Exports:**
  - *Properties:*
    `cardinality/3`, `num_solutions/2`, `relations/2`, `finite_solutions/1`, `solutions/2`.

44.2 Documentation on exports

cardinality/3: PROPERTY

Usage: `cardinality(Goal,Lower,Upper)`

`Goal` has a number of solutions between `Lower` and `Upper`.

- *The following properties should hold globally:*
  `cardinality(Goal,Lower,Upper)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `cardinality(goal,?,?)`.

num_solutions/2: PROPERTY

Usage 1: `num_solutions(X,N)`

Calls of the form `X` have `N` solutions, i.e., `N` is the cardinality of the solution set of `X`.

- *If the following properties should hold at call time:*
  `X` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `N` is an integer. (`basic_props:int/1`)

Usage 2: `num_solutions(Goal,Check)`

For a call to `Goal`, `Check(X)` succeeds, where `X` is the number of solutions.

- *If the following properties should hold at call time:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `Check` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

relations/2: PROPERTY

`relations(X,N)`

Calls of the form `X` produce `N` solutions, i.e., `N` is the cardinality of the solution set of `X`.

Usage: `relations(X,N)`

Goal `X` produces `N` solutions.

- *If the following properties should hold at call time:*

  `X` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `N` is an integer. (`basic_props:int/1`)

  then the following properties should hold globally:

  The runtime check of this property is `unimplemented`. (`basic_props:rtcheck/2`)

Meta-predicate with arguments: `relations(goal,?)`.

finite_solutions/1: PROPERTY

`finite_solutions(X)`

Calls of the form `X` produce a finite number of solutions [DLGH97].

Usage: `finite_solutions(X)`

All the calls of the form `X` have a finite number of solutions.

- *The following properties should hold globally:*

  `finite_solutions(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `finite_solutions(goal)`.

solutions/2: PROPERTY

Usage: `solutions(Goal,Sols)`

Goal `Goal` produces the solutions listed in `Sols`.

- *If the following properties should hold at call time:*

  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Sols` is a list. (`basic_props:list/1`)

44.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

45 Properties related to exceptions and signals

These properties are part of the `native_props` library.

45.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assertions/native_props))`
  or also as a package `:- use_package(nativeprops)`.
  Note the slightly different names of the library and the package.
- **Exports:**
  - *Properties:*
    `exception/1`, `exception/2`, `possible_exceptions/2`, `no_exception/1`, `no_exception/2`, `signal/1`, `signal/2`, `possible_signals/2`, `no_signal/1`, `no_signal/2`.

45.2 Documentation on exports

exception/1: PROPERTY

Usage: `exception(Goal)`

Calls of the form `Goal` will throw an (unspecified) exception.

Meta-predicate with arguments: `exception(goal)`.

exception/2: PROPERTY

Usage: `exception(Goal,E)`

Calls to `Goal` will throw an exception that unifies with `E`.

Meta-predicate with arguments: `exception(goal,?)`.

possible_exceptions/2: PROPERTY

Usage: `possible_exceptions(Goal,Es)`

Calls of the form `Goal` may throw exceptions, but only the ones that unify with the terms listed in `Es`.

- *If the following properties should hold at call time:*
  `Es` is a list. (`basic_props:list/1`)
  then the following properties should hold globally:
  The runtime check of this property is `unimplemented`. (`basic_props:rtcheck/2`)

Meta-predicate with arguments: `possible_exceptions(goal,?)`.

no_exception/1: PROPERTY

Usage: `no_exception(Goal)`

Calls of the form `Goal` do not throw any exception.

Meta-predicate with arguments: `no_exception(goal)`.

no_exception/2: PROPERTY

Usage: `no_exception(Goal,E)`

Calls of the form `Goal` do not throw any exception that unifies with `E`.

Meta-predicate with arguments: `no_exception(goal,?)`.

signal/1: PROPERTY

Usage: `signal(Goal)`

Calls to `Goal` will send an (unspecified) signal.

Meta-predicate with arguments: `signal(goal)`.

signal/2: PROPERTY

Usage: `signal(Goal,E)`

Calls to `Goal` will send a signal that unifies with `E`.

Meta-predicate with arguments: `signal(goal,?)`.

possible_signals/2: PROPERTY

Usage: `possible_signals(Goal,Es)`

Calls of the form `Goal` may generate signals, but only the ones that unify with the terms listed in `Es`.

- *The following properties should hold globally:*
  The runtime check of this property is `unimplemented`. (`basic_props:rtcheck/2`)

Meta-predicate with arguments: `possible_signals(goal,?)`.

no_signal/1: PROPERTY

Usage: `no_signal(Goal)`

Calls of the form `Goal` do not send any signal.

Meta-predicate with arguments: `no_signal(goal)`.

no_signal/2: PROPERTY

Usage: `no_signal(Goal,E)`

Calls of the form `Goal` do not send any signals that unify with `E`.

Meta-predicate with arguments: `no_signal(goal,?)`.

45.3 Documentation on imports

This module has the following direct dependencies:

– *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

46 Properties related to side effects

These properties are part of the `native_props` library.

46.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assertions/native_props))`
  or also as a package `:- use_package(nativeprops)`.
  Note the slightly different names of the library and the package.
- **Exports:**
  - *Properties:*
    `sideff_pure/1`, `sideff_soft/1`, `sideff_hard/1`.

46.2 Documentation on exports

sideff_pure/1: PROPERTY

Usage: `sideff_pure(X)`

`X` is pure, i.e., has no side-effects.

- *The following properties should hold globally:*
  `sideff_pure(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `sideff_pure(goal)`.

sideff_soft/1: PROPERTY

Usage: `sideff_soft(X)`

`X` has *soft side-effects*, i.e., those not affecting program execution (e.g., input/output).

- *The following properties should hold globally:*
  `sideff_soft(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `sideff_soft(goal)`.

sideff_hard/1: PROPERTY

Usage: `sideff_hard(X)`

`X` has *hard side-effects*, i.e., those that might affect program execution (e.g., assert/retract).

- *The following properties should hold globally:*
  `sideff_hard(X)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `sideff_hard(goal)`.

46.3 Documentation on imports

This module has the following direct dependencies:

– *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

47 Properties related to polyhedral constraints

These properties are part of the `native_props` library.

47.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assertions/native_props))`
  or also as a package `:- use_package(nativeprops)`.
  Note the slightly different names of the library and the package.
- **Exports:**
  - *Properties:*
    `constraint/1`.

47.2 Documentation on exports

constraint/1: PROPERTY

`constraint(C)`

`C` contains a list of linear (in)equalities that relate variables and `int` values. For example, `[A < B + 4]` is a constraint while `[A < BC + 4]` or `[A = 3.4, B >= C]` are not. Used by polyhedra-based analyses.

Usage: `constraint(C)`

`C` is a list of linear equations.

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

47.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

48 Properties related to data sizes, cost, termination

These properties are part of the `native_props` library.

48.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assertions/native_props))`
  or also as a package `:- use_package(nativeprops)`.
  Note the slightly different names of the library and the package.
- **Exports:**
  - *Properties:*
    `resource_id/1`, `size/2`, `size/3`, `size/4`, `size_lb/2`, `size_ub/2`, `size_o/2`, `size_metric/3`, `size_metric/4`, `steps_lb/2`, `steps_ub/2`, `steps/2`, `steps_o/2`, `rsize/2`, `costb/4`, `cost/4`, `terminates/1`.
  - *Regular Types:*
    `approx/1`, `cost_expression/1`, `agg_expression/1`, `numeric_constant/1`, `size_term/1`, `indexvar/1`, `number_lattice/1`, `measure_t/1`.

48.2 Documentation on exports

approx/1: REGTYPE

The types of approximations (bounding) supported by the size and cost analyses (see also `resources_basic.pl`).

- `ub`: for upper bounds.
- `lb`: for lower bounds.
- `exact`: for an exact expression.
- `o`: for the big-O.

Usage: `approx(X)`

`X` represents an approximation.

resource_id/1: PROPERTY

A resource is a numerical property that varies (is used) throughout the execution of a piece of code. Resources can be predefined (i.e., in a library) or user-defined. Examples are computational (resolution) steps, time spent, energy consumed, memory usage, bytes sent over a wire, procedure calls, database operations, files left open, monetary units spent, disk space used, etc. See also `resources_decl`.

Usage: `resource_id(X)`

`X` is the name/identifier of a resource, either an atom or a compound term.

cost_expression/1: REGTYPE

A cost expression is a symbolic function representing the cost of executing a piece of code in terms of the sizes of its input arguments and possibly other parameters. It is a term built from elements of the lattice of real numbers (see `number_lattice`), numerical constants (see `numeric_constant/1`), size metrics (see `size_metric`) and the following functors:

- `- /1`: sign reversal.
- `+ /1`: identity.
- `-- /1`: decrement by one.
- `++ /1`: increment by one.
- `+ /2`: addition.
- `- /2`: subtraction.
- `* /2`: multiplication.
- `/ /2`: division.
- `** /2`: exponentiation.
- `exp/2`: exponentiation (DEPRECATED use `** /2`).
- `log/2`: logarithm given the base.
- `log10/1`: 10-base logarithm.
- `log2/1`: 2-base logarithm.
- `exp/1`: exponential (e to the power of).
- `log/1`: natural logarithm (base e).
- `sqrt/1`: square root.
- `fact/1`: factorial.
- `max/2`: maximum.
- `min/2`: minimum.
- `sum(Index,LowerBound,UpperBound,Exp)`: summation of all `Exp`s obtained by instantiating index variable `Index` from `LowerBound` to `UpperBound`.
- `prod(Index,LowerBound,UpperBound,Exp)`: product of all `Exp`s obtained by instantiating index variable `Index` from `LowerBound` to `UpperBound`.

Usage: `cost_expression(X)`

`X` is a cost expression.

agg_expression/1: REGTYPE

An aggregation expression is an expression appearing in an aggregation function (`sum/4`, `prod/4`). An aggregation expression can be a regular cost expression (see `cost_expression/1`) or it may include index variables as well (see `indexvar/1`).

Usage: `agg_expression(X)`

`X` is an aggregation expression.

numeric_constant/1: REGTYPE

A numeric constant is a fixed, well-defined real number.

Currently supported constants:

- `e`: Euler's number.

Usage: `numeric_constant(X)`

`X` represents a numeric constant.

size_term/1: REGTYPE

A size term is a term in a cost expression (see `cost_expression/1`) representing the size of a term in a given metric (see `measure_t/1`).

Usage: `size_term(X)`

`X` represents the size of a term.

indexvar/1: REGTYPE

A variable used as an index in an aggregation expression.

Usage: `indexvar(X)`

`X` is an index variable.

number_lattice/1: REGTYPE

A lattice containing real numbers, `inf` and `bot`.

Usage: `number_lattice(X)`

`X` is a number, `inf` or `bot`.

size/2: PROPERTY

`size(X,Y)`

The exact size (for any approximation) of term `X` is given by expression `Y`, which may depend on the size of other terms. `size(X,Y)` is equivalent to `size(exact,X,Y)`.

Usage: `size(X,Y)`

`Y` is the size of argument `X`, for any approximation.

- *If the following properties should hold at call time:*

  `X` is any term. (`basic_props:term/1`)

  `Y` is a cost expression. (`user(... /native_props_cost_doc):cost_expression/1`)

  then the following properties should hold globally:

  `size(X,Y)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

size/3: PROPERTY

`size(A,X,Y)`

The size of term `X` for the approximation `A` is given by expression `Y`, which may depend on the size of other terms.

Usage: `size(A,X,Y)`

`Y` is the size of argument `X`, for the approximation `A`.

- *If the following properties should hold at call time:*

  `A` represents an approximation. (`user(... /native_props_cost_doc):approx/1`)

  `X` is any term. (`basic_props:term/1`)

  `Y` is a cost expression. (`user(... /native_props_cost_doc):cost_expression/1`)

  then the following properties should hold globally:

  `size(A,X,Y)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

size/4: PROPERTY

`size(A,M,X,Y)`

The size of term `X`, measured in metric `M`, for the approximation `A` is given by expression `Y`, which may depend on the size of other terms.

Usage: `size(A,M,X,Y)`

`Y` is the size of argument `X` measured in `M`, for the approximation `A`.

– *If the following properties should hold at call time:*

  `A` represents an approximation. (`user(... /native_props_cost_doc):approx/1`)

  `M` is a term size metric. (`user(... /native_props_cost_doc):measure_t/1`)

  `X` is any term. (`basic_props:term/1`)

  `Y` is a cost expression. (`user(... /native_props_cost_doc):cost_expression/1`)

  then the following properties should hold globally:

  `size(A,M,X,Y)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

size_lb/2: PROPERTY

`size_lb(X,Y)`

The minimum size of the terms to which the argument `X` is bound is given by the expression `Y`. Various measures can be used to determine the size of an argument, e.g., list-length, term-size, term-depth, integer-value, etc. [DL93, LGHD96]. See `measure_t/1`.

Usage: `size_lb(X,Y)`

`Y` is a lower bound on the size of argument `X`.

– *If the following properties should hold at call time:*

  `X` is any term. (`basic_props:term/1`)

  `Y` is a cost expression. (`user(... /native_props_cost_doc):cost_expression/1`)

  then the following properties should hold globally:

  `size_lb(X,Y)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

size_ub/2: PROPERTY

`size_ub(X,Y)`

The maximum size of the terms to which the argument `X` is bound is given by the expression `Y`. Various measures can be used to determine the size of an argument, e.g., list-length, term-size, term-depth, integer-value, etc. [DL93, LGHD96]. See `measure_t/1`.

Usage: `size_ub(X,Y)`

`Y` is a upper bound on the size of argument `X`.

– *If the following properties should hold at call time:*

  `X` is any term. (`basic_props:term/1`)

  `Y` is a cost expression. (`user(... /native_props_cost_doc):cost_expression/1`)

  then the following properties should hold globally:

  `size_ub(X,Y)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

size_o/2: PROPERTY

Usage: `size_o(X,Y)`

The size of argument `X` is in the order of expression `Y`.

- *If the following properties should hold at call time:*

  `X` is any term. (`basic_props:term/1`)

  `Y` is a cost expression. (`user(... /native_props_cost_doc):cost_expression/1`)

  then the following properties should hold globally:

  `size_o(X,Y)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

size_metric/3: PROPERTY

Usage: `size_metric(Head,Var,Metric)`

`Metric` is the measure used to determine the size of the terms that `Var` is bound to, for any type of approximation.

- *If the following properties should hold at call time:*

  `Head` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Var` is any term. (`basic_props:term/1`)

  `Metric` is a term size metric. (`user(... /native_props_cost_doc):measure_t/1`)

  then the following properties should hold globally:

  `size_metric(Head,Var,Metric)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `size_metric(goal,?,?)`.

size_metric/4: PROPERTY

Usage: `size_metric(Head,Approx,Var,Metric)`

`Metric` is the measure used to determine the size of the terms that variable `Var` bound to, for the approximation `Approx`.

- *If the following properties should hold at call time:*

  `Head` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Approx` represents an approximation. (`user(... /native_props_cost_doc):approx/1`)

  `Var` is any term. (`basic_props:term/1`)

  `Metric` is a term size metric. (`user(... /native_props_cost_doc):measure_t/1`)

  then the following properties should hold globally:

  `size_metric(Head,Approx,Var,Metric)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `size_metric(goal,?,?,?)`.

measure_t/1: REGTYPE

The types of term size measures currently supported in size and cost analysis (see also in `resources_basic.pl`).

- `int`: The size of the term (which is an integer) is the integer value itself.
- `length`: The size of the term (which is a list) is its length.
- `size`: The size is the overall of the term (number of subterms).
- `depth([_|_])`: The size of the term is its depth.
- `void`: Used to indicate that the size of this argument should be ignored.

Usage: `measure_t(X)`

`X` is a term size metric.

steps_lb/2: PROPERTY

`steps_lb(X,Y)`

The minimum computation (in resolution steps) spent by any call of the form `X` is given by the expression `Y` [DLGHL97, LGHD96]

Usage: `steps_lb(X,Y)`

`Y` is a lower bound on the cost of any call of the form `X`.

– *If the following properties should hold at call time:*

`X` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

`Y` is a cost expression. (`user(... /native_props_cost_doc):cost_expression/1`)

then the following properties should hold globally:

`steps_lb(X,Y)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `steps_lb(goal,?)`.

steps_ub/2: PROPERTY

`steps_ub(X,Y)`

The maximum computation (in resolution steps) spent by any call of the form `X` is given by the expression `Y` [DL93, LGHD96].

Usage: `steps_ub(X,Y)`

`Y` is a upper bound on the cost of any call of the form `X`.

– *If the following properties should hold at call time:*

`X` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

`Y` is a cost expression. (`user(... /native_props_cost_doc):cost_expression/1`)

then the following properties should hold globally:

`steps_ub(X,Y)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `steps_ub(goal,?)`.

steps/2: PROPERTY

`steps(X,Y)`

The computation (in resolution steps) spent by any call of the form `X` is given by the expression `Y`

Usage: `steps(X,Y)`

`Y` is the cost (number of resolution steps) of any call of the form `X`.

- *If the following properties should hold at call time:*

  `X` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Y` is a cost expression. (`user(... /native_props_cost_doc):cost_expression/1`)

  then the following properties should hold globally:

  `steps(X,Y)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `steps(goal,?)`.

steps_o/2: PROPERTY

`steps_o(X,Y)`

The big O expression of the computation (in resolution steps) spent by any call of the form `X` is given by the expression `Y`

Usage: `steps_o(X,Y)`

`Y` is the complexity order of the cost of any call of the form `X`.

- *If the following properties should hold at call time:*

  `X` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Y` is a cost expression. (`user(... /native_props_cost_doc):cost_expression/1`)

  then the following properties should hold globally:

  `steps_o(X,Y)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

Meta-predicate with arguments: `steps_o(goal,?)`.

rsize/2: PROPERTY

Usage: `rsize(Var,SizeDescr)`

`Var` has its size defined by `SizeDescr`.

- *The following properties should hold globally:*

  `rsize(Var,SizeDescr)` is not checked during run-time checking. (`basic_props:no_rtcheck/1`)

costb/4: PROPERTY

`costb(Goal,Resource,Lower,Upper)`

`Lower` (res. `Upper`) is a (safe) lower (res. upper) bound on the cost of the computation of `Goal` in terms of `Resource` units.

Usage: `costb(Goal,Resource,Lower,Upper)`

`Lower` (resp. `Upper`) is a (safe) lower (resp. upper) bound on the cost of the computation of `Goal` expressed in terms of `Resource` units.

- *If the following properties should hold at call time:*

  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Resource` is the name/identifier of a resource, either an atom or a compound term. (`user(... /native_props_cost_doc):resource_id/1`)

Lower is a cost expression. (user(... /native_props_cost_doc):cost_expression/1)

Upper is a cost expression. (user(... /native_props_cost_doc):cost_expression/1)

then the following properties should hold globally:

`costb(Goal,Resource,Lower,Upper)` is not checked during run-time checking. (basic_props:no_rtcheck/1)

Meta-predicate with arguments: `costb(goal,?,?,?)`.

cost/4: PROPERTY

`cost(Goal,Approx,Resource,Expr)`

`Expr` is a (safe) upper or lower bounds, depending on the value of `Approx` (see `approx/1`), of the cost of computation of the goal `Goal` in terms of `Resource` units.

Usage: `cost(Goal,Approx,Resource,Expr)`

`Expr` is a safe upper or lower bounds (depending on `Approx`) of the cost of computing `Goal` in terms of `Resource` units.

- *If the following properties should hold at call time:*

  `Goal` is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

  `Approx` represents an approximation. (user(... /native_props_cost_doc):approx/1)

  `Resource` is the name/identifier of a resource, either an atom or a compound term. (user(... /native_props_cost_doc):resource_id/1)

  `Expr` is a cost expression. (user(... /native_props_cost_doc):cost_expression/1)

  then the following properties should hold globally:

  `cost(Goal,Approx,Resource,Expr)` is not checked during run-time checking. (basic_props:no_rtcheck/1)

Meta-predicate with arguments: `cost(goal,?,?,?)`.

terminates/1: PROPERTY

`terminates(X)`

Calls of the form `X` always terminate.

Usage: `terminates(X)`

All calls of the form `X` terminate.

- *If the following properties should hold at call time:*

  `X` is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

  then the following properties should hold globally:

  `terminates(X)` is not checked during run-time checking. (basic_props:no_rtcheck/1)

Meta-predicate with arguments: `terminates(goal)`.

48.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

49 Classical Prolog modes

Author(s): Manuel Hermenegildo.

This package defines a number of modes which are frequently useful in programs when describing predicates, e.g., via `pred` assertions. They correspond to the modes used in many classical Prolog texts and code as documentation, with some additions. However, these modes are actually syntactic sugar for assertions, that can be checked either statically or dynamically. Note that some of these modes use the same symbol as one of the `basicmodes` and `isomodes` packages (see Chapter 51 [Some basic Prolog modes], page 269 and Chapter 50 [ISO-Prolog modes], page 267) but have in some cases subtly different meaning.

As an example, the following declaration:

```
:- pred is(-,+).
```

Expresses that `is/2` should be called with the second argument bound and it will bind the first argument. Also:

```
:- pred is(-num,+arithexpression).
```

(more precise than the above), expresses that `is/2` should be called with the second argument instantiated to an arithmetic expression and that on success it will bind the first argument to a number. The argument of a mode as above can be any property, including, e.g., regular types.

The first declaration is equivalent to (it is in fact translated to) the assertion:

```
:- pred is(X,Y) : nonvar(Y) => nonvar(X).
```

and the second one to:

```
:- pred is(X,Y) : arithexpression(Y) => num(X).
```

Modes can also be included inside comments in markdown format (see the `doccomments` and `markdown` libraries). For example:

```
%! pred is(-,+):
%  Evaluates the expression in the second argument
%  and binds the first argument with the result.
```

and they are also translated to the corresponding assertions.

49.1 Usage and interface

- **Library usage:**

  `:- use_package(modes).`

  or

  `:- module(...,...,[modes]).`
- **New operators defined:**

  `?/1` [500,fx], `@/1` [500,fx].
- **New modes defined:**

  `+/1`, `-/1`, `--/1`, `?/1`, `@/1`, `in/1`, `++/1`, `out/1`, `go/1`, `+/2`, `-/2`, `--/2`, `?/2`, `@/2`, `in/2`, `++/2`, `out/2`, `go/2`.
- **Implicit imports:**
  - *System library modules:*

    `meta_props`.
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `metaprops`, `hiord`.

49.2 Documentation on new modes

+/1: MODE

Argument should be bound (nonvar) when the predicate is called. For example:

```
:- pred + > +.
```

expresses that both arguments of `>/2` should be bound when the predicate is called.

Usage: `+A`

- *The following properties are added at call time:*

  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

-/1: MODE

The argument is an output argument. It may be bound or not at call time. It will be bound (nonvar) if the predicate succeeds.

Usage: `-A`

- *The following properties are added upon exit:*

  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

--/1: MODE

The argument should be a free variable (i.e., unbound) when the predicate is called.

Usage: `--A`

- *The following properties are added at call time:*

  `A` is a free variable. (`term_typing:var/1`)

?/1: MODE

No information is given on this argument.

@/1: MODE

The argument will not be further instantiated, i.e., will not be more instantiated than when the predicate is called.

Usage: `@(A)`

- *The following properties are added globally:*
  `A` is not further instantiated. (`basic_props:not_further_inst/2`)

in/1: MODE

The argument is ground at call time.

Usage: `in(A)`

- *The following properties are added at call time:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

++/1: MODE

Same as `in`: the argument is ground at call time.

Usage: `++A`

- *The following properties are added at call time:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

out/1: MODE

The argument is a variable when the predicate is called and will be ground if the predicate succeeds.

Usage: `out(A)`

- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

go/1: MODE

The argument is ground by the predicate, if it succeeds.

Usage: `go(A)`

- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

+/2: MODE

Argument should be instantiated when the predicate is called to a term that has the indicated type or property. For example:

```
:- pred + > +.
```

expresses that both arguments of `>/2` should be bound to arithmetic expressions (have the `arithexpression` property) when the predicate is called.

Usage: `A+P`

- *The following properties are added at call time:*
  `A` has property `P`. (`meta_props:call/2`)

-/2: MODE

The argument is an output argument. It may be bound or not at call time. It will be instantiated to a term that has the indicated type or property if the predicate succeeds. For example, this assertion:

```
:- pred length(-list,-int).
```

expresses that `length/2` can be called in any mode, but on output the second argument will be instantiated to a number and the first one will be instantiated to a lit. Note that this does not mean that the list will be ground, but rather that it will be a complete list but whose elements can be any term, including variables.

Usage: `A-P`

- *The following properties are added upon exit:*
  `A` has property `P`. (`meta_props:call/2`)

--/2: MODE

The argument should be a free variable (i.e., unbound) when the predicate is called and will be bound to a term that has the indicated type or property if the predicate succeeds.

Usage: `--(A,P)`

- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)
- *The following properties are added upon exit:*
  `A` has property `P`. (`meta_props:call/2`)

?/2: MODE

The argument can be a variable or, if it is intantiated, it is to a term that is compatible with the indicated type or property.

Usage: `?(A,P)`

- *Call and exit are compatible with:*
  `A` has property `P`. (`meta_props:call/2`)

@/2: MODE

The argument will not be further instantiated, i.e., will not be more instantiated than when the predicate is called, and the term is compatible with the indicated type or property.

Usage: `@(A,P)`

- *Call and exit are compatible with:*
  A has property P. (meta_props:call/2)
- *The following properties are added globally:*
  A is not further instantiated. (basic_props:not_further_inst/2)

in/2: MODE

The argument is ground at call time and is compatible with the indicated type or property.

Usage: `in(A,P)`

- *The following properties are added at call time:*
  A is currently ground (it contains no variables). (term_typing:ground/1)
  A has property P. (meta_props:call/2)
- *The following properties are added upon exit:*
  A is currently ground (it contains no variables). (term_typing:ground/1)

++/2: MODE

Same as `in`: the argument is ground at call time and is compatible with the indicated type or property.

Usage: `++(A,P)`

- *The following properties are added at call time:*
  A is currently ground (it contains no variables). (term_typing:ground/1)
  A has property P. (meta_props:call/2)
- *The following properties are added upon exit:*
  A is currently ground (it contains no variables). (term_typing:ground/1)

out/2: MODE

The argument is a variable when the predicate is called and will be be bound to a ground term that is compatible with the indicated type or property, if the predicate succeeds.

Usage: `out(A,P)`

- *The following properties are added at call time:*
  A is a free variable. (term_typing:var/1)
- *The following properties are added upon exit:*
  A is currently ground (it contains no variables). (term_typing:ground/1)
  A has property P. (meta_props:call/2)

go/2: MODE

The argument is ground by the predicate to a ground term that is compatible with the indicated type or property, if the predicate succeeds.

Usage: `go(A,P)`

- *The following properties are added upon exit:*
  A is currently ground (it contains no variables). (term_typing:ground/1)
  A has property P. (meta_props:call/2)

50 ISO-Prolog modes

Author(s): Daniel Cabeza, Manuel Hermenegildo.

This file defines the “ modes” used in the documentation of the ISO-Prolog standard. See also Chapter 49 [Classical Prolog modes], page 261 for an alternative set of modes.

50.1 Usage and interface

- **Library usage:**

  `:- use_package(isomodes).`

  or

  `:- module(...,...,[isomodes]).`
- **New operators defined:**

  `?/1` [200,fy], `@/1` [200,fy].
- **New modes defined:**

  `+/1`, `-/1`, `?/1`, `@/1`, `+/2`, `-/2`, `?/2`, `@/2`.
- **Implicit imports:**
  - *System library modules:*

    `meta_props`.
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `metaprops`, `hiord`.

50.2 Documentation on new modes

+/1: MODE

Usage: `+A`

- *The following properties are added at call time:*

  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

-/1: MODE

Usage: `-A`

- *The following properties are added at call time:*

  `A` is a free variable. (`term_typing:var/1`)

?/1: MODE

Unspecified argument.

@/1: MODE

Usage: `@(A)`

- *The following properties are added globally:*

  `A` is not further instantiated. (`basic_props:not_further_inst/2`)

+/2: MODE

Usage: `A+X`

- *The following properties are added at call time:*
  `A` has property `X`. (`meta_props:call/2`)

-/2: MODE

Usage: `A-X`

- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)
- *The following properties are added upon exit:*
  `A` has property `X`. (`meta_props:call/2`)

?/2: MODE

Usage: `?(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added upon exit:*
  `A` has property `X`. (`meta_props:call/2`)

@/2: MODE

Usage: `@(A,X)`

- *The following properties are added at call time:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added upon exit:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added globally:*
  `A` is not further instantiated. (`basic_props:not_further_inst/2`)

51 Some basic Prolog modes

Author(s): Manuel Hermenegildo.

This file defines a number of very simple " modes" which are frequently useful in programs. These correspond to some of the modes used in classical Prolog texts with simple addtions. Note that some of these modes use the same symbol as one of the `modes` and `isomodes` packages (see Chapter 49 [Classical Prolog modes], page 261 and Chapter 50 [ISO-Prolog modes], page 267) but have in some cases subtly different meaning.

51.1 Usage and interface

- **Library usage:**
  `:- use_package(basicmodes).`
  or
  `:- module(...,...,[basicmodes]).`
- **New operators defined:**
  `?/1` [500,fx], `@/1` [500,fx].
- **New modes defined:**
  `+/1`, `-/1`, `?/1`, `@/1`, `in/1`, `out/1`, `go/1`, `+/2`, `-/2`, `?/2`, `@/2`, `in/2`, `out/2`, `go/2`.
- **Implicit imports:**
  - *System library modules:*
    `meta_props`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `metaprops`, `hiord`.

51.2 Documentation on new modes

+/1: MODE

Input value in argument.

Usage: `+A`

- *The following properties are added at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

-/1: MODE

No input value in argument.

Usage: `-A`

- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)

?/1: MODE

Unspecified argument.

@/1: MODE

No output value in argument.

Usage: `@(A)`

- *The following properties are added globally:*
  `A` is not further instantiated. (`basic_props:not_further_inst/2`)

in/1: MODE

Input argument.

Usage: `in(A)`

- *The following properties are added at call time:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

out/1: MODE

Output argument.

Usage: `out(A)`

- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

go/1: MODE

Ground output (input/output argument).

Usage: `go(A)`

- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

+/2: MODE

Usage: `A+X`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added at call time:*
  `A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

-/2: MODE

Usage: `A-X`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)

?/2: MODE

Usage: `?(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)

@/2: MODE

Usage: `@(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added globally:*
  `A` is not further instantiated. (`basic_props:not_further_inst/2`)

in/2: MODE

Usage: `in(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added at call time:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

out/2: MODE

Usage: `out(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added at call time:*
  `A` is a free variable. (`term_typing:var/1`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

go/2: MODE

Usage: `go(A,X)`

- *Call and exit are compatible with:*
  `A` has property `X`. (`meta_props:call/2`)
- *The following properties are added upon exit:*
  `A` is currently ground (it contains no variables). (`term_typing:ground/1`)

52 Documentation comments

Author(s): Jose F. Morales, Manuel Hermenegildo.

Stability: **[devel]** This is still a beta version for experimentation. Much functionality is implemented but syntax may change in the future.

This package allows including machine-readable documentation (including assertions) inside code comments. Additionally, a simpler lightweight markup syntax is enabled for (`LPdoc`) documentation.

The overall objective is to allow speeding up the process of documentation for many cases that do not require the full power of the documentation system and assertion language, as well as improving the portability (as documentation comments are simply ignored when not supported by other Prolog systems).

The syntax is partially inspired in the mark up syntax for Doxygen (`http://www.stack.nl/~dimitri/doxygen/markdown.html`), Coqdoc (`http://coq.inria.fr/doc/Reference-Manual018.html#toc97`), and Haddock (`http://www.haskell.org/haddock/doc/html/ch03s08.html`).

52.1 Documentation comments as terms

This package enables grammar extensions that allow some special operators, which annotate the source code with documentation, are then translated as *documentation assertions*.

The following pieces of text are understood as both prefix or postfix operators:

`%!` *Comment* `(or)` `/*!` *Comment* `*/`
`%` *...*

which is used to write arbitrary chunks of documentation (usually referring to the code after them).

`%<` *Comment* `(or)` `/*<` *Comment* `*/`
`%` *...*

which is used to write chunks of documentation (usually referring to the code before them).

Comments appear in the abstract syntax tree of the parsed programs as special terms. The `doccomments` package extracts them from the program to generate the documentation.

Note that reading comments symbolically requires cooperation with the internal parsing routines. For more details, see the `doccomments` Prolog flag and its use in the `read` and `tokenize` modules.

As `# "..."` comments in LPdoc, this approach continues the *documentation in the AST*. Other systems take a similar approach (for example, see Scribble (`http://docs.racket-lang.org/scribble/text.html`)). A simpler approach could just parse documentation in one pass and generate clean code. It is not clear which one is better in the long term.

52.2 Relation with comment assertions

This package allows using an alternative syntax for machine-readable comments. Essentially, most comments of the form:

`:- doc(`*CommentType*`,`*Body*`).`

can be written as:

%! @*CommentType* *Body*

Body can expand over several lines but each must have a `%` in the first column. For example, the following:

```
%! @title  A nice module
%
%  @author Pro Grammer
%
%  @module This is a very nice module indeed.
%          It can be used for several purposes.
%
%  @hide   internal/3
```

is equivalent to:

```
:- doc(title, "A nice module").
:- doc(author,"Pro Grammer").
:- doc(module,"This is a very nice module indeed.
            It can be used for several purposes.").
:- doc(hide,internal/3).
```

See files distributed at `markdown/examples/` for more examples.

52.3 Usage and interface

- **Library usage:**

  `:- use_package(doccomments).`

  or

  `:- module(...,...,[doccomments]).`
- **Implicit imports:**
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

53 PART IV - Language extensions

Author(s): The Ciao Development Team.

The libraries documented in this part extend and modify the Ciao language in several different ways. The extensions include:

- restricted languages (pure Prolog programming);
- constraint logic programming;
- multi-paradigm features (higher-order, functional syntax, feature terms or *records*, global variables, mutable terms, etc.);
- alternative search and evaluation strategies (delaying predicate execution, `undo` goals, breadth-first execution, iterative deepening-based execution);
- concurrency and parallel execution (active modules, parallel programming (e.g., &-Prolog style)).

54 No-prelude

Author(s): The Ciao Development Team.

This special package disables the implicit inclusion of the `prelude` in a Ciao module/program. When using this package most of the *engine modules* defined in the basic language are not included. It selects a minimal *kernel* language with very tight control on the language features.

54.1 Usage and interface

- **Library usage:**

  `:- use_package(noprelude).`

  or

  `:- module(...,...,[noprelude]).`
- **Implicit imports:**
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

55 Pure Prolog package

Author(s): The Ciao Development Team.

This library package allows the use of *pure Prolog* in a Ciao module/program. When using this package the default *prelude* is not included and many of the *engine modules* are not imported.

55.1 Usage and interface

- **Library usage:**
  `:- use_package(pure).`
  or
  `:- module(...,...,[pure]).`
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

56 Higher-order support

Author(s): Daniel Cabeza.

This module is a wrapper for the implementation-defined predicate `call/1`, and it implements the `call/2` predicate.

56.1 Usage and interface

- **Library usage:**
  `:- use_module(engine(hiord_rt)).`
- **Exports:**
  - *Predicates:*
    `call/1`, `call/2`, `SYSCALL/1`, `$nodebug_call/1`, `$meta_call/1`, `this_module/1`.

56.2 Documentation on exports

call/1: PREDICATE

`call(G)`

Executes goal `G`, restricting the scope of the cuts to the execution of `G`. Equivalent to writing a variable `G` in a goal position.

Usage: ⟨• ISO •⟩

- *The following properties should hold at call time:*
  `G` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate with arguments: `call(goal)`.

call/2: PREDICATE

`call(Pred,Arg1)`

There exists a set of builtin predicates of the form `call/N` with `N > 1` which execute predicate `Pred` given arguments `Arg1` ... `ArgX`. If `Pred` has already arguments `Arg1` is added to the start, the rest to the end. This predicate, when `Pred` is a variable, can be written using the special Ciao syntax `Pred(Arg1,...,ArgX)`.

Usage:

- *Call and exit should be compatible with:*
  `Arg1` is any term. (`basic_props:term/1`)
- *The following properties should hold at call time:*
  `Pred` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*
  `Arg1` is any term. (`basic_props:term/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

SYSCALL/1: PREDICATE

Usage:

– *The following properties should hold at call time:*

`Arg1` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

$nodebug_call/1: PREDICATE

Usage:

– *The following properties should hold at call time:*

`Arg1` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `$nodebug_call(goal)`.

$meta_call/1: PREDICATE

Usage: `$meta_call(A)`

– *The following properties should hold at call time:*

`A` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`A` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP as `call(A)`. (`basic_props:native/2`)

this_module/1: PREDICATE

Usage: `this_module(Module)`

`Module` is the internal module identifier for current module.

– *The following properties hold upon exit:*

`Module` is an internal module identifier (`basic_props:internal_module_id/1`)

Meta-predicate with arguments: `this_module(addmodule(?))`.

56.3 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`.

57 Traits

Author(s): Jose F. Morales.

Stability: [devel] Currently the subject of active development and research. Syntax may change without warning or deprecation period.

This package extends the Ciao module system with traits. This is a lightweight translation with no overhead w.r.t. traditional use of `multifile` declarations for *hook* predicates.

A trait is defined in Ciao as a collection of predicates that can be implemented for any functor. Functors can implement multiple traits.

This translation delegates on the underlying module system as much as possible, e.g., for dealing with undefined predicates. See ⟨undefined⟩ [Example code and translation], page ⟨undefined⟩ for a complete example that shows syntax, some special cases, and the translation to plain clauses.

Some important notes on the translation:

- internal argument order ensures that first-argument indexing is preserved
- functors data is passed as an extra argument to implementation clauses as follows:
  - constants add no extra arguments
  - unary functors `f(Datum)` are passed as `Datum`
  - any other functor is passed unaltered

57.1 Usage and interface

- **Library usage:**
  `:- use_package(traits).`
  or
  `:- module(...,...,[traits]).`
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

57.2 Other information

57.3 Example code and translation

Example code:

```
:- module(trait_test, [], [traits, assertions]).

:- trait(gadget, [
    p1/0,
    p2/0,
    q/1,
    r/2
]).
```

```
:- impl(gadget, datum0).

(datum0 as gadget).p1.

(datum0 as gadget).p2 :- fail.

(datum0 as gadget).q(X) :- X = r1.

(datum0 as gadget).r(X, Y) :- X = r1, Y = r2.

:- impl(gadget, datum1/1).

(datum1(_) as gadget).p1.

(datum1(_) as gadget).p2 :- fail.

(datum1(E1) as gadget).q(X) :- X = r1(E1).

(datum1(E1) as gadget).r(X, Y) :- X = r1(E1), Y = r2.

:- impl(gadget, datum2/2).

(datum2(_,_) as gadget).p1.

(datum2(_,_) as gadget).p2 :- fail.

(datum2(E1,_) as gadget).q(X) :- X = r1(E1).

(datum2(E1,E2) as gadget).r(X, Y) :- X = r1(E1), Y = r2(E2).

% (tests)

:- use_module(library(aggregates), [findall/3]).

:- test trait_test(X) =>
     X = [[true,false,r1,    [r1,    r2]],
      [true,false,r1(e1),[r1(e1),r2]],
      [true,false,r1(e1),[r1(e1),r2(e2)]]].

:- export(trait_test/1).
trait_test([Y1,Y2,Y3]) :-
    row(datum0,Y1),
    row(datum1(e1),Y2),
    row(datum2(e1,e2),Y3).

row(A,Xs) :- findall(X,col(A,X),Xs).

col(A,X) :- ( (A as gadget).p1 -> X = true ; X = false ).
col(A,X) :- ( (A as gadget).p2 -> X = true ; X = false ).
col(A,X) :- (A as gadget).q(X).
```

```
col(A,[X,Y]) :- (A as gadget).r(X, Y).
```

which should be equivalent to:

```
:- module(trait_orig, [], [traits, assertions]).

% :- trait gadget { ... }.
:- discontiguous 'gadget.p1'/1.
:- multifile 'gadget.p1'/1.
:- discontiguous 'gadget.p2'/1.
:- multifile 'gadget.p2'/1.
:- discontiguous 'gadget.q'/2.
:- multifile 'gadget.q'/2.
:- discontiguous 'gadget.r'/3.
:- multifile 'gadget.r'/3.

% :- impl gadget for datum0.
'gadget.p1'(datum0) :- '<datum0 as gadget>.p1'.
'gadget.p2'(datum0) :- '<datum0 as gadget>.p2'.
'gadget.q'(datum0, X) :- '<datum0 as gadget>.q'(X).
'gadget.r'(datum0, X, Y) :- '<datum0 as gadget>.r'(X, Y).

'<datum0 as gadget>.p1' :- true.

'<datum0 as gadget>.p2' :- fail.

'<datum0 as gadget>.q'(X) :- X = r1.

'<datum0 as gadget>.r'(X, Y) :- X = r1, Y = r2.

% :- impl gadget for datum1/1.
'gadget.p1'(M) :- M = datum1(D), '<datum1/1 as gadget>.p1'(D).
'gadget.p2'(M) :- M = datum1(D), '<datum1/1 as gadget>.p2'(D).
'gadget.q'(M, X) :- M = datum1(D), '<datum1/1 as gadget>.q'(X, D).
'gadget.r'(M, X, Y) :- M = datum1(D), '<datum1/1 as gadget>.r'(X, D, Y).

'<datum1/1 as gadget>.p1'(_E1) :- true.

'<datum1/1 as gadget>.p2'(_E1) :- fail.

'<datum1/1 as gadget>.q'(X, E1) :- X = r1(E1).

'<datum1/1 as gadget>.r'(X, E1, Y) :- X = r1(E1), Y = r2.

% :- impl gadget for datum2/2.
'gadget.p1'(M) :- M = datum2(_,_), '<datum2/2 as gadget>.p1'(M).
'gadget.p2'(M) :- M = datum2(_,_), '<datum2/2 as gadget>.p2'(M).
'gadget.q'(M, X) :- M = datum2(_,_), '<datum2/2 as gadget>.q'(X, M).
'gadget.r'(M, X, Y) :- M = datum2(_,_), '<datum2/2 as gadget>.r'(X, M, Y).

'<datum2/2 as gadget>.p1'(datum2(_E1, _E2)) :- true.

'<datum2/2 as gadget>.p2'(datum2(_E1, _E2)) :- fail.
```

```
'<datum2/2 as gadget>.q'(X, datum2(E1, _E2)) :- X = r1(E1).

'<datum2/2 as gadget>.r'(X, datum2(E1, E2), Y) :- X = r1(E1), Y = r2(E2).

% (tests)

:- use_module(library(aggregates), [findall/3]).

:- test trait_test(X) =>
     X = [[true,false,r1,    [r1,    r2]],
      [true,false,r1(e1),[r1(e1),r2]],
      [true,false,r1(e1),[r1(e1),r2(e2)]]].

:- export(trait_test/1).
trait_test([Y1,Y2,Y3]) :-
    row(datum0,Y1),
    row(datum1(e1),Y2),
    row(datum2(e1,e2),Y3).

row(A,Xs) :- findall(X,col(A,X),Xs).

col(A,X) :- ( 'gadget.p1'(A) -> X = true ; X = false ).
col(A,X) :- ( 'gadget.p2'(A) -> X = true ; X = false ).
col(A,X) :- 'gadget.q'(A, X).
col(A,[X,Y]) :- 'gadget.r'(A, X, Y).
```

58 Terms with named arguments -records/feature terms

Author(s): Daniel Cabeza, Manuel Hermenegildo, Jose F. Morales.

This library package provides syntax which allows accessing term arguments by name (these terms are sometimes also referred to as *records*, and are also similar to *feature terms* [AKPS92]).

58.1 Usage and interface

- **Library usage:**
  `:- use_package(argnames).`
  or
  `:- module(...,...,[argnames]).`
- **Exports:**
  - *Predicates:*
    `$~/3`.
- **New operators defined:**
  `$/2` [150,xfx], `=>/2` [950,xfx], `argnames/1` [1150,fx].
- **New declarations defined:**
  `argnames/1`.
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

58.2 Documentation on new declarations

argnames/1: DECLARATION

Usage: :- `argnames(ArgNamedPredSpec)`.

An `argnames/1` declaration assigns names to the argument positions of terms (or literal/goals) which use a certain functor/arity. This allows referring to these arguments by their name rather than by their argument position. Sometimes, argument names may be clearer and easier to remember than argument positions, specially for predicates with many arguments. Also, in some cases this may allow adding arguments to certain predicates without having to change the code that uses them. These terms with named arguments are sometimes also referred to as records, and are also similar to feature terms [AKPS92]. For example, in order to write a program for the *zebra* puzzle we might declare:

```
:- use_package([argnames]).
:- argnames house(color, nation, pet, drink, car).
```

which first includes the package and then assigns a name to each of the arguments of any term (or literal/goal) with `house/5` as the main functor.

For convenience the package extends the built-in `data/1` declaration so that names to arguments can be asigned as with the `argnames/1` declaration, as for example:

```
:- data product(id, description, brand, quantity).
```

Once an `argnames/1` is given, it is possible to use the names to refer to the arguments of any term (or literal/goal) which has the same main functor as that of the term which appears in the `argnames/1` declaration. This is done by first writing the functor name, then the infix operator `$`, and then, between curly brackets, zero, one, or more pairs *argument-name*`=>`*argument-value*, separated by commas (i.e., the infix operator `=>` is used between the name and the value). Again, argument names must be atomic. Argument values can be any term. Arguments which are not specified are assumed to have a value of "`_`" (i.e., they are left unconstrained).

Thus, after the declaration for `house/5` in the example above, any occurrence in that code of, for example, `house${nation=>Owns_zebra,pet=>zebra}` is exactly equivalent to `house(_,Owns_zebra,zebra,_,_)`. Also, `house${}` is equivalent to `house(_,_,_,_,_)`. The actual zebra puzzle specification might include a clause such as:

```
zebra(Owns_zebra, Drinks_water, Street) :-
   Street = [house${},house${},house${},house${},house${}],
   member(house${nation=>Owns_zebra,pet=>zebra}, Street),
   member(house${nation=>Drinks_water,drink=>water}, Street),
   member(house${drink=>coffee,color=>green}, Street),
   left_right(house${color=>ivory}, house${color=>green}, Street),
   member(house${car=>porsche,pet=>snails}, Street),
    ...
```

Another syntax supported, useful mainly in declarations to avoid specifying the arity, is `house${/}`, which is equivalent in our example to `house/5` (but for data declarations there is a special syntax as we have seen).

Any number of `argnames/1` declarations can appear in a file, one for each functor whose arguments are to be accessed by name. As with other packages, argument name declarations are *local to the file* in which they appear. The `argnames/1` declarations affect only program text which appears after the declaration. It is easy to make a set of declarations affect several files for example by putting such declarations in a sepatate file which is included by all such files.

An `argnames/1` declaration does not change in any way the internal representation of the associated terms and does not affect run-time efficiency. It is simply syntactic sugar.

Runtime support

It is possible to write pairs with unbound argument names. In that case, runtime information is emitted to resolve the argument name at execution time.

58.3 Documentation on exports

$~/3: PREDICATE

Usage: `$~(Term,Replacement,NewTerm)`

`NewTerm` is as `Term` but with the arguments specified in `Replacement` changed (they need to be in argnames syntax). The predicate is in fact virtual, since it is translated by the package to a pair of unifications. For example, given the declaration `:- argnames house(color, nation, pet, drink, car)`, the goal

```
$~(House, house${car => seat, pet => mouse}, NewHouse)
```

would be compiled to the unifications

```
House = house(C,N,_,D,_), NewHouse = house(C,N,mouse,D,seat).
```

58.4 Other information

Two simple examples of the use of the argnames library package follow.

58.4.1 Using argument names in a toy database

```
:- module(simple_db,_,[argnames,assertions,regtypes]).
:- use_module(library(aggregates)).

:- doc(title,"A simple database application using argument names").

:- data
product( id,    description,    brand,          quantity        ).
%       ----------------------------------------------------------
product(  1,    "Keyboard",     "Logitech",     6               ).
product(  2,    "Mouse",        "Logitech",     5               ).
product(  3,    "Monitor",      "Philips",      3               ).
product(  4,    "Laptop",       "Dell",         4               ).
% (${/} must go after argnames)
:- pred product${/}
   ::    int    * string        * string        * int.

% Compute the stock of products from a given brand.
% Note call to findall is equivalent to: findall(Q,product(_,_,Brand,Q),L).

brand_stock(Brand,Stock) :-
    findall(Q,product${brand=>Brand,quantity=>Q},L),
    sumlist(L,Stock).

sumlist([],0).
sumlist([X|T],S) :-
    sumlist(T,S1),
    S is X + S1.
```

58.4.2 Complete code for the zebra example

```
:- module(_,zebra/3,[argnames]).

/*     There are five consecutive houses, each of a different
color and inhabited by men of different nationalities. They each
own a different pet, have a different favorite drink, and drive a
different car.

1.   The Englishman lives in the red house.
2.   The Spaniard owns the dog.
3.   Coffee is drunk in the green house.
4.   The Ukrainian drinks tea.
5.   The green house is immediately to the right of the ivory
     house.
6.   The Porsche driver owns snails.
```

```
7.   The Masserati is driven by the man who lives in the yellow
     house.
8.   Milk is drunk in the middle house.
9.   The Norwegian lives in the first house on the left.
10.  The man who drives a Saab lives in the house next to the man
     with the fox.
11.  The Masserati is driven by the man in the house next to the
     house where the horse is kept.
12.  The Honda driver drinks orange juice.
13.  The Japanese drives a Jaguar.
14.  The Norwegian lives next to the blue house.

The problem is: Who owns the Zebra?  Who drinks water?
*/

:- argnames house(color, nation, pet, drink, car).

zebra(Owns_zebra, Drinks_water, Street) :-
    Street = [house${},house${},house${},house${},house${}],
    member(house${nation => Owns_zebra, pet => zebra}, Street),
    member(house${nation => Drinks_water, drink => water}, Street),
    member(house${nation => englishman, color => red}, Street),
    member(house${nation => spaniard, pet => dog}, Street),
    member(house${drink => coffee, color => green}, Street),
    member(house${nation => ukrainian, drink => tea}, Street),
    left_right(house${color => ivory}, house${color => green}, Street),
    member(house${car => porsche, pet => snails}, Street),
    member(house${car => masserati, color => yellow}, Street),
    Street = [_, _, house${drink => milk}, _, _],
    Street = [house${nation => norwegian}|_],
    next_to(house${car => saab}, house${pet => fox}, Street),
    next_to(house${car => masserati}, house${pet => horse}, Street),
    member(house${car => honda, drink => orange_juice}, Street),
    member(house${nation => japanese, car => jaguar}, Street),
    next_to(house${nation => norwegian}, house${color => blue}, Street).


member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

left_right(L,R,[L,R|_]).
left_right(L,R,[_|T]) :- left_right(L,R,T).

next_to(X,Y,L) :- left_right(X,Y,L).
next_to(X,Y,L) :- left_right(Y,X,L).
```

59 Functional notation

Author(s): Daniel Cabeza, Amadeo Casas, Manuel Hermenegildo, Jose F. Morales.

This library package allows the use of functional notation in a Ciao module/program. It supports function application, predefined evaluable functors, functional definitions, quoting, and (combined with the `lazy` library) lazy evaluation. The extensions implemented by this library are also composable with higher-order features and can be combined with other Ciao packages such as constraints, assertions, etc.

The package provides *syntactic sugar* for defining and using predicates as if they were functions. However, they can still retain the power of predicates. Any function definition written using this package is in fact defining a predicate, and any predicate can be used as a function.

The predicate associated with a function has the same name and one more argument, meant as the place holder for the "result" of the function. In fact, this argument is just the one that will be syntactically connected to the surrounding goal or function, but it does not necessarily imply any directionality, i.e., it does not necessarily mean that this argument is an output or an input. This argument is by default added to the right, i.e., it is the last argument, but can be changed by using a declaration, as explained below.

59.1 Function applications

Any term preceded by the `~ /1` operator is a function application, as can be seen in the goal `write(~arg(1,T))`, which is strictly equivalent to the sequence `arg(1,T,A), write(A)`. The declaration `fun_return/1` allows using a predicate argument other than the last as the return argument. For example with `:- fun_return functor(~,_,_)` the expression `~functor(f,2)` will be evaluated to the term `f(_,_)`. This definition of the return argument can also be done on the fly in each invocation in the following way: `~functor(~,f,2)`.

Functors can be declared as evaluable by using the declaration `fun_eval/1`. This allows avoiding the need to use the `~` operator. Thus, `:- fun_eval arg/2` allows writing `write(arg(1,T))` instead of `write(~arg(1,T))` as above. This declaration can be combined with the previous one: `:- fun_eval functor(~,_,_)`.

59.2 Predefined evaluable functors

By using the declaration `:- fun_eval arith(true)`, all the functors understood by `is/2` will be also evaluated. This is active from the declaration downwards until a `:- fun_eval arith(false)` declaration or the end of the module is reached. Beware that arithmetic functors are used in some cases for other purposes than arithmetic: e.g. `abolish(p/2)`. But this is not so disturbing as it may appear because this package is not active in declarations, except for the goal-including declarations `initialization/1` and `on_abort/1`. Note that all the declarations introduced by this package, as is customary in Ciao, are local to the module where they are included.

In addition to functors declared with the declaration `fun_eval/1`, the package defines as evaluable the functors used for disjunctive and conditional expressions: `| /2` and `? /2` (defined as operators). A disjunctive expression has the form `(V1|V2)`, and its value when first evaluated is `V1`, and on backtracking `V2`. A conditional expression has the form `(Cond ? V1)`, or more commonly `(Cond ? V1 | V2)`, and its value, if the execution of `Cond` as a goal succeeds, is `V1`, otherwise in the first form it causes backtracking, and on the second form its value is `V2`. Note that due to the operator precedences, these expressions normally need to be surrounded by parenthesis. Also, a nested expression: `(Cond1 ? V1 | Cond2 ? V2 | V3)` is evaluated as `(Cond1 ? V1 | (Cond2 ? V2 | V3))`.

59.3 Functional definitions

A functional definition is composed of one or more functional clauses. A functional clause is written using the binary operator `:= /2`, as in:

```
opposite(red) := green.
```

which is equivalent to `opposite(red,green).` or

```
addlast(X,L) := ~append(L,[X]).
```

which is equivalent to `addlast(X,L,R) :- append(L,[X],R).`

Functional clauses can also have a body, which is executed before the result value is computed. It can serve as a guard for the clause or to provide the equivalent of where-clauses in functional languages:

```
fact(0) := 1.
fact(N) := N * ~fact(--N) :- N > 0.
```

Note that guards can often be defined more compactly using conditional expressions:

```
fact(N) := N = 0 ? 1
     | N > 0 ? N * ~fact(--N).
```

The declaration `:- fun_eval defined(true)` allows to locally define as evaluable functions being defined, so that the `~` operator does not need to be used within a functional definition for the functor being defined. For example, for the `fact` invocations in the previous definitions, which can now be written as, e.g. (we provide the full module definition):

```
:- module(_,_,[fsyntax]).

:- fun_eval arith(true).
:- fun_eval defined(true).

fact(0) := 1.
fact(N) := N * fact(--N) :- N > 0.

%% Or,alternatively:
%
% fact(N) := N=0 ? 1
%          | N>0 ? N * fact(--N).
```

This behaviour is reverted using `:- fun_eval defined(false)`.

The translation of functional clauses has the following properties:

- The translation produces *steadfast* predicates, that is, output arguments are unified after possible cuts.
- Defining recursive predicates in functional style maintains the tail recursion of the original predicate, thus allowing the usual compiler optimizations.

Some implementation details and a discussion of the recent combination of this library (which dates from Ciao version 0.2) with the lazy evaluation library can be found in [CCH06].

59.4 Quoting functors

Functors (either in functional or predicate clauses) can be prevented from being evaluated by using the `^ /1` prefix operator (read as "quote"), as in

```
:- fun_eval arith(true).
pair(A,B) := ^(A-B).
```

Note that this just prevents the evaluation of the principal functor of the enclosed term, not the possible occurrences of other evaluable functors inside.

59.5 Some scoping issues

When using function applications inside the goal arguments of meta-predicates, there is an ambiguity as they could be evaluated either in the scope of the outer execution or the in the scope of the inner execution. The chosen behavior is by default to evaluate function applications in the scope of the outer execution. If they should be evaluated in the inner scope, the goal containing the function application needs to be escaped with the `^^ /1` prefix operator, as in `findall(X, (d(Y), ^^(X = ~f(Y)+1)), L)` (which could also be written as `findall(X, ^^ (d(Y), X = ~f(Y)+1), L)`) and which expands into `findall(X, (d(Y),f(Y,Z),T is Z+1,X=T), L)`. With no escaping the function application is evaluated in the scope of the outer execution, i.e., it expands to `f(Y,Z), T is Z+1, findall(X, (d(Y),X=T), L)`.

59.6 Other functionality

In addition to the basic package `fsyntax`, a package `functional` is also provided, to allow programming with a more functional-flavored style. That package activates the declarations `:- fun_eval arith(true)` and `:- fun_eval defined(true)`, and defines the `. /2` operator for use in lists (but be careful: this period cannot be followed by a whitespace!) and the operator `++ /2` as a function for appending lists. The factorial example above can be written as follows using the `functional` package:

```
:- module(_,_,[functional]).

fact(N) := N=0 ? 1
     | N>0 ? N * fact(--N).
```

Which is equivalent to:

```
:- module(_,_,[fsyntax]).

:- fun_eval arith(true).
:- fun_eval defined(true).

fact(0) := 1.
fact(N) := N * fact(--N) :- N > 0.

%% Or,alternatively:
%
% fact(N) := N=0 ? 1
%          | N>0 ? N * fact(--N).
```

See the end of this chapter for additional examples.

59.7 Combining with higher order

Ciao provides in its standard library the `hiord` package, which supports a form of higher-order untyped logic programming with predicate abstractions [CH99a, Cab04, CHL04]. Predicate abstractions are Ciao's translation to logic programming of the lambda expressions of functional programming: they define unnamed predicates which will be ultimately executed by a higher-order call, unifying its arguments appropriately. A function abstraction is provided as functional syntactic sugar for predicate abstractions:

Predicate abstraction: `''(X,Y) :- p(X,Z), q(Z,Y).`

Function abstraction: `''(X) := ~q(~p(X)).`

and function application is syntactic sugar over predicate application:

Predicate application: `..., P(X,Y), ...` Function application: `..., Y = ~P(X), ...`

The combination of this `hiord` package with the `fsyntax` and `lazy` packages (and, optionally, the type inference and checking provided by CiaoPP [HPBLG05]) basically provide the functionality present in modern functional languages (currying is not *syntactically* implemented, but its results can be obtained by deriving higher-order data from any other higher-order data (see [Cab04]), as well as some of the functionality of full higher-order logic programming.

At this moment, it is necessary to specify the `:- fun_eval hiord(true)` option to enable correct handling of function abstractions.

59.8 Usage and interface

- **Library usage:**
  `:- use_package(fsyntax).`
  or
  `:- module(...,...,[fsyntax]).`
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

59.9 Other information

59.10 Some examples using functional syntax

We now illustrate some of the uses of the package through examples. The following example defines a simple unary function `der(X)` which returns the derivative of a polynomial arithmetic expression:

```
der(x)      := 1.
der(C)      := 0                 :- number(C).
der(A + B)  := der(A) + der(B).
der(C * A)  := C * der(A)        :- number(C).
der(x ** N) := N * x ** ~(N - 1) :- integer(N), N > 0.
```

Note that if we include the directive mentioned before which makes arithmetic functors evaluable then we would have to write the program in the following (clearly, less pleasant and more obfuscated) way:

```
:- fun_eval(arith(true)).
der(x)         := 1.
der(C)         := 0                       :- number(C).
der(^(A + B))  := ^(der(A) + der(B)).
der(^(C * A))  := ^(C * der(A))           :- number(C).
der(^(x ** N)) := ^(N * ^(x ** (N - 1))) :- integer(N), N > 0.
```

Both of the previous code fragments translate to the following code:

```
der(x, 1).
der(C, 0) :-
      number(C).
der(A + B, X + Y) :-
      der(A, X),
      der(B, Y).
der(C * A, C * X) :-
      number(C),
      der(A, X).
der(x ** N, N * x ** N1) :-
      integer(N),
      N > 0,
      N1 is N - 1.
```

Functional notation interacts well with other Ciao language features. For example, it provides compact and familiar notation for regular types and other properties:

```
:- module(_,_,[hiord,functional,assertions,regtypes,sr/bfall]).

:- regtype color/1. color := red | blue | green.

:- regtype slist/1. slist := [] | [ _ | slist].

:- regtype list_of/2. list_of(T) := [] | [~T | list_of(T)].
```

where the functional clauses expand to (note the use of higher-order in the third example):

```
color(red). color(blue). color(green).
list([]).
list([_|T]) :- list(T).
list_of(_, []).
list_of(T, [X|Xs]) :- T(X), list_of(T, Xs).
```

Such types and properties are then admissible in the usual way in assertions, e.g.:

```
:- pred append/3 :: list * list * list.
:- pred color_value/2 :: list(color) * int.
```

The combination of functional syntax and user-defined operators brings significant flexibility, as can be seen in the following definition of a list concatenation (`append`) operator (note that these are the definitions mentioned before which are active by default in the `functional` package):

```
:- op(600, xfy, (.)).
:- op(650, xfy, (++)).
:- fun_eval (++)/2.
[]   ++ L := L.
X.Xs ++ L := X.(Xs ++ L).
```

This definition will be compiled exactly to the standard definition of `append` (and, thus, will be reversible). The functional syntax and user-defined operators allow writing for example `Space`

`= ' ', write("Hello" ++ Space ++ "world!")` instead of the equivalent forms `Space = ' ', write( append("Hello", append(Space, "world!")))` (if `append/2` is defined as evaluable) or `Space = ' ', append(Space, "world!", T1), append("Hello", T1, T2), write(T2)`.

As another example, we can define an array indexing operator for fixed-size, multi-dimensional arrays. Assume that arrays are built using nested structures whose main functor is `a` and whose arities are determined by the specified dimensions, i.e., a two-dimensional array *A* of dimensions *[N,M]* will be represented by the nested structure `a(a(`*A11*`,...,`*A1M*`), a(`*A21*`,..,`*A2M*`), ..., a(`*AN1*`,..., `*ANM*`))`, where *A11*,... *ANM* may be arbitrary terms (we ignore for simplicity arity limitations, solved in any case typically by further nesting with logarithmic access time). The following recursive definition defines the property `fixed_array/2` and also the array access operator `@`:

```
fixed_array([N|Ms],A):-
    functor(A,a,N),
    rows(N,Ms,A).
fixed_array([N],A):-
    functor(A,a,N).

rows(0,_,_).
rows(N,Ms,A) :-
    N > 0,
    arg(N,A,Arg),
    array(Ms,Arg),
    rows(N-1,Ms,A).

:- pred @(Array,Index,Elem) :: array * list(int) * int
   # "@var{Elem} is the @var{Index}-th element of @var{Array}.".

:- op(55, xfx, '@').
:- fun_eval (@)/2.
V@[I]    := ~arg(I,V).       %% Or: V@[] := V.
V@[I|Js] := ~arg(I,V)@Js.
```

This allows writing, e.g., `M = fixed_array([2,2]), M@[2,1] = 3` (which could also be expressed as `fixed_array([2,2])@[2,1] = 3`), where the call to the `fixed_array` property generates an empty *2 x 2* array *M* and `M@[2,1] = 3` puts *3* in *M[2,1]*. This can be done in the top level:

```
?- M = ~fixed_array([2,2]), M@[2,1] = 3.
```

provided the `op` and `function` declarations are loaded into the top level also. Another example of use is: `A3@[N+1,M] = A1@[N-1,M] + A2@[N,M+2]`.

Such functionality can be grouped into a *package* as follows. The package main file (`arrays.pl`) might be:

```
:- package(arrays).
:- include(arrays_ops).

:- use_module(arrays_rt).
```

where file `arrays_ops.pl` may contain:

```
:- use_package(functional).

:- op(150,xfx,[@]).
:- fun_eval '@'/2.
```

```
:- op(500,yfx,<+>).
:- fun_eval '<+>'/2.

:- op(400,yfx,<*>).
:- fun_eval '<*>'/2.
```

The main file is `arrays_rt.pl` which would contain for example (note that it also uses `arrays_ops.pl`, and that is why the contents of `arrays_ops.pl` were not put directly in `arrays.pl`):

```
:- module(arrays_rt,_,[functional,hiord,assertions,regtypes,isomodes]).

:- include(arrays_ops).

:- doc(title,"Some simple array operations with syntactic support").
:- doc(author,"Pro Grammer").

:- doc(module,"This library implements a very simple set of
   operations on arrays. The idea is to illustrate the use of
   functional syntax (operators) by providing syntactic support for
   invoking array operations such as element access, array (vector)
   addition, etc.").

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Regtypes

%% :- doc(doinclude,array/1).
%% :- doc(doinclude,vector/1).
%% :- doc(doinclude,dim/1).

:- regtype array(A) #"@var{A} is a multi-dimensional array.".
% Should obviously be defined in more detail...
array(A) :- struct(A).

:- regtype dim(D) # "@var{D} represents the dimensions of an array.".
dim(D) :- list(int,D).

:- regtype vector(V) # "@var{V} is a one-dimensional fixed-size array.".
vector(V) :- fixed_array([N],V), int(N).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- pred fixed_array(Dim,Array) :: dim * array
# "@var{Array} is an array of fixed dimensions @var{Dim}.".

fixed_array([N|Ms],A):-
    functor(A,a,N),
    rows(N,Ms,A).
fixed_array([N],A):-
    functor(A,a,N).
```

```
rows(0,_Ms,_A).
rows(N,Ms,A):-
    N > 0,
    arg(N,A,Arg),
    fixed_array(Ms,Arg),
    rows(N-1,Ms,A).

:- pred @(Array,Index,Elem):: array * dim * int
# "@var{Elem} is the @var{Index}-th element of @var{Array}.".

V@[I]    := ~arg(I,V).
V@[I|Js] := ~arg(I,V)@Js.

:- pred <+>(V1,V2,V3) :: vector * vector * vector
# "@var{V3} is @var{V1} + @var{V2}.".

V1 <+> V2 := V3 :-
    V1 = ~fixed_array([N]),
    V2 = ~fixed_array([N]),
    V3 = ~fixed_array([N]),
    V3 = ~vecplus_(N,V1,V2).

vecplus_(0,_,_,_).
vecplus_(N,V1,V2,V3) :-
    N > 0,
    V3@[N] = V1@[N] + V2@[N],
    vecplus_(N-1,V1,V2,V3).

:- pred <*>(V1,V2,V3) :: vector * vector * vector
# "@var{V3} is @var{V1} * @var{V2} (inner product).".

V1 <*> V2 := ~vecmul_(N,V1,V2,0) :-
    V1 = ~fixed_array([N]),
    V2 = ~fixed_array([N]).

vecmul_(0,  _,  _, Acc, Acc).
vecmul_(N, V1, V2, Acc, IP) :-
    N > 0,
    vecmul_( N-1, V1, V2, Acc + ( V1@[N] * V2@[N] ), IP).
```

A file using this package would be:

```
:- module(_,_,[]).

:- use_package(library(fsyntax/examples/arrays)).
:- use_module(engine(io_basic)).

main(M) :-
    V1 = a(1,3,4,5),
    V2 = a(5,4,3,1),
    I = 1,
    display(V2@[I+1]),
```

```
      M = V1 <*> ( V2 <+> V1 ).
```

59.11 Examples of combining with higher order

The following `map` and `foldl` definitions (from the `hiordlib` library) illustrate the combination of functional syntax and higher-order logic programming:

```
:- fun_eval map/2.
:- meta_predicate map(_,pred(2),_).
map([], _)     := [].
map([X|Xs], P) := [P(X) | map(Xs, P)].

:- fun_eval foldl/3.
:- meta_predicate foldl(_,_,pred(3),_).
foldl([], Seed, _Op) := Seed.
foldl([X|Xs], Seed, Op) := ~Op(X,~foldl(Xs,Seed,Op)).
```

With this definition:

```
?- L = ~map([1,2,3], ( _(X,Y):- Y = f(X) ) ).

L = [f(1),f(2),f(3)] ?

?- [f(1),f(2),f(3)] = ~map(L, ( _(X,f(X)) :- true ) ).

L = [1,2,3] ?
```

Also, after running:

```
?- ["helloworld", "byeworld"] = map(["hello", "bye"], ++(X)).
```

(where `(++)/2` corresponds to the above definition of `append`) `X` will be bound to `"world"`, which is the only solution to the equation.

And when calling:

```
map(L, ++(X), ["hello.", "bye."]).
```

several values for `L` and `X` are returned through backtracking:

```
L = ["hello","bye"],   X = "." ? ;
L = ["hello.","bye."], X = [] ?
```

(remember to set the flag `write_strings` to on in these examples so that the top level prints strings as strings of characters instead of lists of ASCII codes).

59.12 Some additional examples using functional syntax

A definition of the Fibonacci function, written in functional notation:

```
:- module(_,_,[functional]).

:- use_module(engine(messages_basic), [message/2]).

fib(0) := 0.
fib(1) := 1.
fib(N) := fib(N-1) + fib(N-2) :- integer(N), N > 1.

write_fib(N):-
    message(user, ['The ',N,'. Fibonacci number is: ',~fib(N),'.']).
```

This is the factorial example, written in functional notation and including some assertions:

```
:- module(_,_,[assertions,nativeprops,functional]).

:- pred fact(+int,-int) + is_det.
:- pred fact(-int,+int) + non_det.

fact(N) := N=0 ? 1
     | N>0 ? N * fact(--N).
```

And, the same example written using `clpq` constraints:

```
:- module(_,_,[assertions,nativeprops,fsyntax,clpqf]).

:- fun_eval .=. /1.
:- op(700,fx,[.=.]).
:- fun_eval fact/1.

:- pred fact(+int,-int) + is_det.
:- pred fact(-int,-int) + non_det.

fact( .=. 0) := .=. 1.
fact(N) := .=. N*fact( .=. N-1 ) :- N .>. 0.
```

which allows for example calling it “backwards:”

```
?- 24 = ~fact(X).

X = 4 ?
```

A very simple example using lazy evaluation:

```
:- module(_,_,[functional,lazy]).
:- use_module(library(lazy/lazy_lib), [take/3]).

nums(N) := ~take(N,nums_from(0)).

:- lazy fun_eval nums_from/1.

nums_from(X) := [X | nums_from(X+1)].
```

A naive reverse example, using functional notation:

```
:- module(_, [nrev/2], [functional]).

nrev( [] )    := [].
nrev( [H|T] ) := ~conc( nrev(T),[H] ).

conc( [],    L ) := L.
conc( [H|T], K ) := [ H | conc(T,K) ].
```

And the same example using some assertions:

```
:- module(_, [nrev/2], [assertions,fsyntax,nativeprops]).
```

```
:- entry nrev/2 : {list, ground} * var.

:- pred nrev(A,B) : list(A) => list(B)
   + ( not_fails, is_det, steps_o( exp(length(A),2) ) ).

nrev( [] )    := [].
nrev( [H|L] ) := ~conc( ~nrev(L),[H] ).


:- pred conc(A,_,_) + ( terminates, is_det, steps_o(length(A)) ).

conc( [],    L ) := L.
conc( [H|L], K ) := [ H | ~conc(L,K) ].
```

Finally, a simple stream creation example where assertions are used to define a safety policy (that no file outside `/tmp` should be opened):

```
:- module(_,[create_streams/2],[fsyntax,assertions,regtypes]).

:- use_module(engine(stream_basic)).

:- entry create_streams(A,B) : list(num,A).

create_streams([])     := [].
create_streams([N|NL]) := [ ~open_file(Fname,write) | ~create_streams(NL) ]
    :-
    app("/tmp/../",~number_codes(N),Fname).
%       app("/tmp/",~number_codes(N),Fname).

app([],L) := L.
app([X|Xs],L) := [X|~app(Xs,L)].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% open_file library:

open_file(Fname,Mode) := ~open(File,Mode) :- atom_codes(File,Fname).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Safety policy:

:- check calls open_file(Fname,_,_) : safe_name(Fname).

:- regtype safe_name/1.    safe_name("/tmp/" || L) :- list(alphnum_code,L).

:- regtype alphnum_code/1. alphnum_code := ~alph_code | ~num_code.

:- regtype alph_code/1.    alph_code := 0'a | 0'b | 0'c | 0'd | 0'e | 0'f .

:- regtype num_code/1.
num_code(0'0). num_code(0'1). num_code(0'2). num_code(0'3). num_code(0'4).
num_code(0'5). num_code(0'6). num_code(0'7). num_code(0'8). num_code(0'9).
num_code(0'.). num_code(0'e). num_code(0'E). num_code(0'+). num_code(0'-).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

60 Definite Clause Grammars

Author(s): The Ciao Development Team.

This library package allows the use of DCGs (Definite Clause Grammars) [Col78, PW80] in a Ciao module/program.

Definite clause grammars are an extension of the well-known context-free grammars. Prolog's grammar rules provide a convenient notation for expressing definite clause grammars. A DCG rule in Prolog takes the general form

```
head --> body.
```

meaning "a possible form for `head` is `body`". Both `body` and `head` are sequences of one or more items linked by the standard Prolog conjunction operator `","`.

> **Note:** support for `phrase/2` and `phrase/3` is offered by the `dcg/dcg_phrase` package. Those predicates may perform code translations at runtime, which in some cases is not desired feature (e.g., make precision of static analysis worse or increasing size of static executables). Thus, we separate by design the static and dynamic behaviours.

Definite clause grammars extend context-free grammars in the following ways:

1. A non-terminal symbol may be any Prolog term (other than a variable or number).
2. A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list `[]`. If the terminal symbols are ASCII character codes, such lists can be written (as elsewhere) as strings. An empty sequence is written as the empty list, `[]` or `""`.
3. Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in `{}` brackets.
4. The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).
5. Alternatives may be stated explicitly in the right-hand side of a grammar rule, using the disjunction operator `;`, or, also, as traditionally in Prolog, using `|` (which is treated specially when this package is loaded).
6. The cut symbol may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in `{}` brackets.

As an example, here is a simple grammar which parses an arithmetic expression (made up of digits and operators) and computes its value.

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).

term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).

number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {0'0=<C, C=<0'9, X is C - 0'0}.
```

In the last rule, `C` is the ASCII code of some digit.

The query

```
?- expr(Z, "-2+3*5+1", []).
```

will compute Z=14. The two extra arguments are explained below.

Now, in fact, grammar rules are merely a convenient "syntactic sugar" for ordinary Prolog clauses. Each grammar rule takes an input string, analyses some initial portion, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output strings are not written explicitly in a grammar rule, but the syntax implicitly defines them. We now show how to translate grammar rules into ordinary clauses by making explicit the extra arguments.

A rule such as

```
p(X) --> q(X).
```

translates into

```
p(X, S0, S) :- q(X, S0, S).
```

If there is more than one non-terminal on the right-hand side, as in

```
p(X, Y) -->
    q(X),
    r(X, Y),
    s(Y).
```

then corresponding input and output arguments are identified, as in

```
p(X, Y, S0, S) :-
    q(X, S0, S1),
    r(X, Y, S1, S2),
    r(Y, S2, S).
```

Terminals are translated using the built-in predicate `'C'/3` (this predicate is not normally useful in itself; it has been given the name `'C'` simply to avoid using up a more useful name). Then, for instance

```
p(X) --> [go,to], q(X), [stop].
```

is translated by

```
p(X, S0, S) :-
    'C'(S0, go, S1),
    'C'(S1, to, S2),
    q(X, S2, S3),
    'C'(S3, stop, S).
```

Extra conditions expressed as explicit procedure calls naturally translate as themselves, e.g.

```
p(X) --> [X], {integer(X), X>0}, q(X).
```

translates to

```
p(X, S0, S) :-
    'C'(S0, X, S1),
    integer(X),
    X>0,
    q(X, S1, S).
```

Similarly, a cut is translated literally.

Terminals on the left-hand side of a rule translate into an explicit list in the output argument of the main non-terminal, e.g.

```
is(N), [not] --> [aint].
```

becomes

```
is(N, S0, [not|S]) :- 'C'(S0, aint, S).
```

Disjunction has a fairly obvious translation, e.g.

```
args(X, Y) -->
    (   dir(X), [to], indir(Y)
    ;   indir(Y), dir(X)
    ).
```

translates to

```
args(X, Y, S0, S) :-
    (   dir(X, S0, S1),
        'C'(S1, to, S2),
        indir(Y, S2, S)
    ;   indir(Y, S0, S1),
        dir(X, S1, S)
    ).
```

60.1 Usage and interface

- **Library usage:**
  `:- use_package(dcg).`
  or
  `:- module(...,...,[dcg]).`
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

61 Phrase support for DCGs

Author(s): Jose F. Morales, The Ciao Development Team.

This library extends the DCG package (`dcg`) with support for the `phrase/2` and `phrase/3` predicates.

Those predicates allow the translation and execution of arbitrary terms as DCGs goals at runtime. Those features, are not always desirable, since arbitrary code execution can negatively affect the precision of static analysis and increasing the size of static executables. This package offers a method to include runtime support for DCGs only when necessary.

61.1 Usage and interface

- **Library usage:**

  `:- use_package(dcg/dcg_phrase).`

  or

  `:- module(...,...,[dcg/dcg_phrase]).`
- **Implicit imports:**
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

62 Backtrackable global variables

Author(s): Jose F. Morales, Rémy Haemmerlé.

This module provides a simple way to assign and read fully backtrackable global variables.

Global variables differ from storing information using dynamic predicates:

- Global variables are semantically equivalent to a dictionary passed around as an implicit pair of variables:
  - only one value is associated to a variable at a time
  - the value associated to a variable can be replaced
  - any assignment is undone on backtracking
  - access cost is proportional to a unification
  - variable sharing is preserved during assignment
- Keys of global variables are local to each module.
- Contrary to dynamic predicates, there is no copy of terms. This is particularly interesting for large terms.

The implementation is based on a globally reachable structure in the heap, where each entry is allocated dynamically for each key.

Note that the current implementation has some limitations (see bug entries in this module).

62.1 Usage and interface

- **Library usage:**
  `:- use_module(library(global_vars)).`
- **Exports:**
  - *Predicates:*
    `setval/2`, `getval/2`, `current/2`.

62.2 Documentation on exports

setval/2: PREDICATE

Usage: `setval(Name,Value)`

Associate the term `Value` with the atom `Name`. If `Name` does not refer to an existing global variable, an unbounded global variable `Value` is created and unified to `Value`. On backtracking the assignment is reversed. If `Name` is not a atom the predicate silently fails.

getval/2: PREDICATE

Usage: `getval(Name,Value)`

Unifies `Value` with the current value of the global variable refered to by the atom `Name`. If `Name` does not refer to an exisiting global variable, a free unbound variable is created and unified with `Value`. If `Name` is not an atom the predicate silently fails.

current/2: PREDICATE

Usage: `current(Name,Value)`

Enumerate all defined variables with their value. The order of enumeration is undefined.

62.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `datafacts_rt`, `mutables`.

- *Packages:*

  `prelude`, `initial`, `condcomp`, `hiord`, `assertions`, `assertions/assertions_basic`, `datafacts`.

63 Mutable terms

Author(s): Rémy Haemmerlé.

This module provides mutable terms i.e. an abstract datatype provides with efficient backtrackable destructive assignment. In other words, any destructive assignments are transparently undone on baktracking. Modifications that are intended to survive backtracking must be done by asserting or retracting dynamic program clauses instead. Mutable must be prefered to destructive assignment of arbitrary terms using `setarg/3` of the module `odd` which does not have safe semantics.

63.1 Usage and interface

- **Library usage:**
  `:- use_module(library(mutables)).`
- **Exports:**
  - *Predicates:*
    `create_mutable/2`, `get_mutable/2`, `update_mutable/2`, `mutable/1`.

63.2 Documentation on exports

create_mutable/2: PREDICATE

Usage: `create_mutable(Datum,Mutable)`

Unifies `Datum` with a freshly created mutable term with initial value `Datum`.

get_mutable/2: PREDICATE

Usage: `get_mutable(Datum,Mutable)`

Unifies `Datum` with the current value of the mutable term `Mutable`. `Mutable` must be a mutable term.

update_mutable/2: PREDICATE

Usage: `update_mutable(Datum,Mutable)`

Updates the current value of the mutable term `Mutable` to become `Datum`. `Mutable` must be a mutable term.

mutable/1: PREDICATE

Usage: `mutable(Term)`

Succeeds if `Term` is currently instantiated to a mutable term.

63.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `datafacts_rt`, `odd`.

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `dcg`, `datafacts`.

64 Multiple argument indexing

Author(s):
Anil Nair (original work), Tom Howland (`http://home.pacbell.net/tomjdnh/pd.html`, derived the original work), Francisco Bueno (initial port to Ciao), Jose F. Morales (improvements in implementation and documentation).

Indexing (in Prolog) is an optimization technique that reduces the search space of the predicates without altering the Prolog semantics.

In the most general case, predicate clauses are tryed on backtracking one after the other, in sequential order. We can call this list of clauses a *try-list*. Indexing is based on removing clauses that are known to fail without any observable output (no side-effects) from the try-list. A typical implementation introduces tests before the actual predicate execution to discriminate among a collection of precomputed specialized try-lists. In the best case, this technique can obtain try-lists with 0 or 1 elements for calls.

Currently, the Ciao engine implements a limited but fast 1st-argument-1st-level indexing. The `indexer` package provides more powerful indexing schemes. It lets you pick different combinations of arguments to index on. E.g., it will let you index on the first and third argument or the second and the third argument of a predicate.

The selection of the try-list is based on computing a hash value for the terms (or part of them) to be indexed upon. Given this, the optimization pays off only when the amount of clashing that your original predicate causes without indexing superseeds the cost of the hashing function. Such amount of course depends on the number and form of the facts in your predicate, and the calling modes.

Important Note about Performance

- The current implementation of the package is done at the source level, so it may sometimes not be as fast as expected.
- The complexity of the hashing function currently used is $O(n)$ with n the number of characters in the textual representation of the term. Thus, even if the search tree is reduced, performance can be much slower in some cases that the cheaper internal (1st argument, 1st level) indexing used in Ciao.

Despite this, the package implements some indexing schemes with **low overhead**.

- A single `:- index p(+,?,...?)` indexer (1st argument, 1st level). Reuses the internal indexing.
- A single `:- index p(?,...,+,...?)` indexer (one argument, 1st level). Reuses the internal indexing by reordering the predicate arguments.

64.1 Usage and interface

- **Library usage:**
  This facility is used as a package, thus either including `indexer` in the package list of the module, or by using the `use_package/1` declaration. The facility predicate `hash_term/2`, documented here, is defined in library module `library(indexer/hash)`.
- **Exports:**
  - *Predicates:*
    `hash_term/2`.

64.2 Documentation on exports

hash_term/2: PREDICATE

`hash_term(Term,HashValue)`

Provides an efficient way to calculate an integer `HashValue` for a ground `Term`.

Usage 1: `hash_term(T,N)`

`N` is a hashing index for `T`.

- *The following properties should hold at call time:*
  `T` is currently ground (it contains no variables). (`term_typing:ground/1`)
  `N` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `N` is an integer. (`basic_props:int/1`)

Usage 2: `hash_term(T,N)`

- *The following properties should hold at call time:*
  `T` is not ground. (`native_props:nonground/1`)
  `N` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `N` is a free variable. (`term_typing:var/1`)

64.3 Documentation on internals

index/1: DECLARATION

Usage: :- `index(IndexSpecs)`.

Declares an indexing scheme for a predicate. Each spec declares an indexing on a combination of the arguments. Indexing will be performed using any of the specs in `IndexSpecs` (being thus interpreted as an or).

You should use a `*` in an argument position if you wish to hash on the entire term in that argument. If a `+` is used only one level of the term in the argument is used for hashing. An `i` is used to indicate that argument is already an integer, and therefore its own value will be used for hashing. The argspec `?` simply indicates not to use the argument for indexing.

For example, the index specification:

```
:- index foo(+,?,*,i), foo(?,?,?,i).
```

declares indexing for `foo/4` either on a combination of the first, third, and fourht arguments, or only on the last argument, which is an integer. In the first case, only the principal functor of the first argument will be used for hashing; the third argument will be used in its entirety.

The argspec `n` is a pragmatic extension and can not be used in conjunction with the other specifiers aside from `?`. It stands for `"nonvar"` and implies that the argument will not be used for hashing, since only ground terms can effectively be used in hashing. Thus, it can not be used in combination with other specifiers within a particular index specification. It is often the fastest thing to use.

- *The following properties should hold upon exit:*
  `IndexSpecs` is an index specification. (`indexer_doc:indexspecs/1`)

indexspecs/1: REGTYPE

An index specification is defined as follows:

```
indexspecs(Spec) :-
    indexspec(Spec).
indexspecs((Spec,Specs)) :-
    indexspec(Spec),
    indexspecs(Specs).

indexspec(Spec) :-
    Spec=..[_F|Args],
    list(argspec,Args).
```

Usage: `indexspecs(IndexSpecs)`

`IndexSpecs` is an index specification.

argspec/1: REGTYPE

An argument hash specification is defined as follows:

```
argspec(+).
argspec(*).
argspec(i).
argspec(n).
argspec(?).
```

Usage: `argspec(Spec)`

`Spec` is an argument hash specification.

64.4 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`native_props`, `hash`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

65 Block declarations

Author(s): Rémy Haemmerlé, Jose F. Morales (documentation improvements).

Version: 0.1 (2008/25/5)

This package provides compatibility with SICStus' block declarations.

Convention: The recommended style is to write the block declarations in front of the source code of the predicate they refer to. Indeed, they are part of the source code of the predicate and must precede the first clause. Moreover it is suggested to use `?` for specifying non conditioned arguments.

Example: The following definition calls to `merge/3` having uninstantiated arguments in the first and third position or in the second and third position will suspend.

```
:- block merge(-,?,-), merge(?,-,-).

merge([], Y, Y).
merge(X, [], X).
merge([H|X], [E|Y], [H|Z]) :- H @< E,  merge(X, [E|Y], Z).
merge([H|X], [E|Y], [E|Z]) :- H @>= E, merge([H|X], Y, Z).
```

Simulating `block` with

: In the predicate above, execution of `merge(X,Y,Z)` is suspended while `(var(X),var(Z);var(Y),var(Z))` holds. A similar effect can be obtained with

`when/2` using the negated condition `((nonvar(X);nonvar(Z)),(nonvar(Y);nonvar(Z)))`.

```
:- use_module(library(when)).

merge(X,Y,Z) :-
    when(((nonvar(X);nonvar(Z)),
          (nonvar(Y);nonvar(Z))), merge_(X,Y,Z)).

merge_([], Y, Y).
merge_(X, [], X).
merge_([H|X], [E|Y], [H|Z]) :- H @< E,  merge(X, [E|Y], Z).
merge_([H|X], [E|Y], [E|Z]) :- H @>= E, merge([H|X], Y, Z).
```

65.1 Usage and interface

- **Library usage:**
  `:- use_package(block).`
  or
  `:- module(...,...,[block]).`
- **New operators defined:**
  `block/1` [**1150,fx**].
- **New declarations defined:**
  `block/1`.
- **Implicit imports:**
  - *Packages:*
    `prelude`,
    `initial`,
    `condcomp`,
    `assertions`,
    `assertions/assertions_basic`.

65.2 Documentation on new declarations

block/1: DECLARATION

Usage: :- `block(BlockSpecs)`.

`BlockSpecs` **specifies a disjunction of conditions. Each condition is of the form** `predname(C1, ..., CN)` **where each** `CI` **is either a** `-` **if the call must suspend until the corresponding argument is bound, or anything else otherwise.**

- *The following properties should hold at call time:*
  `BlockSpecs` **is a sequence or list of** `cgoals`**.**

  (`basic_props:sequence_or_list/2`)

66 Delaying predicates (freeze)

Author(s): Remy Haemmerle, Manuel Carro, Daniel Cabeza.

This library offers a simple implementation of `freeze/2`, `frozen/2`, etc. [Col82, Nai85, Nai91, Car87] based on the use of attributed variables [Hol92, Hol90].

66.1 Usage and interface

- **Library usage:**
  `:- use_module(library(freeze)).`
- **Exports:**
  - *Predicates:*
    `freeze/2`, `frozen/2`.

66.2 Documentation on exports

freeze/2: PREDICATE

Usage: `freeze(X,Goal)`

If `X` is free delay `Goal` until `X` is non-variable.

- *The following properties should hold at call time:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `freeze(?,goal)`.

frozen/2: PREDICATE

Usage: `frozen(X,Goal)`

`Goal` is currently delayed until variable `X` becomes bound.

- *The following properties should hold upon exit:*
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `frozen(?,goal)`.

66.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`.

67 Delaying predicates (when)

Author(s): Manuel Carro, Remy Haemmerle.

`when/2` delays a predicate until some condition in its variable is met. For example, we may want to find out the maximum of two numbers, but we are not sure when they will be instantiated. We can write the standard `max/3` predicate (but changing its name to `gmax/3` to denote that the first and second arguments must be ground) as

```
gmax(X, Y, X):- X > Y, !.
gmax(X, Y, Y):- X =< Y.
```

and then define a 'safe' `max/3` as

```
max(X, Y, Z):-
    when((ground(X),ground(Y)), gmax(X, Y, Z)).
```

which can be called as follows:

```
?- max(X, Y, Z) , Y = 0, X = 8.

X = 8,
Y = 0,
Z = 8 ?

yes
```

Alternatively, `max/3` could have been defined as

```
max(X, Y, Z):-
    when(ground((X, Y)), gmax(X, Y, Z)).
```

with the same effects as above. More complex implementations are possible. Look, for example, at the `max.pl` implementation under the `when` library directory, where a `max/3` predicate is implemented which waits on all the arguments until there is enough information to determine their values:

```
?- use_module(library(when/max)).

yes
?- max(X, Y, Z), Z = 5, Y = 4.

X = 5,
Y = 4,
Z = 5 ?

yes
```

67.1 Usage and interface

- **Library usage:**
  `:- use_module(library(when)).`
- **Exports:**
  - *Predicates:*
    `when/2`.
  - *Regular Types:*
    `wakeup_exp/1`.

67.2 Documentation on exports

when/2: PREDICATE

Usage: `when(WakeupCond,Goal)`

Delays / executes `Goal` according to `WakeupCond` given. The `WakeupConds` now acceptable are `ground(T)` (`Goal` is delayed until `T` is ground), `nonvar(T)` (`Goal` is delayed until `T` is not a variable), and conjunctions and disjunctions of conditions:

```
wakeup_exp(ground(_1)).
wakeup_exp(nonvar(_1)).
wakeup_exp((C1,C2)) :-
    wakeup_exp(C1),
    wakeup_exp(C2).
wakeup_exp((C1;C2)) :-
    wakeup_exp(C1),
    wakeup_exp(C2).
```

`when/2` only fails it the `WakeupCond` is not legally formed. If `WakeupCond` is met at the time of the call no delay mechanism is involved — but there exists a time penalty in the condition checking.

In case that an instantiation fires the execution of several predicates, the order in which these are executed is not defined.

– *The following properties should hold at call time:*

  `WakeupCond` **is a legal expression for delaying goals.** (`when:wakeup_exp/1`)

  `Goal` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `when(?,goal)`.

wakeup_exp/1: REGTYPE

Usage: `wakeup_exp(T)`

`T` **is a legal expression for delaying goals.**

67.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

  `terms_vars`, `sort`, `sets`.

– *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

68 Andorra execution

Author(s): Claudio Vaucheret, Francisco Bueno.

This package allows the execution under the Basic Andorra Model [War88]. The model classifies goals as a *determinate goal*, if at most one clause matches the goal, or nondeterminate goal, otherwise. In this model a goal is delayed until either it becomes determinate or it becomes the leftmost goal and no determinate goal is available. The implementation of this selection rule is based on the use of attributed variables [Hol92, Hol90].

In order to test determinacy we verify only the heads of clauses and builtins in the bodies of clauses before the first cut, if any. By default, determinacy of a goal is detected dynamically: when called, if at most one clause matches, it is executed; otherwise, it is delayed. For goals delayed the test is repeated each time a variable appearing in the goal is instantiated. In addition, efficiency can be improved by using declarations that specify the determinacy conditions. These will be considered for testing instead of the generic test on all clauses that can match.

As with any other Ciao package, the andorra computation rule affects only the module that uses the package. If execution passes across two modules that use the computation rule, determinate goals are run in advance *within* one module and also within the other module. But determinate goals of one module do not run ahead of goals of the other module.

It is however possible to preserve the computation rule for calls to predicates defined in other modules. These modules should obviously also use this package. In addition *all* predicates from such modules should imported, i.e., the directive `:- use_module(module)`, should be used in this case instead of `:- use_module(module,[...])`. Otherwise calls to predicates outside the module will only be called when they became the leftmost goal.

68.1 Usage and interface

- **Library usage:**
  `:- use_package(andorra).`
  or
  `:- module(...,...,[andorra]).`
- **Exports:**
  - *Regular Types:*
    `detcond/1`, `path/1`.
- **New operators defined:**
  `?\=/2` [700,xfx], `?=/2` [700,xfx].
- **New declarations defined:**
  `determinate/2`.
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

68.2 Documentation on new declarations

determinate/2: DECLARATION

`:- determinate(Pred,Cond).`

Declares determinacy conditions for a predicate. Conditions `Cond` are on variables of arguments of `Pred`. For example, in:

```
:- determinate(member(A,B,C), ( A ?= term(B,[1])  ; C?=[_|_]) ).

member(A,[A|B],B).
member(A,[B|C],[B|D]) :-
    A==B,
    member(A,C,D).
```

the declaration states that a call `member(A,B,C)` is determinate when either `A` doesn't unify with the first argument of `B` or `C` doesn't unify with `[_|_]`.

Usage: :- `determinate(Pred,Cond)`.

States that the predicate `Pred` is determinate when `Cond` holds.

- *The following properties should hold at call time:*

  `Pred` **is a predicate name.** (`basic_props:predname/1`)

  `Cond` **is a determinacy condition.** (`user(... /andorra_doc):detcond/1`)

68.3 Documentation on exports

detcond/1: REGTYPE

Defined by:

```
detcond(ground(X)) :-
    var(X).
detcond(nonvar(X)) :-
    var(X).
detcond(instatiated(A,Path)) :-
    var(A),
    list(int,Path).
detcond(?\=(Term1,Term2)) :-
    path(Term1),
    path(Term2).
detcond(?=(Term1,Term2)) :-
    path(Term1),
    path(Term2).
detcond(Test) :-
    test(Test).
```

- `ground/1` **and** `nonvar/1` **have the usual meaning.**
- `instatiated(A,Path)` **means that the subterm of `A` addressed by `Path` is not a variable. `Path` is a list of integer numbers describing a path to the subterm regarding the whole term A as a tree. For example, `instantiated(f(g(X),h(i(Z),Y)),[2,1])` tests whether `i(Z)` is not a variable.**
- `Term1 ?\= Term2` **means "terms `Term1` and `Term2` do not unify (when instantiated)". `Term1` and `Term2` can be either an argument of the predicate or a term `term(V,Path)`, which refers to the subterm of `V` addressed by `Path`.**
- `Term1 ?= Term2` **means "terms `Term1` and `Term2` unify (when instantiated)". The same considerations above apply to `Term1` and `Term2`.**
- **any other test that does not unify variables can also be used (`==/2`, `\==/2`, `atomic/1`).**

Usage: `detcond(X)`

`X` **is a determinacy condition.**

path/1: REGTYPE

Defined by:

```
path(X) :-
    var(X).
path(X) :-
    list(int,X).
```

68.4 Other information

The andorra transformation will include the following predicates into the code of the module that uses the package. Be careful not to define predicates by these names:

- `detcond_andorra/4`
- `path_andorra/4`
- `detcond_susp/4`
- `path_susp/4`
- `list_andorra2/5`
- `test_andorra2/4`

69 Call on determinate

Author(s): Jose F. Morales, Manuel Carro.

Offers an enriched variant of call and cut `!!/0` which executes pending goals when the computation has no more alternatives.

This library is useful to, for example, get rid of external connections once the necessary data has been obtained.

69.1 Usage and interface

- **Library usage:**
  `:- use_package(det_hook).`
  or
  `:- module(...,...,[det_hook]).`
- **Implicit imports:**
  - *System library modules:*
    `det_hook_rt`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

69.2 Other information

As an example, the program

```
:- module(_, _, [det_hook]).

enumerate(X):-
    display(enumerating), nl,
    OnCut = (display('goal cut'), nl),
    OnFail = (display('goal failed'), nl),
    det_try(enum(X), OnCut, OnFail).

enum(1).
enum(2).
enum(3).
```

behaves as follows:

```
?- enumerate(X).
enumerating

X = 1 ? ;

X = 2 ? ;

X = 3 ? ;
goal failed
```

(note the message inserted on failure). The execution can be cut as follows:

```
?- use_package(det_hook).
{Including /home/clip/lib/ciao/ciao-1.7/library/det_hook/det_hook.pl
}

yes
?- enumerate(X), '!!'.
enumerating
goal cut

X = 1 ? ;

no
```

70 Runtime predicates for call on determinate

Author(s): Jose F. Morales, Manuel Carro.

Implementation of variant of call and cut which executes pending goals when the computation has no more alternatives.

70.1 Usage and interface

- **Library usage:**
  `:- use_module(library(det_hook/det_hook_rt)).`
- **Exports:**
  - *Predicates:*
    `det_try/3`.

70.2 Documentation on exports

det_try/3: PREDICATE

Usage: `det_try(Goal,OnCut,OnFail)`

`Action` **is called, and** `OnCut` **and** `OnFail` **are goals to be executed when** `Goal` **is cut or when it finitely fails, respectively. In order for this to work, cutting must be performed in a special way, by using the** `!!/0` **predicate, also provided by this module.**

- *The following properties should hold at call time:*
  `Goal` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
  `OnCut` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
  `OnFail` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `det_try(goal,goal,goal)`.

70.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `datafacts`.

71 Lazy evaluation

Author(s): Amadeo Casas, Jose F. Morales (minor modifications).

This library package allows the use of lazy evaluation in a Ciao module/program.

Lazy Evaluation is a program evaluation technique used particularly in functional languages. When using lazy evaluation, an expression is not evaluated as soon as it is assigned, but rather when the evaluator is forced to produce the value of the expression. Although the `when` or `freeze` control primitives present in many modern logic programming systems are more powerful than lazy evaluation, they lack the simplicity of use and cleaner semantics of functional lazy evaluation.

The objective of this package is to allow evaluating the functions lazily. Functions are the subset of relations (predicates) which have a designated argument through which a single output is obtained for any set of inputs (the other arguments). In logic programming systems which have syntactic support for functions (including Ciao), functions are typically translated to predicates whose `last` argument is designated as a (single value) output and the rest as inputs.

In our proposal, a function can be declared as lazy via the following declaration:

```
:- lazy fun_eval f/N.
```

This function could be represented as:

```
:- lazy fun_eval f(~_,_,_,_).
```

where ~ indicates the argument through which the single output will be obtained. Another possible representation may be:

```
:- lazy fun_return f(~_,_,_,_).
```

In order to achieve the intended behavior, the execution of each function declared as lazy is suspended until the return value of the function is needed.

A simple example of the use of lazy evaluation would be the definition of a function which returns the (potentially) infinite list of integers starting with a given one:

```
:- lazy fun_eval nums_from/1.
nums_from(X) := [X | nums_from(X+1)].
```

While lazy functions certainly increase the overhead in the execution, they also allow the user to develop in an easy way predicates which can handle infinite terms, and this is the main advantage of the proposed functionality.

Lazy evaluation can be also a better option than eager evaluation when a function in a different module is used and it returns a big amount of data. As an example, we have the following module `module1`:

```
:- module(module1, [test/1], [fsyntax, lazy, hiord]).

:- use_module(library(lazy/lazy_lib), [nums_from/2, takeWhile/3]).
:- use_module(module2, [squares/2]).
:- use_module(library(arithpreds)).

:- fun_eval test/0.
test := ~takeWhile((''(X) :- X < 10000), ~squares(~nums_from(1))).
```

and another module `module2`:

```
:- module(module1, [test/1], [fsyntax, lazy, hiord]).

:- use_module(library(lazy/lazy_lib), [nums_from/2, takeWhile/3]).
```

```
:- use_module(module2, [squares/2]).
:- use_module(library(arithpreds)).

:- fun_eval test/0.
test := ~takeWhile((''(X) :- X < 10000), ~squares(~nums_from(1))).
```

Function `test/0` in module `m1` needs to execute function `squares/1`, in module `m2`, which will return a very long list (in the case of this example this list will be infinite, but the conclusions also apply with finite but long lists). If `squares/1` were executed eagerly then the entire list would be returned, to immediately execute the `take/2` function with the entire list, but creating this intermediate result is wasteful in terms of memory requirements. In order to solve this problem, the `squares/1` function could be moved to module `m1` and merged with `take/2` (or, also, they could exchange a size parameter). But rearranging the program is not always possible and may perhaps complicate other aspects of the overall program design.

If instead the `squares/1` function is evaluated lazily, it is possible to keep the definitions unchanged and in different modules and there will be a smaller memory penalty for storing the intermediate result. As more values are needed by the `take/2` function, more values in the list returned by `squares/1` are built (in this example, only 10 values). These values that have been consumed and passed over will be recovered by the garbage collector and the corresponding memory freed. The query:

```
?- test(X).
```

will compute `X` = [1,4,9,16,25,36,49,64,81,100].

Some common lazy predicates are provided in `lazy_lib`.

71.1 Usage and interface

- **Library usage:**
  `:- use_package(lazy).`
  or
  `:- module(...,...,[lazy]).`
- **New operators defined:**
  `lazy/1` [1170,fx].
- **Implicit imports:**
  - *System library modules:*
    `freeze`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

71.2 Other information

The translation of the code in order to execute it lazily is explained below.

A sentence translation is provided to handle the `lazy` directives. The translation of a lazy function into a predicate is done in two steps. First, the function is converted into a predicate (using the fsyntax package). Then, the resulting predicate is transformed to suspend its execution until the value of the last variable (i.e., the output variable) is needed. This suspension

is achieved by the use of the `freeze/1` control primitive that many modern logic programming systems implement quite efficiently (`block` or `when` declarations can obviously also be used, but we explain the transformation in terms of `freeze` because it is more widespread). The translation will rename the original predicate to an internal name and add a bridge predicate with the original name which invokes the internal predicate through a call to `freeze/1`. This will delay the execution of the internal predicate until its result is required, which will be detected as a binding (i.e., demand) of its output variable.

We show now an example of the use of lazy evaluation, and how a lazy function is translated by this package. The following code returns an (infinite) list of fibonacci numbers:

```
:- lazy fun_eval fiblist/0.
fiblist := [0, 1 | ~zipWith(add, FibL, ~tail(FibL))]
    :- FibL = fiblist.
```

which is translated into:

```
fiblist(X) :-
      freeze(X, 'fiblist_$$lazy$$'(X)).

'fiblist_$$lazy$$'([0, 1 | Rest]) :-
      fiblist(FibL),
      tail(FibL, T),
      zipWith(add, FibL, T, Rest).
```

In the `fiblist` function defined, any element in the resulting infinite list of fibonacci numbers can be referenced, as for example, `nth(X, ~fiblist, Value)..` The other functions used in the definition are `tail/2` and `zipWith/3`. These two functions can be found in the lazy_lib.pl runtime file.

72 Lazy evaluation library

Author(s): Amadeo Casas, Jose F. Morales.

This module provides several predicates that make easier to develop predicates that will be executed lazily.

72.1 Usage and interface

- **Library usage:**
  `:- use_module(library(lazy/lazy_lib)).`
- **Exports:**
  - *Predicates:*
    `nums_from/2`, `nums_from_inc/3`, `repeat/2`, `cycle/2`, `take/3`, `takeWhile/3`, `drop/3`, `dropWhile/3`, `splitAt/3`, `span/3`, `tail/2`, `lazy_map/3`, `lazy_foldr/4`, `lazy_foldl/4`, `zipWith/4`.

72.2 Documentation on exports

nums_from/2: PREDICATE

Usage: `nums_from(X,List)`

`List` **is unified with an infinite list of successive numbers starting in** `X`

- *The following properties should hold at call time:*
  - `X` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `List` **is a free variable.** (`term_typing:var/1`)
  - `X` **is an integer.** (`basic_props:int/1`)
  - `List` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  - `X` **is an integer.** (`basic_props:int/1`)
  - `List` **is a list of** `ints`**.** (`basic_props:list/2`)

nums_from_inc/3: PREDICATE

Usage: `nums_from_inc(X,Y,List)`

`List` **is unified with an infinite list of successive numbers starting in** `X` **with an increment of** `Y`

- *The following properties should hold at call time:*
  - `X` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `Y` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `List` **is a free variable.** (`term_typing:var/1`)
  - `X` **is an integer.** (`basic_props:int/1`)
  - `Y` **is an integer.** (`basic_props:int/1`)
  - `List` **is a free variable.** (`term_typing:var/1`)

- *The following properties should hold upon exit:*
  X is an integer. (basic_props:int/1)
  Y is an integer. (basic_props:int/1)
  List is a list of ints. (basic_props:list/2)

repeat/2: PREDICATE

Usage: `repeat(X,List)`

List is unified with an infinite list of the term Y

- *The following properties should hold at call time:*
  X is currently a term which is not a free variable. (term_typing:nonvar/1)
  List is a free variable. (term_typing:var/1)
  X is any term. (basic_props:term/1)
  List is a free variable. (term_typing:var/1)
- *The following properties should hold upon exit:*
  X is any term. (basic_props:term/1)
  List is a list of terms. (basic_props:list/2)

cycle/2: PREDICATE

Usage: `cycle(X,List)`

List is unified with an infinite list of the term Y repeated infinite times

- *The following properties should hold at call time:*
  X is currently a term which is not a free variable. (term_typing:nonvar/1)
  List is a free variable. (term_typing:var/1)
  X is any term. (basic_props:term/1)
  List is a free variable. (term_typing:var/1)
- *The following properties should hold upon exit:*
  X is any term. (basic_props:term/1)
  List is a list of terms. (basic_props:list/2)

take/3: PREDICATE

Usage: `take(X,ListA,ListR)`

ListR is unified with the first X elements of the infinite list ListA

- *The following properties should hold at call time:*
  X is currently a term which is not a free variable. (term_typing:nonvar/1)
  ListA is currently a term which is not a free variable. (term_typing:nonvar/1)
  ListR is a free variable. (term_typing:var/1)
  X is a counter. (lazy_lib:counter/1)
  ListA is a list of terms. (basic_props:list/2)
  ListR is any term. (basic_props:term/1)
- *The following properties should hold upon exit:*
  X is a counter. (lazy_lib:counter/1)
  ListA is a list of terms. (basic_props:list/2)
  ListR is a list of terms. (basic_props:list/2)

takeWhile/3: PREDICATE

Usage: `takeWhile(P,ListA,ListR)`

`ListR` is unified with the first elements of the infinite list `ListA` while the condition `P` is true

- *The following properties should hold at call time:*

  `P` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `ListA` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `ListR` is a free variable. (`term_typing:var/1`)

  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `ListA` is a list of `terms`. (`basic_props:list/2`)

  `ListR` is any term. (`basic_props:term/1`)
- *The following properties should hold upon exit:*

  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `ListA` is a list of `terms`. (`basic_props:list/2`)

  `ListR` is a list of `terms`. (`basic_props:list/2`)

drop/3: PREDICATE

Usage: `drop(X,ListA,ListR)`

`ListR` is unified with the infinite list `ListA` dropping the first `X` elements

- *The following properties should hold at call time:*

  `X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `ListA` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `ListR` is a free variable. (`term_typing:var/1`)

  `X` is a counter. (`lazy_lib:counter/1`)

  `ListA` is a list of `terms`. (`basic_props:list/2`)

  `ListR` is any term. (`basic_props:term/1`)
- *The following properties should hold upon exit:*

  `X` is a counter. (`lazy_lib:counter/1`)

  `ListA` is a list of `terms`. (`basic_props:list/2`)

  `ListR` is a list of `terms`. (`basic_props:list/2`)

dropWhile/3: PREDICATE

Usage: `dropWhile(P,ListA,ListR)`

`ListR` is unified with the infinite list `ListA` dropping the first elements while the condition `P` is true

- *The following properties should hold at call time:*

  `P` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `ListA` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `ListR` is a free variable. (`term_typing:var/1`)

  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `ListA` is a list of `terms`. (`basic_props:list/2`)

  `ListR` is any term. (`basic_props:term/1`)

– *The following properties should hold upon exit:*
  P is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)
  ListA is a list of terms. (basic_props:list/2)
  ListR is a list of terms. (basic_props:list/2)

splitAt/3: PREDICATE

Usage: `splitAt(X,ListA,Res)`

Res is unified with a tuple of lists where the first list is composed by the first X elements of the list ListA and the second list is composed by the rest of the elements of ListA

– *The following properties should hold at call time:*
  X is currently a term which is not a free variable. (term_typing:nonvar/1)
  ListA is currently a term which is not a free variable. (term_typing:nonvar/1)
  Res is a free variable. (term_typing:var/1)
  X is a counter. (lazy_lib:counter/1)
  ListA is a list of terms. (basic_props:list/2)
  Res is any term. (basic_props:term/1)

– *The following properties should hold upon exit:*
  X is a counter. (lazy_lib:counter/1)
  ListA is a list of terms. (basic_props:list/2)
  This type represents a tuple of lists. (lazy_lib:tuple_of_lists/1)

span/3: PREDICATE

Usage: `span(P,ListA,Res)`

Res is unified with a tuple of lists where the first list is composed by the elements of ListA which verify the condition P and the second list is composed by the rest of the elements of the initial list

– *The following properties should hold at call time:*
  P is currently a term which is not a free variable. (term_typing:nonvar/1)
  ListA is currently a term which is not a free variable. (term_typing:nonvar/1)
  Res is a free variable. (term_typing:var/1)
  P is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)
  ListA is a list of terms. (basic_props:list/2)
  Res is any term. (basic_props:term/1)

– *The following properties should hold upon exit:*
  P is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)
  ListA is a list of terms. (basic_props:list/2)
  This type represents a tuple of lists. (lazy_lib:tuple_of_lists/1)

tail/2: PREDICATE

Usage: `tail(ListA,ListR)`

`ListR` **is unified with the tail of the infinite list** `ListA`

- *The following properties should hold at call time:*

  `ListA` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `ListR` **is a free variable.** (`term_typing:var/1`)

  `ListA` **is a list of** `terms`**.** (`basic_props:list/2`)

  `ListR` **is any term.** (`basic_props:term/1`)

- *The following properties should hold upon exit:*

  `ListA` **is a list of** `terms`**.** (`basic_props:list/2`)

  `ListR` **is a list of** `terms`**.** (`basic_props:list/2`)

lazy_map/3: PREDICATE

Usage: `lazy_map(ListA,P,ListR)`

Version of the map/3 predicate to be executed lazily

- *The following properties should hold at call time:*

  `ListA` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `P` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `ListR` **is a free variable.** (`term_typing:var/1`)

  `ListA` **is a list of** `terms`**.** (`basic_props:list/2`)

  `P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

  `ListR` **is any term.** (`basic_props:term/1`)

- *The following properties should hold upon exit:*

  `ListA` **is a list of** `terms`**.** (`basic_props:list/2`)

  `P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

  `ListR` **is a list of** `terms`**.** (`basic_props:list/2`)

Meta-predicate **with arguments:** `lazy_map(?,pred(2),?)`.

lazy_foldr/4: PREDICATE

Usage: `lazy_foldr(P,Xs,V0,V)`

Lazy version of `foldr/4`

- *Call and exit should be compatible with:*

  `P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

  `Xs` **is a list of** `terms`**.** (`basic_props:list/2`)

  `V0` **is any term.** (`basic_props:term/1`)

  `V` **is any term.** (`basic_props:term/1`)

- *The following properties should hold at call time:*

  `P` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Xs` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `V0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `V` **is a free variable.** (`term_typing:var/1`)

Meta-predicate **with arguments:** `lazy_foldr(pred(3),?,?,?)`.

lazy_foldl/4: PREDICATE

Usage: `lazy_foldl(P,Xs,V0,V)`

Lazy version of `foldl/4`

– *Call and exit should be compatible with:*

`P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

`Xs` **is a list of** `terms`**.** (`basic_props:list/2`)

`V0` **is any term.** (`basic_props:term/1`)

`V` **is any term.** (`basic_props:term/1`)

– *The following properties should hold at call time:*

`P` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Xs` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`V0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`V` **is a free variable.** (`term_typing:var/1`)

Meta-predicate **with arguments:** `lazy_foldl(pred(3),?,?,?)`.

zipWith/4: PREDICATE

Usage: `zipWith(P,ListA,ListB,ListR)`

`ListR` **is a list whose elements are calculated from the function** `P` **and the elements of input lists** `ListA` **and** `ListB` **occuring at the same position in both lists**

– *The following properties should hold at call time:*

`P` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`ListA` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`ListB` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`ListR` **is a free variable.** (`term_typing:var/1`)

`P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

`ListA` **is a list of** `terms`**.** (`basic_props:list/2`)

`ListB` **is a list of** `terms`**.** (`basic_props:list/2`)

`ListR` **is any term.** (`basic_props:term/1`)

– *The following properties should hold upon exit:*

`P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

`ListA` **is a list of** `terms`**.** (`basic_props:list/2`)

`ListB` **is a list of** `terms`**.** (`basic_props:list/2`)

`ListR` **is a list of** `terms`**.** (`basic_props:list/2`)

Meta-predicate **with arguments:** `zipWith(pred(3),?,?,?)`.

72.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`freeze`, `arithpreds`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `regtypes`, `fsyntax`, `assertions`, `assertions/assertions_basic`, `basicmodes`, `hiord`, `lazy`.

73 Breadth-first execution

Author(s): Daniel Cabeza, Manuel Carro, Manuel Hermenegildo.

This package implements breadth-first execution of predicates. This may be useful in search problems when a proof procedure is needed that will find all solutions (even if it may still loop for some failures). This is in contrast with the default depth-first search, which may loop in some cases even if there are correct answers to a given query. This library is also useful when experimenting with pure programs as well as when teaching logic programming, for illustrating the expected theoretical results that should be expected from the declarative semantics (see for example the slides in `http://www.cliplab.org/proglog`).

It is important to realize, however, that the improved behaviour of breadth first execution comes at a high (exponential!) price in terms of both time ad memory. This library allows the programmer to control this overhead by selecting which predicates will be executed in breadth-first mode and which predicates in depth-first mode. More concretely, predicates written with operators `'<-'/1` (facts) and `'<-'/2` (clauses) are executed using breadth-first search, while predicates using the standard syntax will be executed depth-first.

The following example implements two versions of a predicate meant to succeed if two nodes of a directed graph are connected. The `chain/2` predicate (which will be executed depth-first) loops without finding the connection between `a` and `d`, while the `bfchain/2` predicate (which will be executed breadth-first) will find the connection correctly:

```
:- module(chain, _, [bf]).

test(bf) :- bfchain(a,d).
test(df) :- chain(a,d).   % loops!

bfchain(X,X) <- .
bfchain(X,Y) <- arc(X,Z), bfchain(Z,Y).

chain(X,X).
chain(X,Y) :- arc(X,Z), chain(Z,Y).

arc(a,b).
arc(a,d).
arc(b,c).
arc(c,a).
```

A second package, `'sr/bfall'`, allows executing *all* the predicates in a given module in breadth-first mode. In this case, predicates should be written using the standard syntax. This is useful to be able to switch easily between depth-first and breadth-first execution (e.g., for testing purposes) for all predicates in a given module without having to modify the program. The following program (written in standard syntax) runs breadth-first:

```
:- module(chain_bfall, _, [sr/bfall]).
%:- module(chain_bfall, _, [sr/afall]).
%:- module(chain_bfall, _, [sr/bf]).

test :- chain(a,d).

chain(X,X).
chain(X,Y) :- arc(X,Z), chain(Z,Y).

arc(a,b).
```

```
arc(a,d).
arc(b,c).
arc(c,a).
```

There is another version, package 'sr/af', which ensures AND-fairness by goal shuffling. This reduces the number of cases in which an execution that is a failure loops instead (infinite failures) at a small additional cost. For example, by using 'sr/af' the following code correctly answers "no" when executing test/0:

```
:- module(sublistapp, [test/0, sublistapp/2], [sr/af]).

test :- sublistapp([a], [b]).

sublistapp(S, L) <- append(_, S, Y), append(Y, _, L).

append([], L, L) <- .
append([X|Xs], L, [X|Ys]) <- append(Xs, L, Ys).
```

There is also a package 'sr/afall' which again allows executing *all* the predicates in a given module in breadth-first, and-fair mode, where also all predicates should be written using the standard syntax. This package offers (at a cost, of course) very nice results for many programs, and is used extensively in programming courses by the Ciao developers.

Finally, it should be noted that a separate library, id, implements *iterative-deepening* search, which can in many cases be a better alternative to breadth-first search, since it achieves the same improvement in the completeness results in many cases at a greatly reduced execution cost (but the enumeration order of solutions is not as nice, and that is why these packages are very attractive for prototyping and teaching).

73.1 Usage and interface

- **Library usage:**
  `:- use_package(sr/bf).`
  or
  `:- module(...,...,[sr/bf]).`
- **New operators defined:**
  `<-/2` [**1200,xfx**], `<-/1` [**1200,xf**].
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

74 Iterative-deepening execution

Author(s): Rémy Haemmerlé, Manuel Carro, Claudio Vaucheret, Manuel Hermenegildo.

This package applies a *compiling control* technique to implement *depth first iterative deepening* execution [Kor85]. It changes the usual *depth-first* search rule by *iterative-deepening* on those predicates specifically marked. This is very useful in search problems when a complete proof procedure is needed.

When this search rule is used, first all goals are expanded only up to a given depth. If no solution is found or more solutions are needed by backtracking, the depth limit is incremented and the whole goal is repeated. Although it might seem that this approach is very inefficient because all higher levels are repeated for the deeper ones, it has been shown that is performs only about b/(b - 1) times as many operations than the corresponding breadth-first search, (where b is the branching factor of the proof tree) while the waste of memory is the same as depth first.

The usage is by means of the following directive:

`:- iterative(Name, FirstCut, Formula).`

which states than the predicate 'Name' given in functor/arity form will be executed using iterative deepening rule starting at the depth 'FirstCut' with depth being incremented by the predicate 'Formula'. This predicate compute the new depth using the previous one. It must implement a dilating function i.e. the new depth must be greater. For example, to start with depth 5 and increment by 10 you can write:

`:- iterative(p/1,5,f).`

`f(X,Y) :- Y is X + 10.`

or if you prefer,

`:- iterative(p/1,5,(_(X,Y):- Y is X + 10)).`

You can also use a fourth parameter to set a limiting depth. All goals below the given depth limit simply fail. Thus, with the following directive:

`:- iterative(p/1,5,(_(X,Y):- Y is X + 10),100).`

all goals deeper than 100 will fail.

An example of code using this package would be:

```
:- module(example_id, _,[id]).

test(id) :-
    idchain(a,d).
test(df) :-
    chain(a,d).   % loops!

:- iterative(idchain/2, 3, ( _(X,Z) :- Z is X + 1) ).

idchain(X,X).
idchain(X,Y) :-
    arc(X,Z),
    idchain(Z,Y).

chain(X,X).
chain(X,Y) :-
    arc(X,Z),
    chain(Z,Y).
```

```
arc(a,b).
arc(a,d).
arc(b,c).
arc(c,a).
```

The order of solutions are first the shallower and then the deeper. Solutions which are between two cutoff are given in the usual left to right order. For example,

```
:- module(_,_,[id]).

% All goals deeper than 2 will fail
:- iterative(p/1,0,(_(X,Z) :- Z is X + 1),2).

% Change the solutions' order to goal p(X).
%:- iterative(p/1,1,(_(X,Z) :- Z is X + 3)).

p(X) :- q(X).
p(a).

q(X) :- r(X).
q(b).

r(X) :- s(X).
r(c).

s(d).
```

Another complete proof procedure implemented is the `bf` **package (breadth first execution).**

74.1 Usage and interface

- **Library usage:**
  `:- use_package(id).`
  or
  `:- module(...,...,[id]).`
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

75 Miscellaneous predicates

Author(s): Manuel Carro, Daniel Cabeza.

This module implements some miscellaneous non-logical (but sometimes very useful) predicates.

75.1 Usage and interface

- **Library usage:**
  `:- use_module(library(odd)).`
- **Exports:**
  - *Predicates:*
    `setarg/3`, `undo/1`.

75.2 Documentation on exports

setarg/3: PREDICATE

Usage: `setarg(Index,Term,NewArg)`

Replace destructively argument `Index` in `Term` by `NewArg`. The assignment is undone on backtracking. This is a major change to the normal behavior of data assignment in Ciao Prolog.

- *The following properties should hold at call time:*
  `Index` **is currently instantiated to an integer.** (`term_typing:integer/1`)
  `Term` **is a compound term.** (`basic_props:struct/1`)
  `NewArg` **is any term.** (`basic_props:term/1`)

undo/1: PREDICATE

Usage: `undo(Goal)`

`call(Goal)` **is executed on backtracking. This is a major change to the normal control of Ciao Prolog execution.**

- *The following properties should hold at call time:*
  `Goal` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*
  `Goal` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `undo(goal)`.

75.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

76 Aggregates: gathering predicate solutions

Author(s): Richard A. O'Keefe (first version), David H.D. Warren (first version), Mats Carlsson (changes), Daniel Cabeza, Manuel Hermenegildo.

This module implements the standard solution aggregation predicates.

When there are many solutions to a problem, and when all those solutions are required to be collected together, this can be achieved by repeatedly backtracking and gradually building up a list of the solutions. The following built-in predicates are provided to automate this process.

76.1 Usage and interface

- **Library usage:**
  `:- use_module(library(aggregates)).`
- **Exports:**
  - *Predicates:*
    `setof/3`, `bagof/3`, `findall/3`, `findall/4`, `findnsols/4`, `findnsols/5`, `^/2`.

76.2 Documentation on exports

setof/3: PREDICATE

`setof(Template,Generator,Set)`

Finds the `Set` of instances of the `Template` satisfying `Generator`. The set is in ascending order (see Chapter 33 [Comparing terms], page 165 for a definition of this order) without duplicates, and is non-empty. If there are no solutions, `setof` fails. `setof` may succeed in more than one way, binding free variables in `Generator` to different values. This can be avoided by using existential quantifiers on the free variables in front of `Generator`, using `^/2`. For example, given the clauses:

```
father(bill, tom).
father(bill, ann).
father(bill, john).
father(harry, july).
father(harry, daniel).
```

The following query produces two alternative solutions via backtracking:

```
?- setof(X,father(F,X),Sons).

F = bill,
Sons = [ann,john,tom] ? ;

F = harry,
Sons = [daniel,july] ? ;

no
?-
```

Usage: ⟨ • ISO • ⟩

– *Call and exit should be compatible with:*

`Set` is a list. (`basic_props:list/1`)

- *The following properties should hold at call time:*

  `Template` is any term. (`basic_props:term/1`)

  `Generator` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*

  `Template` is any term. (`basic_props:term/1`)

  `Set` is a list. (`basic_props:list/1`)
- *The following properties should hold globally:*

  `Template` is not further instantiated. (`basic_props:not_further_inst/2`)

Meta-predicate **with arguments:** `setof(?,goal,?)`.

Other properties:

`setof(X,Y,Z)`

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP as `findall(X,Y,Z)`. (`basic_props:native/2`)

bagof/3: PREDICATE

`bagof(Template,Generator,Bag)`

Finds all the instances of the `Template` produced by the `Generator`, and returns them in the `Bag` in the order in which they were found. If the `Generator` contains free variables which are not bound in the `Template`, it assumes that this is like any other Prolog question and that you want bindings for those variables. This can be avoided by using existential quantifiers on the free variables in front of the `Generator`, using `^/2`.

Usage: ⟨• ISO •⟩

- *Call and exit should be compatible with:*

  `Bag` is a list. (`basic_props:list/1`)
- *The following properties should hold at call time:*

  `Template` is any term. (`basic_props:term/1`)

  `Generator` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*

  `Template` is any term. (`basic_props:term/1`)

  `Bag` is a list. (`basic_props:list/1`)
- *The following properties should hold globally:*

  `Template` is not further instantiated. (`basic_props:not_further_inst/2`)

Meta-predicate **with arguments:** `bagof(?,goal,?)`.

Other properties:

`bagof(X,Y,Z)`

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP as `findall(X,Y,Z)`. (`basic_props:native/2`)

findall/3: PREDICATE

`findall(Template,Generator,List)`

A special case of bagof, where all free variables in the `Generator` are taken to be existentially quantified. Faster than the other aggregation predicates.

Usage: ⟨ • ISO • ⟩

- *Calls should, and exit will be compatible with:*
  `List` **is a list.** (`basic_props:list/1`)
- *The following properties should hold at call time:*
  `Template` **is any term.** (`basic_props:term/1`)
  `Generator` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties hold upon exit:*
  `Template` **is any term.** (`basic_props:term/1`)
  `List` **is a list.** (`basic_props:list/1`)
- *The following properties hold globally:*
  `Template` **is not further instantiated.** (`basic_props:not_further_inst/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  All the calls of the form `findall(Template,Generator,List)` **do not fail.** (`native_props:not_fails/1`)
  All calls of the form `findall(Template,Generator,List)` **are deterministic.** (`native_props:is_det/1`)

Meta-predicate **with arguments:** `findall(?,goal,?)`.

findall/4: PREDICATE

Usage:

As `findall/3`**, but returning in** `Tail` **the tail of** `List` **(findall(**`Template`, `Generator`, `List`, `Tail`**)).**

- *Call and exit should be compatible with:*
  `Arg3` **is any term.** (`basic_props:term/1`)
  `Arg4` **is any term.** (`basic_props:term/1`)
- *The following properties should hold at call time:*
  `Arg1` **is any term.** (`basic_props:term/1`)
  `Arg2` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*
  `Arg1` **is any term.** (`basic_props:term/1`)
  `Arg3` **is any term.** (`basic_props:term/1`)
  `Arg4` **is any term.** (`basic_props:term/1`)
- *The following properties should hold globally:*
  `Arg1` **is not further instantiated.** (`basic_props:not_further_inst/2`)

Meta-predicate **with arguments:** `findall(?,goal,?,?)`.

findnsols/4: PREDICATE

`findnsols(N,Template,Generator,List)`

As `findall/3`, but generating at most `N` solutions of `Generator`. Thus, the length of `List` will not be greater than `N`. If `N`=<0, returns directly an empty list. This predicate is especially useful if `Generator` may have an infinite number of solutions.

Usage:

- *Call and exit should be compatible with:*
  `List` **is a list.** (`basic_props:list/1`)
- *The following properties should hold at call time:*
  `N` **is an integer.** (`basic_props:int/1`)
  `Template` **is any term.** (`basic_props:term/1`)
  `Generator` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*
  `Template` **is any term.** (`basic_props:term/1`)
  `List` **is a list.** (`basic_props:list/1`)
- *The following properties should hold globally:*
  `Template` **is not further instantiated.** (`basic_props:not_further_inst/2`)

Meta-predicate **with arguments:** `findnsols(?,?,goal,?)`.

findnsols/5: PREDICATE

`findnsols(N,Template,Generator,List,Tail)`

As `findnsols/4`, but returning in `Tail` the tail of `List`.

Usage:

- *The following properties should hold at call time:*
  `N` **is an integer.** (`basic_props:int/1`)
  `Template` **is any term.** (`basic_props:term/1`)
  `Generator` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*
  `Template` **is any term.** (`basic_props:term/1`)
- *The following properties should hold globally:*
  `Template` **is not further instantiated.** (`basic_props:not_further_inst/2`)

Meta-predicate **with arguments:** `findnsols(?,?,goal,?,?)`.

^/2: PREDICATE

Usage: `X^P`

Existential quantification: `X` is existentially quantified in `P`. E.g., in `A^p(A,B)`, `A` is existentially quantified. Used only within aggregation predicates. In all other contexts, simply, execute the procedure call `P`.

- *The following properties should hold at call time:*
  `X` **is a free variable.** (`term_typing:var/1`)
  `P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `? ^goal`.

Other properties:

`_X^Y`

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP as `call(Y)`. (`basic_props:native/2`)

76.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `native_props`, `datafacts_rt`, `sort`, `lists`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`, `nativeprops`, `hiord`, `datafacts`.

77 Fast/concurrent update of facts

Author(s): Daniel Cabeza, Manuel Carro.

Prolog implementations traditionally implement the concept of dynamic predicates: predicates which can be inspected or modified at run-time, adding or deleting individual clauses. The power of this feature comes at a cost: as new clause bodies can be arbitrarily added to the program, new predicate calls can arise which are not 'visible' at compile-time, thus complicating global analysis and optimization of the code. But it is the case that most of the time what the programmer wants is simply to store data, with the purpose of sharing it between search branches, predicates, or even execution threads. In Ciao the concept of data predicate serves this purpose: a data predicate is a predicate composed exclusively by facts, which can be inspected, and dynamically added or deleted, at run-time. Using data predicates instead of normal dynamic predicates brings benefits in terms of speed, but above all makes the code much easier to analyze automatically and thus allows better optimization.

Also, a special kind of data predicates exists, *concurrent predicates*, **which can be used to communicate/synchronize among different execution threads (see Chapter 88 [Low-level concurrency/multithreading primitives], page 395).**

Data predicates must be declared through a `data/1` **declaration. Concurrent data predicates must be declared through a** `concurrent/1` **declaration. The allowed operations on data predicates are defined in** `datafacts_rt`**.**

77.1 Usage and interface

- **Library usage:**
  `:- use_package(datafacts).`
  or
  `:- module(...,...,[datafacts]).`
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

78 Fast/concurrent update of facts (runtime)

Author(s): The Ciao Development Team, Daniel Cabeza, Manuel Carro.

This module implements the assert/retract family of predicates to manipulate data predicates (facts).

78.1 Usage and interface

- **Library usage:**
  Do not use this module directly (use the `datafacts` package instead).
- **Exports:**
  - *Predicates:*
    `asserta_fact/1`, `asserta_fact/2`, `assertz_fact/1`, `assertz_fact/2`, `current_fact/1`, `current_fact/2`, `retract_fact/1`, `retractall_fact/1`, `current_fact_nb/1`, `retract_fact_nb/1`, `close_predicate/1`, `open_predicate/1`, `set_fact/1`, `erase/1`.
  - *Regular Types:*
    `reference/1`.

78.2 Documentation on exports

asserta_fact/1: PREDICATE

`asserta_fact(Fact)`

`Fact` is added to the corresponding data predicate. The fact becomes the first clause of the predicate concerned.

Usage:

- *The following properties should hold at call time:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `asserta_fact(fact)`.

asserta_fact/2: PREDICATE

`asserta_fact(Fact,Ref)`

Same as `asserta_fact/1`, instantiating `Ref` to a unique identifier of the asserted fact.

Usage:

- *The following properties should hold at call time:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `Ref` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Ref` is a reference of a dynamic or data clause. (`datafacts_rt:reference/1`)

Meta-predicate with arguments: `asserta_fact(fact,?)`.

assertz_fact/1: PREDICATE

`assertz_fact(Fact)`

`Fact` is added to the corresponding data predicate. The fact becomes the last clause of the predicate concerned.

Usage:

- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `assertz_fact(fact)`.

assertz_fact/2: PREDICATE

`assertz_fact(Fact,Ref)`

Same as `assertz_fact/1`, instantiating `Ref` to a unique identifier of the asserted fact.

Usage:

- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Ref` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*

  `Ref` is a reference of a dynamic or data clause. (`datafacts_rt:reference/1`)

Meta-predicate **with arguments:** `assertz_fact(fact,?)`.

current_fact/1: PREDICATE

`current_fact(Fact)`

Gives on backtracking all the facts defined as data or concurrent which unify with `Fact`. It is faster than calling the predicate explicitly, which do invoke the meta-interpreter. If the `Fact` has been defined as concurrent and has not been closed, `current_fact/1` will wait (instead of failing) for more clauses to appear after the last clause of `Fact` is returned.

Usage:

- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `current_fact(fact)`.

current_fact/2: PREDICATE

`current_fact(Fact,Ref)`

`Fact` is a fact of a data predicate and `Ref` is its reference identifying it uniquely.

Usage 1:

Gives on backtracking all the facts defined as data which unify with `Fact`, instantiating `Ref` to a unique identifier for each fact.

- *The following properties should hold at call time:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `Ref` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Ref` is a reference of a dynamic or data clause. (`datafacts_rt:reference/1`)

Usage 2:

Given `Ref`, unifies `Fact` with the fact identified by it.

- *Call and exit should be compatible with:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties should hold at call time:*
  `Ref` is a reference of a dynamic or data clause. (`datafacts_rt:reference/1`)
- *The following properties should hold upon exit:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `current_fact(fact,?)`.

retract_fact/1: PREDICATE

`retract_fact(Fact)`

Unifies `Fact` with the first matching fact of a data predicate, and then erases it. On backtracking successively unifies with and erases new matching facts. If `Fact` is declared as concurrent and is non- closed, `retract_fact/1` will wait for more clauses or for the closing of the predicate after the last matching clause has been removed.

Usage:

- *The following properties should hold at call time:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*
  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `retract_fact(fact)`.

retractall_fact/1: PREDICATE

`retractall_fact(Fact)`

Erase all the facts of a data predicate unifying with `Fact`. Even if all facts are removed, the predicate continues to exist.

Usage:

- *The following properties should hold at call time:*
  Fact is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)
- *The following properties should hold upon exit:*
  Fact is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

Meta-predicate with arguments: `retractall_fact(fact)`.

current_fact_nb/1: PREDICATE

`current_fact_nb(Fact)`

Behaves as `current_fact/1` but a fact is never waited on even if it is concurrent and non-closed.

Usage:

- *The following properties should hold at call time:*
  Fact is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)
- *The following properties should hold upon exit:*
  Fact is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

Meta-predicate with arguments: `current_fact_nb(fact)`.

retract_fact_nb/1: PREDICATE

`retract_fact_nb(Fact)`

Behaves as `retract_fact/1`, but never waits on a fact, even if it has been declared as concurrent and is non- closed.

Usage:

- *The following properties should hold at call time:*
  Fact is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)
- *The following properties should hold upon exit:*
  Fact is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

Meta-predicate with arguments: `retract_fact_nb(fact)`.

close_predicate/1: PREDICATE

`close_predicate(Pred)`

Changes the behavior of the predicate `Pred` if it has been declared as a concurrent predicate: calls to this predicate will fail (instead of wait) if no more clauses of `Pred` are available.

Usage:

- *The following properties should hold at call time:*
  Pred is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)

- *The following properties should hold upon exit:*
  `Pred` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `close_predicate(fact)`.

open_predicate/1: PREDICATE

`open_predicate(Pred)`

Reverts the behavior of concurrent predicate `Pred` to waiting instead of failing if no more clauses of `Pred` are available.

Usage:

- *The following properties should hold at call time:*
  `Pred` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*
  `Pred` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `open_predicate(fact)`.

set_fact/1: PREDICATE

`set_fact(Fact)`

Sets `Fact` as the unique fact of the corresponding data predicate.

Usage:

- *The following properties should hold at call time:*
  `Fact` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*
  `Fact` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `set_fact(fact)`.

erase/1: PREDICATE

`erase(Ref)`

Deletes the clause referenced by `Ref`.

Usage:

- *The following properties should hold at call time:*
  `Ref` **is a reference of a dynamic or data clause.** (`datafacts_rt:reference/1`)
- *The following properties should hold upon exit:*
  `Ref` **is a reference of a dynamic or data clause.** (`datafacts_rt:reference/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

reference/1: REGTYPE

Usage: `reference(R)`

`R` **is a reference of a dynamic or data clause.**

78.3 Documentation on internals

data/1: DECLARATION

Usage: :- `data(Predicates).`

Defines each predicate in `Predicates` as a data predicate. If a predicate is defined data in a file, it must be defined data in every file containing clauses for that predicate. The directive should precede all clauses of the affected predicates. This directive is defined as a prefix operator in the compiler.

– *The following properties should hold at call time:*

`Predicates` **is a sequence or list of** `prednames`. (`basic_props:sequence_or_list/2`)

concurrent/1: DECLARATION

Usage: :- `concurrent(Predicates).`

Defines each predicate in `Predicates` as a concurrent predicate. If a predicate is defined concurrent in a file, it must be defined concurrent in every file containing clauses for that predicate. The directive should precede all clauses of the affected predicates. This directive is defined as a prefix operator in the compiler.

– *The following properties should hold at call time:*

`Predicates` **is a sequence or list of** `prednames`. (`basic_props:sequence_or_list/2`)

78.4 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`noprelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`.

79 Dynamic predicates (not source preserving)

Author(s): The Ciao Development Team.

This module implements the assert/retract family of predicates to manipulate dynamic predicates.

> **This module does not preserve the original source definition of dynamic predicates. That is, `clause/2` may obtain the asserted clauses in lower-level expanded form rather than in the original shape. Use `dynamic_clauses` if you need consulting the original form.**

The defined predicates (see `dynamic_rt`) allow modification of the program as it is actually running. Clauses can be added to the program (*asserted*) or removed from the program (*retracted*). For these predicates, the argument which corresponds to the clause head must be instantiated to an atom or a compound term. The argument corresponding to the clause must be instantiated either to a term `Head :- Body` or, if the body part is empty, to `Head`. An empty body part is represented as `true`.

Note that using this library is very detrimental to global analysis, and that for most uses the predicates listed in Chapter 77 [Fast/concurrent update of facts], page 353 suffice.

79.1 Usage and interface

- **Library usage:**

  `:- use_package(dynamic).`

  or

  `:- module(...,...,[dynamic]).`
- **Implicit imports:**
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

80 Dynamic predicates (not source preserving) (runtime)

Author(s): The Ciao Development Team.

This module implements the assert/retract family of predicates to manipulate dynamic predicates. This module does not preserve the original source definition of dynamic predicates.

80.1 Usage and interface

- **Library usage:**
  Do not use this module directly (use the `dynamic` package instead).
- **Exports:**
  - *Predicates:*
    `asserta/1`, `asserta/2`, `assertz/1`, `assertz/2`, `assert/1`, `assert/2`, `retract/1`, `retractall/1`, `abolish/1`, `clause/2`, `clause/3`, `wellformed_body/3`, `current_predicate/1`, `current_predicate/2`, `dynamic/1`, `data/1`.
  - *Multifiles:*
    `do_on_abolish/1`.

80.2 Documentation on exports

asserta/1: PREDICATE

Usage: `asserta(Clause)` ⟨• ISO •⟩

The current instance of `Clause` is interpreted as a clause and is added to the current program. The predicate concerned must be dynamic. The new clause becomes the *first* clause for the predicate concerned. Any uninstantiated variables in `Clause` will be replaced by new private variables.

- *The following properties should hold at call time:*
  `dynamic_rt:cl(Clause)` (`dynamic_rt:cl/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate with arguments: `asserta(clause)`.

asserta/2: PREDICATE

Usage: `asserta(Clause,Ref)`

Like `asserta/1`. `Ref` is a unique identifier of the asserted clause.

- *The following properties should hold at call time:*
  `Clause` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Ref` is a free variable. (`term_typing:var/1`)
  `dynamic_rt:cl(Clause)` (`dynamic_rt:cl/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate with arguments: `asserta(clause,?)`.

assertz/1: PREDICATE

Usage: `assertz(Clause)` ⟨• ISO •⟩

Like `asserta/1`, except that the new clause becomes the *last* clause for the predicate concerned.

- *The following properties should hold at call time:*
  `Clause` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `dynamic_rt:cl(Clause)` (`dynamic_rt:cl/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate **with arguments:** `assertz(clause)`.

assertz/2: PREDICATE

Usage: `assertz(Clause,Ref)`

Like `assertz/1`. `Ref` is a unique identifier of the asserted clause.

- *The following properties should hold at call time:*
  `Clause` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Ref` **is a free variable.** (`term_typing:var/1`)
  `dynamic_rt:cl(Clause)` (`dynamic_rt:cl/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate **with arguments:** `assertz(clause,?)`.

assert/1: PREDICATE

Usage: `assert(Clause)`

Identical to `assertz/1`. Included for compatibility.

- *The following properties should hold at call time:*
  `Clause` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `dynamic_rt:cl(Clause)` (`dynamic_rt:cl/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate **with arguments:** `assert(clause)`.

assert/2: PREDICATE

Usage: `assert(Clause,Ref)`

Identical to `assertz/2`. Included for compatibility.

- *The following properties should hold at call time:*
  `Clause` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Ref` **is a free variable.** (`term_typing:var/1`)
  `dynamic_rt:cl(Clause)` (`dynamic_rt:cl/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate **with arguments:** `assert(clause,?)`.

retract/1: PREDICATE

Usage: `retract(Clause)` ⟨• ISO •⟩

The first clause in the program that matches `Clause` is erased. The predicate concerned must be dynamic.

The predicate `retract/1` may be used in a non-determinate fashion, i.e., it will successively retract clauses matching the argument through backtracking. If reactivated by backtracking, invocations of the predicate whose clauses are being retracted will proceed unaffected by the retracts. This is also true for invocations of `clause` for the same predicate. The space occupied by a retracted clause will be recovered when instances of the clause are no longer in use.

- *The following properties should hold at call time:*

  `Clause` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `dynamic_rt:cl(Clause)` (`dynamic_rt:cl/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate **with arguments:** `retract(clause)`.

retractall/1: PREDICATE

Usage: `retractall(Head)`

Erase all clauses whose head matches `Head`, where `Head` must be instantiated to an atom or a compound term. The predicate concerned must be dynamic. The predicate definition is retained.

- *The following properties should hold at call time:*

  `Head` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Head` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate **with arguments:** `retractall(fact)`.

abolish/1: PREDICATE

Usage: `abolish(Spec)` ⟨• ISO •⟩

Erase all clauses of the predicate specified by the predicate spec `Spec`. The predicate definition itself is also erased (the predicate is deemed undefined after execution of the abolish). The predicates concerned must all be user defined.

- *The following properties should hold at call time:*

  `Spec` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Spec` **is a predicate name.** (`basic_props:predname/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate **with arguments:** `abolish(spec)`.

clause/2: PREDICATE

Usage: `clause(Head,Body)` ⟨ • ISO • ⟩

The clause 'Head :- Body' exists in the current program. The predicate concerned must be dynamic.

- *The following properties should hold at call time:*

  `Head` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Head` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*

  `dynamic_rt:body(Body)` (`dynamic_rt:body/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate **with arguments:** `clause(fact,?)`.

clause/3: PREDICATE

`clause(Head,Body,Ref)`

Like `clause(Head,Body)`**, plus the clause is uniquely identified by** `Ref`**.**

Usage 1: `clause(Head,Body,Ref)`

`Head` **must be instantiated to an atom or a compound term.**

- *The following properties should hold at call time:*

  `Head` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Head` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties should hold upon exit:*

  `dynamic_rt:body(Body)` (`dynamic_rt:body/1`)

  `Ref` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

Usage 2: `clause(Head,Body,Ref)`

`Ref` **must be instantiated to a valid identifier.**

- *The following properties should hold at call time:*

  `Ref` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties should hold upon exit:*

  `Head` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

  `dynamic_rt:body(Body)` (`dynamic_rt:body/1`)

Meta-predicate **with arguments:** `clause(fact,?,?)`.

wellformed_body/3: PREDICATE

`wellformed_body(BodyIn,Env,BodyOut)`

`BodyIn` **is a well-formed clause body.** `BodyOut` **is its counterpart with no single-variable meta-goals (i.e., with** `call(X)` **for** `X`**).** `Env` **denotes if global cuts are admissible in** `BodyIn` **(+ if they are, - if they are not).**

current_predicate/1: PREDICATE

Usage: `current_predicate(Spec)` ⟨• ISO •⟩

A predicate in the current module is named `Spec`.

- *The following properties should hold upon exit:*
  `Spec` **is a predicate name.** (`basic_props:predname/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

current_predicate/2: PREDICATE

Usage: `current_predicate(Spec,Module)`

A predicate in `Module` is named `Spec`. `Module` never is an engine module.

- *The following properties should hold upon exit:*
  `Spec` **is a predicate name.** (`basic_props:predname/1`)
  `Module` **is an internal module identifier** (`basic_props:internal_module_id/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

dynamic/1: PREDICATE

`dynamic(Spec)`

`Spec` **is of the form `F/A`. The predicate named `F` with arity `A` is made dynamic in the current module at runtime (useful for predicate names generated on-the-fly). If the predicate functor name `F` is uninstatiated, a new, unique, predicate name is generated at runtime.**

Usage: `dynamic(Spec)`

- *The following properties should hold at call time:*
  `Spec` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Spec` **is a predicate name.** (`basic_props:predname/1`)

data/1: PREDICATE

`data(Spec)`

`Spec` **is of the form `F/A`. The predicate named `F` with arity `A` is made data in the current module at runtime (useful for predicate names generated on-the-fly). If the predicate functor name `F` is uninstatiated, a new, unique, predicate name is generated at runtime.**

Usage: `data(Spec)`

- *The following properties should hold at call time:*
  `Spec` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Spec` **is a predicate name.** (`basic_props:predname/1`)

80.3 Documentation on multifiles

do_on_abolish/1: PREDICATE

`do_on_abolish(Head)`

A hook predicate which will be called when the definition of the predicate of `Head` is abolished.

Usage: `do_on_abolish(G)`

- *The following properties should hold at call time:*

  `G` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

The predicate is *multifile*.

80.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `datafacts_rt`.
- *Packages:*

  `noprelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `regtypes`, `datafacts`.

81 Dynamic predicates (source preserving)

Author(s): Daniel Cabeza, The Ciao Development Team.

The package `dynamic_clauses` provides the assert/retract family of predicates to manipulate dynamic predicates.

The defined predicates (see `dynamic_clauses_rt`) allow modification of the program as it is actually running. Clauses can be added to the program (*asserted*) or removed from the program (*retracted*), as well as inspected. Note that in Ciao only the dynamic predicates of the current module (or accessible dynamic multifile predicates) can be accessed and modified. This limits the bad impact to global analysis of this dynamic modification of the program. Thus, if dynamic predicates are exported, to be able to inspect or modify them externally some accessing predicates need to be implemented and exported alongside.

For the inspecting/manipulating predicates, the argument which corresponds to the clause head must be instantiated to an atom or a compound term. The argument corresponding to the clause must be instantiated either to a term `Head :- Body` or, if the body part is empty, to `Head`. An empty body part is represented as `true`.

Note that using this library is very detrimental to global analysis, and that for most uses the predicates listed in Chapter 77 [Fast/concurrent update of facts], page 353 suffice.

Example:

```
:- module(_,[test/1],[dynamic_clauses]).

% A simple (and very artificial) example of self-modifying code.

:- dynamic loop/1.

test(Xs) :-
    clause(initial_loop(Xs), Body),
    retractall(loop(_)),
    assertz((loop(Xs) :- Body)),
    loop(Xs).

:- dynamic initial_loop/1. % (otherwise we do not get the clause)
initial_loop([100|Xs]) :-
    mutate,
    loop(Xs).

mutate :-
    clause(loop([N|Xs]), Body),
    retractall(loop(_)),
    ( N = 0 ->
        assertz(loop([]))
    ; N1 is N - 1,
      assertz((loop([N1|Xs]) :- Body))
    ).
```

81.1 Usage and interface

- **Library usage:**
  `:- use_package(dynamic_clauses).`
  or
  `:- module(...,...,[dynamic_clauses]).`
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

82 Dynamic predicates (source preserving) (runtime)

Author(s): Daniel Cabeza, The Ciao Development Team.

This module implements the assert/retract family of predicates to manipulate dynamic predicates, preserving the original source definitions.

82.1 Usage and interface

- **Library usage:**
  Do not use this module directly (use the `dynamic_clauses` **package instead).**
- **Exports:**
  - *Predicates:*
    `asserta/1`, `asserta/2`, `assertz/1`, `assertz/2`, `assert/1`, `assert/2`, `retract/1`, `retractall/1`, `abolish/1`, `clause/2`, `mfclause/2`, `current_predicate/1`, `current_predicate/2`, `dynamic/1`, `data/1`, `wellformed_body/3`.
  - *Multifiles:*
    `do_on_abolish/1`.

82.2 Documentation on exports

asserta/1: PREDICATE

Usage: `asserta(Clause)` ⟨ • ISO • ⟩

The current instance of `Clause` **is interpreted as a clause and is added to the current program. The predicate concerned must be dynamic. The new clause becomes the** *first* **clause for the predicate concerned. Any uninstantiated variables in** `Clause` **will be replaced by new private variables.**

- *The following properties should hold at call time:*
  `Clause` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Clause` **is a well-formed clause** (`dynamic_clauses_rt:clause/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

asserta/2: PREDICATE

Usage: `asserta(Clause,Ref)`

Like `asserta/1`. `Ref` **is a unique identifier of the asserted clause.**

- *The following properties should hold at call time:*
  `Clause` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Ref` **is a free variable.** (`term_typing:var/1`)
  `Clause` **is a well-formed clause** (`dynamic_clauses_rt:clause/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

assertz/1: PREDICATE

Usage: `assertz(Clause)` ⟨• ISO •⟩

Like `asserta/1`, except that the new clause becomes the *last* clause for the predicate concerned.

- *The following properties should hold at call time:*

  `Clause` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Clause` is a well-formed clause (`dynamic_clauses_rt:clause/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

assertz/2: PREDICATE

Usage: `assertz(Clause,Ref)`

Like `assertz/1`. `Ref` is a unique identifier of the asserted clause.

- *The following properties should hold at call time:*

  `Clause` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Ref` is a free variable. (`term_typing:var/1`)

  `Clause` is a well-formed clause (`dynamic_clauses_rt:clause/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

assert/1: PREDICATE

Usage: `assert(Clause)`

Identical to `assertz/1`. Included for compatibility.

- *The following properties should hold at call time:*

  `Clause` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Clause` is a well-formed clause (`dynamic_clauses_rt:clause/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

assert/2: PREDICATE

Usage: `assert(Clause,Ref)`

Identical to `assertz/2`. Included for compatibility.

- *The following properties should hold at call time:*

  `Clause` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Ref` is a free variable. (`term_typing:var/1`)

  `Clause` is a well-formed clause (`dynamic_clauses_rt:clause/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

retract/1: PREDICATE

Usage: `retract(Clause)` ⟨• ISO •⟩

The first clause in the program that matches `Clause` is erased. The predicate concerned must be dynamic.

The predicate `retract/1` may be used in a non-determinate fashion, i.e., it will successively retract clauses matching the argument through backtracking. If reactivated by backtracking, invocations of the predicate whose clauses are being retracted will proceed unaffected by the retracts. This is also true for invocations of `clause` for the same predicate. The space occupied by a retracted clause will be recovered when instances of the clause are no longer in use.

- *The following properties should hold at call time:*

  `Clause` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Clause` **is a well-formed clause** (`dynamic_clauses_rt:clause/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

retractall/1: PREDICATE

Usage: `retractall(Head)`

Erase all clauses whose head matches `Head`, where `Head` must be instantiated to an atom or a compound term. The predicate concerned must be dynamic. The predicate definition is retained.

- *The following properties should hold at call time:*

  `Head` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Head` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

abolish/1: PREDICATE

Usage: `abolish(Spec)` ⟨• ISO •⟩

Erase all clauses of the predicate specified by the predicate spec `Spec`. The predicate definition itself is also erased (the predicate is deemed undefined after execution of the abolish). The predicates concerned must all be user defined.

- *The following properties should hold at call time:*

  `Spec` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Spec` **is a predicate name.** (`basic_props:predname/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Meta-predicate **with arguments:** `abolish(spec)`.

clause/2: PREDICATE

Usage: `clause(Head,Body)` ⟨• ISO •⟩

The clause '`Head :- Body`' exists in the current module. The predicate concerned must be dynamic.

- *The following properties should hold at call time:*

  Head is currently a term which is not a free variable. (term_typing:nonvar/1)

  Head is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)
- *The following properties should hold upon exit:*

  Body is a clause body (dynamic_clauses_rt:body/1)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (basic_props:native/1)

mfclause/2: PREDICATE

Usage: mfclause(Head,Body) ⟨• ISO •⟩

There is a clause 'Head :- Body' of a dynamic multifile predicate accessible from this module.

- *The following properties should hold at call time:*

  Head is currently a term which is not a free variable. (term_typing:nonvar/1)

  Head is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)
- *The following properties should hold upon exit:*

  Body is a clause body (dynamic_clauses_rt:body/1)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (basic_props:native/1)

current_predicate/1: PREDICATE

Usage: current_predicate(Spec) ⟨• ISO •⟩

A predicate in the current module is named Spec.

- *The following properties should hold upon exit:*

  Spec is a predicate name. (basic_props:predname/1)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (basic_props:native/1)

current_predicate/2: PREDICATE

Usage: current_predicate(Spec,Module)

A predicate in Module is named Spec. Module never is an engine module.

- *The following properties should hold upon exit:*

  Spec is a predicate name. (basic_props:predname/1)

  Module is an internal module identifier (basic_props:internal_module_id/1)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (basic_props:native/1)

dynamic/1: PREDICATE

`dynamic(Spec)`

`Spec` is of the form `F/A`. The predicate named `F` with arity `A` is made dynamic in the current module at runtime (useful for predicate names generated on-the-fly). If the predicate functor name `F` is uninstatiated, a new, unique, predicate name is generated at runtime.

Usage: `dynamic(Spec)`

- *The following properties should hold at call time:*
  `Spec` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Spec` is a predicate name. (`basic_props:predname/1`)

data/1: PREDICATE

`data(Spec)`

`Spec` is of the form `F/A`. The predicate named `F` with arity `A` is made data in the current module at runtime (useful for predicate names generated on-the-fly). If the predicate functor name `F` is uninstatiated, a new, unique, predicate name is generated at runtime.

Usage: `data(Spec)`

- *The following properties should hold at call time:*
  `Spec` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Spec` is a predicate name. (`basic_props:predname/1`)

erase/1: PREDICATE

`erase(Ref)`

Deletes the clause referenced by `Ref`, the identifier obtained by using `asserta/2` or `assertz/2`.

wellformed_body/3: PREDICATE

`wellformed_body(BodyIn,Env,BodyOut)`

`BodyIn` is a well-formed clause body. `BodyOut` is its counterpart with no single-variable meta-goals (i.e., with `call(X)` for `X`). `Env` denotes if global cuts are admissible in `BodyIn` (`+` if they are, `-` if they are not).

82.3 Documentation on multifiles

do_on_abolish/1: PREDICATE

`do_on_abolish(Head)`

A hook predicate which will be called when the definition of the predicate of `Head` is abolished.

Usage: `do_on_abolish(G)`

- *The following properties should hold at call time:*
  `G` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

The predicate is *multifile*.

82.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `datafacts_rt`, `iso_misc`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `regtypes`, `datafacts`.

83 Persistent predicate database

Author(s): José Manuel Gómez Pérez, Daniel Cabeza, Manuel Hermenegildo, The Ciao Development Team.

83.1 Introduction to persistent predicates

This library implements a *generic persistent predicate database*. The basic notion implemented by the library is that of a persistent predicate. The persistent predicate concept provides a simple, yet powerful generic persistent data access method [CHGT98, Par97]. A persistent predicate is a special kind of dynamic, data predicate that "resides" in some persistent medium (such as a set of files, a database, etc.) that is typically external to the program using such predicates. The main effect is that any changes made to a persistent predicate from a program "survive" across executions. I.e., if the program is halted and restarted the predicate that the new process sees is in precisely the same state as it was when the old process was halted (provided no change was made in the meantime to the storage by other processes or the user).

Persistent predicates appear to a program as ordinary predicates, and calls to these predicates can appear in clause bodies in the usual way. However, the definitions of these predicates do not appear in the program. Instead, the library maintains automatically the definitions of predicates which have been declared as persistent in the persistent storage.

Updates to persistent predicates can be made using enhanced versions of `asserta_fact/1`, `assertz_fact/1` and `retract_fact/1`. The library makes sure that each update is a transactional update, in the sense that if the update terminates, then the permanent storage has definitely been modified. For example, if the program making the updates is halted just after the update and then restarted, then the updated state of the predicate will be seen. This provides security against possible data loss due to, for example, a system crash. Also, due to the atomicity of the transactions, persistent predicates allow concurrent updates from several programs.

83.2 File-based backend

The concept of persistent predicates provided by this library essentially implements a light-weight, simple, and at the same time powerful form of relational database (a deductive database), and which is standalone, in the sense that it does not require external support, other than the file management capabilities provided by the operating system. This is due to the fact that the persistent predicates are in fact stored in one or more auxiliary files below a given directory.

This type of database is specially useful when building small to medium-sized standalone applications in Prolog which require persistent storage. In many cases it provides a much easier way of implementing such storage than using files under direct program control. For example, interactive applications can use persistent predicates to represent their internal state in a way that is close to the application. The persistence of such predicates then allows automatically restoring the state to that at the end of a previous session. Using persistent predicates amounts to simply declaring some predicates as such and eliminates having to worry about opening files, closing them, recovering from system crashes, etc.

83.3 Other backends

In other cases, however, it may be convenient to use a relational database as persistent storage. This may be the case, for example, when the data already resides in such a database (where it is perhaps accessed also by other applications) or the volume of data is very large. We distribute these backend separately from the core libraries.

A nice characteristic of the notion of persistent predicates is that it abstracts away how the predicate is actually stored. Thus, a program can use persistent predicates stored in files or in external relational databases interchangeably, and the type of storage used for a given predicate can be changed without having to modify the program (except for replacing the corresponding `persistent/2` declarations).

83.4 Using file-based persistent predicates

Persistent predicates can be declared statically, using `persistent/2` declarations (which is the preferred method, when possible), or dynamically via calls to `make_persistent/2`. Currently, persistent predicates may only contain facts, i.e., they are *dynamic* predicates of type `data/1`.

Predicates declared as persistent are linked to directory, and the persistent state of the predicate will be kept in several files below that directory. The files in which the persistent predicates are stored are in readable, plain ASCII format, and in Prolog syntax. One advantage of this approach is that such files can also be created or edited by hand, in a text editor, or even by other applications.

An example definition of a persistent predicate implemented by files follows:

```
:- persistent(p/3,dbdir).

persistent_dir(dbdir, '/home/clip/public_html/db').
```

The first line declares the predicate `p/3` persistent. The argument `dbdir` is a key used to index into a fact of the relation `persistent_dir/2-4`, which specifies the directory where the corresponding files will be kept. The effect of the declaration, together with the `persistent_dir/2-4` fact, is that, although the predicate is handled in the same way as a normal data predicate, in addition the system will create and maintain efficiently a persistent version of `p/3` via files in the directory `/home/clip/public_html/db`.

The level of indirection provided by the `dbdir` argument makes it easy to place the storage of several persistent predicates in a common directory, by specifying the same key for all of them. It also allows changing the directory for several such persistent predicates by modifying only one fact in the program. Furthermore, the `persistent_dir/2-4` predicate can even be dynamic and specified at run-time.

83.5 Implementation Issues

We outline the current implementation approach. This implementation attempts to provide at the same time efficiency and security. To this end, up to three files are used for each predicate (the persistence set): the data file, the operations file, and the backup file. In the updated state the facts (tuples) that define the predicate are stored in the data file and the operations file is empty (the backup file, which contains a security copy of the data file, may or may not exist).

While a program using a persistent predicate is running, any insertion (assert) or deletion (retract) operations on the predicate are performed on both the program memory and on the persistence set. However, in order to incurr only a small overhead in the execution, rather than changing the data file directly, a record of each of the insertion and deletion operations is *appended* to the operations file. The predicate is then in a transient state, in that the contents of the data file do not reflect exactly the current state of the corresponding predicate. However, the complete persistence set does.

When a program starts, all pending operations in the operations file are performed on the data file. A backup of the data file is created first to prevent data loss if the system crashes during this operation. The order in which this updating of files is done ensures that, if at any point the process dies, on restart the data will be completely recovered. This process of updating the persistence set can also be triggered at any point in the execution of the program (for example, when halting) by calling `update_files`.

83.6 Defining an initial database

It is possible to define an initial database by simply including in the program code facts of persistent predicates. They will be included in the persistent database when it is created. They are ignored in successive executions.

83.7 Using persistent predicates from the top level

Special care must be taken when loading into the top level modules or user files which use persistent predicates. Beforehand, a goal `use_module(library(persdb/persdb_rt))` must be issued. Furthermore, since persistent predicates defined by the loaded files are in this way defined dynamically, a call to `initialize_db/0` is commonly needed after loading and before calling predicates of these files.

83.8 Usage and interface

- **Library usage:**

  There are two packages which implement persistence: `persdb` and `'persdb/ll'` (for low level). In the first, the standard builtins `asserta_fact/1`, `assertz_fact/1`, and `retract_fact/1` are replaced by new versions which handle persistent data predicates, behaving as usual for normal data predicates. In the second package, predicates with names starting with `p` are defined, so that there is no overhead in calling the standard builtins. In any case, each package is used as usual: including it in the package list of the module, or using the `use_package/1` declaration.
- **Exports:**
  - *Predicates:*

    `passerta_fact/1`, `passertz_fact/1`, `pretract_fact/1`, `pretractall_fact/1`, `asserta_fact/1`, `assertz_fact/1`, `retract_fact/1`, `retractall_fact/1`, `initialize_db/0`, `make_persistent/2`, `update_files/0`, `update_files/1`, `create/2`.
  - *Regular Types:*

    `meta_predname/1`, `directoryname/1`.
  - *Multifiles:*

    `$is_persistent/2`, `persistent_dir/2`, `persistent_dir/4`.

83.9 Documentation on exports

passerta_fact/1: PREDICATE

Usage: `passerta_fact(Fact)`

Persistent version of `asserta_fact/1`: the current instance of `Fact` is interpreted as a fact (i.e., a relation tuple) and is added at the beginning of the definition of the corresponding predicate. The predicate concerned must be declared `persistent`. Any

uninstantiated variables in the `Fact` will be replaced by new, private variables. Defined in the `'persdb/ll'` package.

– *The following properties should hold at call time:*

`Fact` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `passerta_fact(fact)`.

passertz_fact/1: PREDICATE

Usage: `passertz_fact(Fact)`

Persistent version of `assertz_fact/1`: the current instance of `Fact` is interpreted as a fact (i.e., a relation tuple) and is added at the end of the definition of the corresponding predicate. The predicate concerned must be declared `persistent`. Any uninstantiated variables in the `Fact` will be replaced by new, private variables. Defined in the `'persdb/ll'` package.

– *The following properties should hold at call time:*

`Fact` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `passertz_fact(fact)`.

pretract_fact/1: PREDICATE

`pretract_fact(P)`

Retracts a predicate in both, the dynamic and the persistent databases.

Usage: `pretract_fact(Fact)`

Persistent version of `retract_fact/1`: deletes on backtracking all the facts which unify with `Fact`. The predicate concerned must be declared `persistent`. Defined in the `'persdb/ll'` package.

– *The following properties should hold at call time:*

`Fact` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `pretract_fact(fact)`.

pretractall_fact/1: PREDICATE

`pretractall_fact(P)`

Retracts all the instances of a predicate in both, the dynamic and the persistent databases.
Meta-predicate **with arguments:** `pretractall_fact(fact)`.

asserta_fact/1: PREDICATE

Usage: `asserta_fact(Fact)`

Same as `passerta_fact/1`, but if the predicate concerned is not persistent then behaves as the builtin of the same name. Defined in the `persdb` package.

– *The following properties should hold at call time:*

`Fact` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `asserta_fact(fact)`.

assertz_fact/1: PREDICATE

Usage: `assertz_fact(Fact)`

Same as `passertz_fact/1`, but if the predicate concerned is not persistent then behaves as the builtin of the same name. Defined in the `persdb` package.

- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `assertz_fact(fact)`.

retract_fact/1: PREDICATE

Usage: `retract_fact(Fact)`

Same as `pretract_fact/1`, but if the predicate concerned is not persistent then behaves as the builtin of the same name. Defined in the `persdb` package.

- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `retract_fact(fact)`.

retractall_fact/1: PREDICATE

Usage: `retractall_fact(Fact)`

Same as `pretractall_fact/1`, but if the predicate concerned is not persistent then behaves as the builtin of the same name. Defined in the `persdb` package.

- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `retractall_fact(fact)`.

initialize_db/0: PREDICATE

Usage:

Initializes the whole database, updating the state of the declared persistent predicates. Must be called explicitly after dynamically defining clauses for `persistent_dir/2`.

make_persistent/2: PREDICATE

Usage: `make_persistent(PredDesc,Keyword)`

Dynamic version of the `persistent` declaration.

- *The following properties should hold at call time:*

  `persdb_rt:meta_predname(PredDesc)` (`persdb_rt:meta_predname/1`)

  `Keyword` is an atom corresponding to a directory identifier. (`persdb_cache:keyword/1`)

Meta-predicate with arguments: `make_persistent(spec,?)`.

update_files/0: PREDICATE

Usage:

Updates the files comprising the persistence set of all persistent predicates defined in the application.

update_files/1: PREDICATE

Usage: `update_files(PredSpecList)`

Updates the files comprising the persistence set of the persistent predicates in `PredSpecList`**.**

- *Call and exit should be compatible with:*

  `PredSpecList` **is a list of** `prednames`**.** (`basic_props:list/2`)

Meta-predicate **with arguments:** `update_files(list(spec))`**.**

create/2: PREDICATE

No further documentation available for this predicate.

meta_predname/1: REGTYPE

A regular type, defined as follows:

```
meta_predname($:(P)) :-
    predname(P).
```

directoryname/1: REGTYPE

Usage: `directoryname(X)`

`X` **is an atom, the name of a directory.**

83.10 Documentation on multifiles

$is_persistent/2: PREDICATE

`$is_persistent(Spec,Key)`

Predicate `Spec` **persists within database** `Key`**. Programmers should not define this predicate directly in the program. The predicate is** *multifile***.**

The predicate is of type *data***.**

persistent_dir/2: PREDICATE

Usage: `persistent_dir(Keyword,Location_Path)`

Relates identifiers of locations (the `Keywords`**) with descriptions of such locations (**`Location_Paths`**).** `Location_Path` **is a directory and it means that the definition for the persistent predicates associated with** `Keyword` **is kept in files below that directory (which must previously exist). These files, in the updated state, contain the actual definition of the predicate in Prolog syntax (but with module names resolved).**

- *Call and exit should be compatible with:*

  `Keyword` **is an atom corresponding to a directory identifier.** (`persdb_cache:keyword/1`)

  `Location_Path` **is an atom, the name of a directory.** (`persdb_rt:directoryname/1`)

The predicate is *multifile*.

The predicate is of type *data*.

persistent_dir/4: PREDICATE

Usage: `persistent_dir(Keyword,Location_Path,DirPerms,FilePerms)`

The same as `persistent_dir/2`, but including also the permission modes for persistent directories and files.

- *Call and exit should be compatible with:*

  `Keyword` **is an atom corresponding to a directory identifier.** (`persdb_cache:keyword/1`)

  `Location_Path` **is an atom, the name of a directory.** (`persdb_rt:directoryname/1`)

  `DirPerms` **is an integer.** (`basic_props:int/1`)

  `FilePerms` **is an integer.** (`basic_props:int/1`)

The predicate is *multifile*.

The predicate is of type *data*.

83.11 Documentation on internals

persistent/2: DECLARATION

Usage: :- `persistent(PredDesc,Keyword).`

Declares the predicate `PredDesc` as persistent. `Keyword` is the identifier of a location where the persistent storage for the predicate is kept. The location `Keyword` is described in the `persistent_dir` predicate, which must contain a fact in which the first argument unifies with `Keyword`.

- *The following properties should hold upon exit:*

  `PredDesc` **is a predicate name.** (`basic_props:predname/1`)

  `Keyword` **is an atom corresponding to a directory identifier.** (`persdb_cache:keyword/1`)

keyword/1: PREDICATE

An atom which identifies a fact of the `persistent_dir/2` relation. This fact relates this atom to a directory in which the persistent storage for one or more persistent predicates is kept. Storage is expected under a subdirectory by the name of the module and in a file by the name of the predicate.

83.12 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`, `aggregates`, `terms_io`, `lists`, `system`, `file_locks`, `persdb_cache`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `nortchecks`, `datafacts`, `persdb/persdb_decl`.

84 Using the persdb library

Author(s): The Ciao Development Team.

Through the following examples we will try to illustrate the two mains ways of declaring and using persistent predicates: statically (the preferred method) and dynamically (necessary when the new persistent predicates have to be defined at run-time). The final example is a small application implementing a simple persistent queue.

84.1 An example of persistent predicates (static version)

```
:- module(example_static,[main/0],[persdb]).

:- use_module(library(aggregates)).
:- use_module(library(write)).
:- use_module(library(read)).

%% Declare the directory associated to the key "db" where the
%% persistence sets of the persistent predicates are stored:
persistent_dir(db,'./').

%% Declare a persistent predicate:
:- persistent(bar/1, db).

%% Read a term, storing it in a new fact of the persistent predicate
%% and list all the current facts of that predicate
main :-
    read(X),
    assertz_fact(bar(X)),
    findall(Y,bar(Y),L),
    write(L).

erase_one :-
    retract_fact(bar(_)).
erase_all :-
    retractall_fact(bar(_)).
```

84.2 An example of persistent predicates (dynamic version)

```
:- module(example_dynamic, [main/1], [persdb, dynamic]).

:- use_module(library(aggregates)).
:- use_module(library(write)).

:- impl_defined(bar/1).

main([X]):-
    % Declare the directory associated to the key "db"
    asserta_fact(persistent_dir(db,'./')),
    % Declare the predicate bar/1 as dynamic (and data) at run-time
    data(bar/1),
```

```
    % Declare the predicate bar/1 as persistent at run-time
    make_persistent(bar/1, db),
    assertz_fact(bar(X)),
    findall(Y, bar(Y), L),
    write(L).
```

84.3 A simple application / a persistent queue

```
:- module(queue, [main/0],[persdb]).

:- use_module(library(streams)).
:- use_module(library(read)).
:- use_module(library(write)).
:- use_module(library(aggregates)).

persistent_dir(queue_dir,'./pers').

:- persistent(queue/1, queue_dir).

queue(first).
queue(second).

main:-
     write('Action ( in(Term). | slip(Term) | out. | list. | halt. ): '),
     read(A),
     (  handle_action(A)
     -> main
     ;  write('Unknown command.'), nl, main ).

handle_action(end_of_file) :-
     halt.
handle_action(halt) :-
     halt.
handle_action(in(Term)) :-
     assertz_fact(queue(Term)).
handle_action(slip(Term)) :-
     asserta_fact(queue(Term)).
handle_action(out) :-
     (  retract_fact(queue(Term))
     -> write('Out '), write(Term)
     ;  write('FIFO empty.') ),
     nl.
handle_action(list) :-
     findall(Term,queue(Term),Terms),
     write('Contents: '), write(Terms), nl.
```

85 Manager for persistent data directories

Author(s): Jose F. Morales.

This module abstracts the location of directories for storing persistent data.

The directory for persistent data will be placed under `deploy_data_root_dir/1` (multifile) or inside the temporary build directory (see `config_common:data_root_dir/1`.

See `ensure_datadir/2` for usage. This module is handy to select the directory for `persdb` databases, e.g.:

```
persistent_dir(db, Dir) :- ensure_datadir('yourapp', Dir).
```

85.1 Usage and interface

- **Library usage:**
  `:- use_module(library(persdb/datadir)).`
- **Exports:**
  - *Predicates:*
    `ensure_datadir/2`.
  - *Multifiles:*
    `deploy_data_root_dir/1`.

85.2 Documentation on exports

ensure_datadir/2: PREDICATE

Usage: `ensure_datadir(RelPath,Path)`

Obtain the absolute path `Path` for storing persistent data under relative `RelPath` directory. The directory is created if needed.

- *Call and exit should be compatible with:*
  `RelPath` is an atom. (`basic_props:atm/1`)
  `Path` is an atom. (`basic_props:atm/1`)
- *The following properties should hold at call time:*
  `RelPath` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Path` is a free variable. (`term_typing:var/1`)

85.3 Documentation on multifiles

deploy_data_root_dir/1: PREDICATE

Directory for persistent data on deployment. If this is a relative path, it will be placed under the `HOME` directory automatically. The predicate is *multifile*.

85.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`, `pathnames`, `system`, `system_extra`, `config_common`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `basicmodes`, `fsyntax`, `datafacts`.

86 Filed predicates

Author(s): Francisco Bueno.

This package allows using files as a " cache" for predicates defined by facts. This is useful for huge tables of facts that may push the memory limits of the system too far. Goals of a filed predicate are executed simply by reading from the corresponding file.

Anything in the DB file used for the predicate that is different from a fact for the corresponding predicate is ignored. Each call to a filed predicate forces opening the file, so the use of this package is subject to the limit on the number of open files that the system can support.

Dynamic modification of the filed predicates is also allowed during execution of the program. Thus filed predicates are regarded as dynamic, data predicates residing in a file. However, dynamic modifications to the predicates do not affect the file, unless the predicate is also declared persistent.

The package is compatible with `persdb` in the sense that a predicate can be made both filed and persistent. In this way, the predicate can be used in programs, but it will not be loaded (saving memory), can also be modified during execution, and modifications will persist in the file. Thus, the user interface to both packages is the same (so the DB file must be one for both filing and persistency).

86.1 Usage and interface

- **Library usage:**

  This facility is used as a package, thus either including `factsdb` in the package list of the module, or by using the `use_package/1` declaration. The facility predicates are defined in library module `factsdb_rt`.
- **New operators defined:**

  `concurrent/1` [**1150,fx**], `data/1` [**1150,fx**].
- **Implicit imports:**
  - *System library modules:*

    `datafacts_rt`, `factsdb_rt`.
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `datafacts`.

86.2 Documentation on multifiles

$factsdb$cached_goal/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

87 Filed predicates (runtime)

Author(s): Francisco Bueno.

Runtime module for the `factsdb` package.

87.1 Usage and interface

- **Library usage:**
  `:- use_module(library(factsdb/factsdb_rt)).`
- **Exports:**
  - *Predicates:*
    `asserta_fact/1`, `assertz_fact/1`, `call/1`, `current_fact/1`, `retract_fact/1`.
  - *Multifiles:*
    `$factsdb$cached_goal/3`, `persistent_dir/2`, `file_alias/2`.

87.2 Documentation on exports

asserta_fact/1: PREDICATE

Usage: `asserta_fact(Fact)`

Version of `datafacts_rt:asserta_fact/1` for filed predicates. The current instance of `Fact` is interpreted as a fact and is added at the beginning of the definition of the corresponding predicate. Therefore, before all the facts filed in the DB file for the predicate. The predicate concerned must be declared as `facts`; if it is not, then `datafacts_rt:asserta_fact/1` is used.

- *The following properties should hold at call time:*
  `Fact` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `asserta_fact(fact)`.

assertz_fact/1: PREDICATE

Usage: `assertz_fact(Fact)`

Version of `datafacts_rt:assertz_fact/1` for filed predicates. The current instance of `Fact` is interpreted as a fact and is added at the end of the definition of the corresponding predicate. Therefore, after all the facts filed in the DB file for the predicate. The predicate concerned must be declared as `facts`; if it is not, then `datafacts_rt:assertz_fact/1` is used.

- *The following properties should hold at call time:*
  `Fact` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `assertz_fact(fact)`.

call/1: PREDICATE

Usage: `call(Fact)`

Same as `current_fact/1` if the predicate concerned is declared as `facts`. If it is not, an exception is raised.

- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `call(fact)`.

current_fact/1: PREDICATE

Usage: `current_fact(Fact)`

Version of `datafacts_rt:current_fact/1` for filed predicates. The current instance of `Fact` is interpreted as a fact and is unified with an actual fact in the current definition of the corresponding predicate. Therefore, with a fact previously asserted or filed in the DB file for the predicate, if it has not been retracted. The predicate concerned must be declared as `facts`; if it is not, then `datafacts_rt:current_fact/1` is used.

- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `current_fact(fact)`.

retract_fact/1: PREDICATE

Usage: `retract_fact(Fact)`

Version of `datafacts_rt:retract_fact/1` for filed predicates. The current instance of `Fact` is interpreted as a fact and is unified with an actual fact in the current definition of the corresponding predicate; such a fact is deleted from the predicate definition. This is true even for the facts filed in the DB file for the predicate; but these are NOT deleted from the file (unless the predicate is persistent). The predicate concerned must be declared as `facts`; if it is not, then `datafacts_rt:retract_fact/1` is used.

- *The following properties should hold at call time:*

  `Fact` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `retract_fact(fact)`.

87.3 Documentation on multifiles

$factsdb$cached_goal/3: PREDICATE

`$factsdb$cached_goal(Spec,Spec,Key)`

Predicate `Spec` is filed within database `Key`. Programmers should not define this predicate directly in the program. The predicate is *multifile*.

persistent_dir/2: PREDICATE

See `persdb`. The predicate is *multifile*.

The predicate is of type *data*.

file_alias/2: PREDICATE

See `symfnames`. This predicate is used only if `persistent_dir/2` fails. The predicate is *multifile*.

The predicate is of type *data*.

87.4 Documentation on internals

facts/2: DECLARATION

Usage: :- `facts(PredDesc,Keyword)`.

Declares the predicate `PredDesc` as filed. `Keyword` is the identifier of a location where the file DB for the predicate is kept. The location `Keyword` is described in the `file_alias` predicate, which must contain a fact in which the first argument unifies with `Keyword`.

– *The following properties should hold upon exit:*

  `PredDesc` is a predicate name. (`basic_props:predname/1`)

  `Keyword` is an atom corresponding to a directory identifier. (`persdb_cache:keyword/1`)

keyword/1: PREDICATE

See `persdb_rt`. The same conventions for location of DB files apply in both packages.

87.5 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

  `datafacts_rt`, `counters`, `read`, `persdb_cache`.

– *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `datafacts`.

88 Low-level concurrency/multithreading primitives

Author(s): Manuel Carro.

This module provides basic mechanisms for using concurrency and implementing multi-goal applications. It provides a means for arbitrary goals to be specified to be run in a separate stack set; in that case, they are assigned a goal identifier with which further accesses (e.g., asking for more solutions) to the goal can be made. Additionally, in some architectures, these goals can be assigned an O.S. thread, separate from the one which made the initial call, thus providing concurrency and, in multiprocessors, parallelism capabilities.

88.1 Usage and interface

- **Library usage:**
  `:- use_module(library(concurrency)).`
- **Exports:**
  - *Predicates:*
    `eng_call/4`, `eng_call/3`, `eng_backtrack/2`, `eng_cut/1`, `eng_release/1`, `eng_wait/1`, `eng_kill/1`, `eng_killothers/0`, `eng_status/0`, `eng_goal_id/1`, `lock_atom/1`, `unlock_atom/1`, `atom_lock_state/2`, `concurrent/1`.

88.2 Documentation on exports

eng_call/4: PREDICATE

Usage: `eng_call(Goal,EngineCreation,ThreadCreation,GoalId)`

Calls `Goal` in a new engine (stack set), possibly using a new thread, and returns a `GoalId` to designate this new goal henceforth. `EngineCreation` can be either `wait` or `create`; the distinction is not yet meaningful. `ThreadCreation` can be one of `self`, `wait`, or `create`. In the first case the creating thread is used to execute `Goal`, and thus it has to wait until its first result or failure. The call will fail if `Goal` fails, and succeed otherwise. However, the call will always suceed when a remote thread is started. The space and identifiers reclaimed for the thread must be explicitly deallocated by calling `eng_release/1`. `GoalIds` are unique in each execution of a Ciao Prolog program.

- *The following properties should hold at call time:*
  - `Goal` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `EngineCreation` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `ThreadCreation` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `GoalId` **is a free variable.** (`term_typing:var/1`)
  - `Goal` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
  - `EngineCreation` **is an atom.** (`basic_props:atm/1`)
  - `ThreadCreation` **is an atom.** (`basic_props:atm/1`)
  - `GoalId` **is an integer.** (`basic_props:int/1`)

Meta-predicate **with arguments:** `eng_call(goal,?,?,?)`.

eng_call/3: PREDICATE

Usage: `eng_call(Goal,EngineCreation,ThreadCreation)`

Similar to `eng_call/4`, but the thread (if created) and stack areas are automatically released upon success or failure of the goal. No `GoalId` is provided for further interaction with the goal.

– *The following properties should hold at call time:*

`Goal` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`EngineCreation` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`ThreadCreation` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

`EngineCreation` is an atom. (`basic_props:atm/1`)

`ThreadCreation` is an atom. (`basic_props:atm/1`)

Meta-predicate with arguments: `eng_call(goal,?,?)`.

eng_backtrack/2: PREDICATE

Usage: `eng_backtrack(GoalId,ThreadCreation)`

Performs backtracking on the goal designed by `GoalId`. A new thread can be used to perform backtracking, according to `ThreadCreation` (same as in `eng_call/4`). Fails if the goal is backtracked over by the local thread, and there are no more solutions. Always succeeds if executed by a remote thread. The engine is not automatically released up upon failure: `eng_release/1` must be called to that end.

– *The following properties should hold at call time:*

`GoalId` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`ThreadCreation` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`GoalId` is an integer. (`basic_props:int/1`)

`ThreadCreation` is an atom. (`basic_props:atm/1`)

eng_cut/1: PREDICATE

Usage: `eng_cut(GoalId)`

Performs a *cut* in the execution of the goal `GoalId`. The next call to `eng_backtrack/2` will therefore backtrack all the way and fail.

– *The following properties should hold at call time:*

`GoalId` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`GoalId` is an integer. (`basic_props:int/1`)

eng_release/1: PREDICATE

Usage: `eng_release(GoalId)`

Cleans up and releases the engine executing the goal designed by `GoalId`. The engine must be idle, i.e., currently not executing any goal. `eng_wait/1` can be used to ensure this.

– *The following properties should hold at call time:*

`GoalId` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`GoalId` is an integer. (`basic_props:int/1`)

eng_wait/1: PREDICATE

Usage: `eng_wait(GoalId)`

Waits for the engine executing the goal denoted by `GoalId` to finish the computation (i.e., it has finished searching for a solution, either with success or failure).

– *The following properties should hold at call time:*

`GoalId` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`GoalId` is an integer. (`basic_props:int/1`)

eng_kill/1: PREDICATE

Usage: `eng_kill(GoalId)`

Kills the thread executing `GoalId` (if any), and frees the memory used up by the stack set. Usually one should wait (`eng_wait/1`) for a goal, and then release it, but killing the thread explicitly allows recovering from error states. A goal cannot kill itself. This feature should be used with caution, because there are situations where killing a thread might render the system in an unstable state. Threads should cooperate in their killing, but if the killed thread is blocked in a I/O operation, or inside an internal critical region, this cooperation is not possible and the system, although stopped, might very well end up in a incosistent state.

– *The following properties should hold at call time:*

`GoalId` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`GoalId` is an integer. (`basic_props:int/1`)

eng_killothers/0: PREDICATE

Usage:

Kills threads and releases stack sets of all active goals, but the one calling `eng_killothers`. Again, a safety measure. The same cautions as with `eng_kill/1` should be taken.

eng_status/0: PREDICATE

Usage:

Prints to standard output the current status of the stack sets.

eng_goal_id/1: PREDICATE

Usage: `eng_goal_id(GoalId)`

`GoalId` is unified with the identifier of the goal within which `eng_goal_id/1` is executed.

– *The following properties should hold at call time:*

`GoalId` is an integer. (`basic_props:int/1`)

lock_atom/1: PREDICATE

Usage 1: `lock_atom(Atom)`

The semaphore associated to `Atom` is accessed; if its value is nonzero, it is atomically decremented and the execution of this thread proceeds. Otherwise, the goal waits until a nonzero value is reached. The semaphore is then atomically decremented and the execution of this thread proceeds.

– *The following properties should hold at call time:*

`Atom` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`Atom` is an integer. (`basic_props:int/1`)

Usage 2:

– *The following properties should hold at call time:*

`Arg1` is an integer. (`basic_props:int/1`)

unlock_atom/1: PREDICATE

Usage 1: `unlock_atom(Atom)`

The semaphore associated to `Atom` is atomically incremented.

– *The following properties should hold at call time:*

`Atom` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`Atom` is an integer. (`basic_props:int/1`)

Usage 2:

– *The following properties should hold at call time:*

`Arg1` is an integer. (`basic_props:int/1`)

atom_lock_state/2: PREDICATE

Usage 1: `atom_lock_state(Atom,Value)`

Sets the semaphore associated to `Atom` to `Value`. This is usually done at the beginning of the execution, but can be executed at any time. If not called, semaphore associated to atoms are by default inited to 1. It should be used with caution: arbitrary use can transform programs using locks in a mess of internal relations. The change of a semaphore value in a place other than the initialization stage of a program is not among the allowed operations as defined by Dijkstra [Dij65, BA82].

– *The following properties should hold at call time:*

`Atom` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`Value` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`Atom` is an integer. (`basic_props:int/1`)

`Value` is an integer. (`basic_props:int/1`)

Usage 2: `atom_lock_state(Atom,Value)`

Consults the `Value` of the semaphore associated to `Atom`. Use sparingly and mainly as a medium to check state correctness. Not among the operations on semaphore by Djikstra.

– *The following properties should hold at call time:*

`Atom` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`Value` is a free variable. (`term_typing:var/1`)

`Atom` is an atom. (`basic_props:atm/1`)

`Value` is an integer. (`basic_props:int/1`)

Usage 3:

– *The following properties should hold at call time:*

`Arg1` is an integer. (`basic_props:int/1`)

`Arg2` is an integer. (`basic_props:int/1`)

concurrent/1: PREDICATE

`concurrent(PredName)`

The predicate named `PredName` is made concurrent in the current module at runtime (useful for predicate names generated on-the-fly). This difficults a better compile-time analysis, but in turn offers more flexibility to applications. It is also faster for some applications: if several agents have to share data in a stuctured fashion (e.g., the generator knows and wants to restrict the data generated to a set of other threads), a possibility is to use the same concurrent fact and emply a field within the fact to distinguish the receiver/sender. This can cause many threads to access and wait on the same fact, which in turns can create contention problems. It is much better to create a new concurrent fact and to use that new name as a channel to communicate the different threads. `concurrent/1` can either be given a predicate spec in the form `Name/Arity`, with `Name` and `Arity` bound, or to give a value only to `Arity`, and let the system choose a new, unused `Name` for the fact.

Usage:

– *The following properties should hold at call time:*

`PredName` is a predicate name. (`basic_props:predname/1`)

88.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`datafacts_rt`.

– *Packages:*

`noprelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `datafacts`.

89 Aggregates (concurrency-safe)

Author(s): Manuel Carro (concurrency-safeness).

This module implements thread-safe aggregation predicates. Its use and results should be the same as those in the aggregates library, but several goals can use them concurrently without the interference and wrong results (due to implementation reasons) aggregates might lead to. This particular implementation is completely based on the one used in the aggregates library (whose original authors were Richard A. O'Keefe and David H.D. Warren).

89.1 Usage and interface

- **Library usage:**
  `:- use_module(library(conc_aggregates)).`
- **Exports:**
  - *Predicates:*
    `findall/3`, `setof/3`, `bagof/3`.

89.2 Documentation on exports

findall/3: PREDICATE

Usage: `findall(Template,Generator,List)` ⟨• ISO •⟩

A special case of bagof, where all free variables in the `Generator` are taken to be existentially quantified. Safe in concurrent applications.

- *The following properties should hold at call time:*
  `Generator` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
- *The following properties should hold upon exit:*
  `Template` is any term. (`basic_props:term/1`)
  `Goal` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `Set` is a list. (`basic_props:list/1`)
- *The following properties should hold globally:*
  All calls of the form `findall(Template,Generator,List)` are deterministic. (`native_props:is_det/1`)

Meta-predicate with arguments: `findall(?,goal,?)`.

setof/3: PREDICATE

Usage: `setof(Template,Goal,Set)` ⟨• ISO •⟩

Finds the `Set` of instances of the `Template` satisfying the `Generator`. The set is in ascending order (see `compare/3` for a definition of this order) without duplicates, and is non-empty. If there are no solutions, `setof/3` fails. `setof/3` may succeed in more than one way, binding free variables in the `Generator` to different values. This can be avoided by using existential quantifiers on the free variables in front of the `Generator`, using `^/2`. E.g., in `A^p(A,B)`, `A` is existentially quantified. Safe in concurrent apllications.

- *The following properties should hold at call time:*

  `Goal` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties should hold upon exit:*

  `Template` **is any term.** (`basic_props:term/1`)

  `Goal` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

  `Set` **is a list.** (`basic_props:list/1`)
- *The following properties should hold globally:*

  `Template` **is not further instantiated.** (`basic_props:not_further_inst/2`)

bagof/3: PREDICATE

Usage: `bagof(Template,Generator,Bag)` ⟨• ISO •⟩

Finds all the instances of the `Template` produced by the `Generator`, and returns them in the `Bag` in the order in which they were found. If the `Generator` contains free variables which are not bound in the `Template`, it assumes that this is like any other Prolog question and that you want bindings for those variables. This can be avoided by using existential quantifiers on the free variables in front of the `Generator`, using `^/2`. Safe in concurrent applications.

- *The following properties should hold at call time:*

  `Generator` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties should hold upon exit:*

  `Template` **is any term.** (`basic_props:term/1`)

  `Goal` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

  `Set` **is a list.** (`basic_props:list/1`)
- *The following properties should hold globally:*

  `Template` **is not further instantiated.** (`basic_props:not_further_inst/2`)

89.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `native_props`, `datafacts_rt`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `nativeprops`, `datafacts`.

90 Active modules

Author(s): Manuel Hermenegildo, Daniel Cabeza (original version before 1.16), Jose F. Morales (revised version).

Stability: [devel] Currently the subject of active development and/or research. Functionality may be limited and API and/or functionality may change without warning or deprecation period. Not recommended yet for use in production.

This version corresponds to a revised model for active modules. See Ciao version 1.15 for the original first design as described in [CH95].

An *active module* is an ordinary module whose instances (copies sharing the same code but different state or data) have computational resources attached (e.g., computation steps shared in a fair fashion).

Active modules provide a high-level model of concurrency suitable for distributed execution (and inter-process communication) that is similar to *active object*s and *actor*s. Note that Ciao also offers lower-level primitives for concurrency and network-based communication.

90.1 Concurrency model and semantics

Each *active module instance* is internally composed of a local mailbox for queries (a queue of messages) and a query handler loop. Each active module instance is identified by a unique name (which can be provided or created automatically).

Calls to exported predicates of an active module are enqueued in the mailbox. The query handler loop is a deterministic loop that executes queries sequentially, sending back the results to the caller program (another active module instance) if needed. The composition of the (possibly) multiple answers from the callee and the caller is given be the *query protocols* defined below.

Query requests at the handler loop do not fail or leave choicepoints. When required, non-deterministic behaviour must be captured on the answer and treated on the callee by the *query protocols*.

90.1.1 Query protocols

There exist several query protocols depending on the expected answers of a predicate.

- ***all solutions*: all answers to each query are precomputed on the callee side and sent to the caller. Backtracking is supported by enumeration on the caller side. Note: calls to active module predicates with an infinite number of solutions will obviously not terminate with this query method.**
- ***cast*: no answer is required from the callee (equivalent to *message passing* in distributed computation). It is useful when the query performs side-effects or it sends back the answers to the caller active module through another cast (a-la continuation-based programming).**
- **(experimental) *answers with suspensions*: the callee may return a suspended computation and continue the execution on the caller site. Currently predicates must be declared as `suspendable`. The main focus of the current funcrionality is the implementation of `REST`ful applications via the (experimental) HTTP interface. Note: support is limited, recommended only for deterministic computations.**

Further details on the semantics and concurrency model:

- The cost of calls depends on the size of the messages (arguments, results, and the target *location*)
- Deadlocks may happen due to the "message processing lock" (e.g., A calls B, B calls A). Use *cast* instead.

- Query protocols will be changed or extended in the future, specially to optimize cost for particular cases.
- Suspendable predicates rely on the experimental `fibers` package. This may change in the future.
- Or-suspensions for lazily asking for more solutions are not currently implemented (they are in development).

90.1.2 Side-effects

All communication between active module instances should (in principle) happen through message passing. Instances should not share any global data. Sharing via dynamic/data predicates (or other global mechanisms) is seen as an *impure* side-effect w.r.t. this model and must be used with care (e.g., *caching*, hand-made optimizations, etc.).

90.1.3 Distributed

In a distributed setting active module instances may run on separate *node*s that can interchange messages through the network. See `actmod_dist` for more details about the distribution protocol and how it can be extended.

90.2 Using active modules

Using active modules requires the use of the `actmod` package:

```
:- module(...,...,[actmod]).
```

This turns the current module into an active module and enables all the directives and features required to use other active modules.

Predicates exported by an active module can be accessed by other active modules using the `use_module/3` declaration with the `active` option (see below).

Note that the process of *using* an active module does not involve transferring any code, but rather setting up things so that calls in the module using the active module are executed as remote procedure calls to the active module.

90.3 Running active modules

For spawning active module instances (dynamic creation) see `actmod_process`.

Active modules may implement a `main/1` predicate, if they want to receive command-line arguments or use the directive `:- dist_node` to include a default `main/1` for distributed nodes. See `actmod_dist` for more details.

90.4 Examples

The following command:

```
ciaoc simple_server.pl
```

compiles the simple server example that comes with the distribution (in the `actmod/example` directory). The `simple_client_with_main` example (in the same directory) can be compiled as usual:

```
ciaoc simple_client_with_main
```

Now, if the server is running when the client is executed it will connect with the server to access the predicate(s) that it imports from it.

An even simpler client `simple_client.pl` can be loaded into the top level and its predicates called as usual (and they will connect with the server if it is running).

90.5 Usage and interface

- **Library usage:**
  `:- use_package(actmod).`
  or
  `:- module(...,...,[actmod]).`
- **Exports:**
  - *Regular Types:*
    `import_opt/1`.
- **New declarations defined:**
  `use_module/3`.
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

90.6 Documentation on new declarations

use_module/3: DECLARATION

Usage: :- `use_module(ModSpec,Imports,Opts)`.

Import from `ModSpec` the predicates in `Imports` with options `Opts`. If `Imports` is a free variable, all predicates are imported.

- *The following properties should hold at call time:*
  `ModSpec` is a source name. (`stream_basic:sourcename/1`)
  `Imports` is a list of `prednames`. (`basic_props:list/2`)
  `Opts` is a list of `import_opts`. (`basic_props:list/2`)

90.7 Documentation on exports

import_opt/1: REGTYPE

`import_opt(Opt)`

Options for `use_module/3`:

- `active`: **import as an active module (which adds a dependency to the module interface, not its code; it allows `actmod_spawn/3` and static named instances).**
- `reg_protocol(RegProtocol)`: **specify default name server protocol.**
- `libexec`: **use `libexec` as spawning option by default.**
- `binexec(Name)`: **use `binexec(Name)` as spawning option by default.**

91 Distribution protocol for active modules

Author(s): The Ciao Development Team, Jose F. Morales.

This is the code that handles setting up nodes in a distributed environment, and sending and receiving messages (see mailbox) between active module instances.

> The current distribution protocol provide no means for security. The application must take care of this.

This includes:

- Low-level TCP/IP socket connection between nodes
- Publishing/locating ActRef names in a registry
- Sending/receiving messages

A node is an OS process (currently identified by its Hostname:Port). Each node may contain many active module instances. The distributed protocol requires a mapping between the active module instance identifiers and the precise location of the node on the network (its address).

This mapping is maintained by a *registry* of addresses of active module instances. We allow differerent registry protocols to publish and locate these addresses (see ⟨undefined⟩ [Registry protocols], page ⟨undefined⟩).

The current implementation for communication between active modules is based on TCP/IP sockets (thus, the address of an active module instance is an IP socket address in a particular machine).

Requests to execute goals in a module are sent as messages through the mailbox. When the mailbox corresponds to a remote active module instance, the messages are send over TCP/IP socket connection between the local and remote nodes. Similary, response messages that encode the answers are retrieved through sockets into local mailboxes.

91.1 Distributed nodes

When using the `dist_node` directive, the executable serves as a for distributed active modules. It accepts following command-line options:

```
-p Port                    Port number for listening connection
-a Address                 Address for listening connection
--reg-protocol RegProtocol Registry protocol
--reg-addr Address Port    Registry address (optional)
--reg-dir Path             Registry directory (optional)
--name Name                Publish name for the (default) active module instance
```

Some of them can be controlled through the following environment variables:

```
ACTMOD_HOSTNAME        Address for listening connection
```

Other declarations:

- `:- dist_start(G)`: specify a default starting goal to be executed once the node is ready to accept connections.
- `:- actmod_reg_protocol(<RegProtocol>)`: default registry protocol used in the context of the current active module

91.2 Registry protocols

The implementation currently uses the experimental `traits` package (similar to `multifile` predicates but with better control on the hooks interface).

A registry protocol implements the predicates to publish and locate addresses of active module instances. By convention, the implementation of a registry protocol `RP` is encoded in a module with name `regp_<RP>`.

The *publish* part of the protocol must implement the `actmod_publish` trait, defining the `save_addr/5` predicate. This predicate publishes the current active module address.

The *locate* part of the protocol must implement the `actmod_locate` trait, defining the `remote_address/3` predicate. This predicate locates the address of the specified active module.

For efficiency, the locate methods maintain a cache of addresses, so that the server information only needs to be read from the file system the first time the active module is accessed.

The available implementations are:

- `regp_filebased` **(default)**
- `regp_platformbased`
- `regp_webbased`

91.3 Usage and interface

- **Library usage:**
  `:- use_module(library(actmod/actmod_dist)).`
- **Exports:**
  - *Predicates:*
    `dist_init/3`, `dist_init_args/3`, `get_addr_stream/2`, `dist_send/2`, `dist_get_reg_protocol/1`, `dist_set_reg_protocol/1`, `actI_alloc_named/2`, `chn_actref/2`, `actchn_send/2`, `actref_to_actchn/2`, `actchn_watch_response/2`, `actchn_unwatch_response/1`, `term_to_atom/2`, `atom_to_term/2`.
  - *Regular Types:*
    `actref/1`, `actchn/1`.
  - *Multifiles:*
    `$curr_mod/1`, `fnct.decl_at_mod/2`, `fnct.prop/2`, `$fnct_stub_rename/2`, `$handle_stream/2`, `$current_msg/3`, `gsusp.guard/4`, `gsusp.run/2`, `async.ftypes/2`, `async.run/2`, `$fiber_susp_hook/2`, `transient.decl/1`, `actmod_publish.save_addr/6`, `actmod_locate.remote_address/4`, `actmod_locate.cleanup_actI/2`, `$dist_addr_retry/1`, `$dmod_reg_protocol/2`, `$dmod_src/2`, `$dmod_prop/2`, `$local_actmod/1`, `$static_named_actRef/2`, `$dmod_proxy/2`, `$actmod_exe/3`, `qprot.collect/3`, `qprot.dec/4`, `qprot.enc/3`, `qprot.prepare_query/4`, `qprot.join_answers/7`.

91.4 Documentation on exports

dist_init/3: PREDICATE

Usage: `dist_init(RegProtocol,Mod,Opts)`

Initialize a node for distributed active modules (registry protocol `RegProtocol`, initial module `Mod`, options `Opts`).

dist_init_args/3: PREDICATE

Usage: `dist_init_args(RegProtocol,Mod,Args)`

Like `dist_init/3` but parses options from command-line arguments `Args`.

get_addr_stream/2: PREDICATE

No further documentation available for this predicate.

dist_send/2: PREDICATE

No further documentation available for this predicate.

dist_get_reg_protocol/1: PREDICATE

No further documentation available for this predicate.

dist_set_reg_protocol/1: PREDICATE

No further documentation available for this predicate.

actI_alloc_named/2: PREDICATE

No further documentation available for this predicate.

actref/1: REGTYPE

Usage: `actref(X)`

Reference to an active module instance

actchn/1: REGTYPE

Usage: `actchn(X)`

Channel for active module instance communication

chn_actref/2: PREDICATE

No further documentation available for this predicate.

actchn_send/2: PREDICATE

No further documentation available for this predicate.

actref_to_actchn/2: PREDICATE
No further documentation available for this predicate.

actchn_watch_response/2: PREDICATE
No further documentation available for this predicate.

actchn_unwatch_response/1: PREDICATE
No further documentation available for this predicate.

term_to_atom/2: PREDICATE
No further documentation available for this predicate.

atom_to_term/2: PREDICATE
No further documentation available for this predicate.

91.5 Documentation on multifiles

$curr_mod/1: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

fnct.decl_at_mod/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

fnct.prop/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$fnct_stub_rename/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$handle_stream/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$current_msg/3: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

gsusp.guard/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

gsusp.run/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

async.ftypes/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

async.run/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$fiber_susp_hook/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

transient.decl/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_publish.save_addr/6: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_locate.remote_address/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_locate.cleanup_actI/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dist_addr_retry/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_reg_protocol/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_src/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_prop/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$local_actmod/1: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$static_named_actRef/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$dmod_proxy/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$actmod_exe/3: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.collect/3: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.dec/4: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.enc/3: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.prepare_query/4: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.join_answers/7: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

91.6 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`, `system`, `lists`, `actmod_rt`, `stream_watchdog`, `sockets`, `sockets_io`, `format_to_string`, `read_from_string`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `fsyntax`, `datafacts`, `traits`.

92 The “filebased” registry protocol

The `filebased` saves the IP address and socket number of the server in a <*module_name*>`.addr` file at the address file directory or in the directory that a `.addr` file, if it exists, specifies.

The address file directory can be selected with the `set_reg_dir/1` predicate at `filebased_common` or the `--reg-dir` option for server startup.

By default, this directory is set to the temporary directory (see `get_tmp_dir/1` at `system`).

Note that servers and clients can be started in different machines, provided this directory is shared (e.g., by NFS or Samba), or the file can be moved to an appropriate directory on a different machine –provided the full path is the same.

92.1 Usage and interface

- **Library usage:**
  `:- use_module(library(actmod/regp_filebased)).`

92.2 Documentation on multifiles

actmod_publish.save_addr/6: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_locate.remote_address/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_locate.cleanup_actI/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dist_addr_retry/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_reg_protocol/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_src/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_prop/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$local_actmod/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$static_named_actRef/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_proxy/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$actmod_exe/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.collect/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.dec/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.enc/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.prepare_query/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.join_answers/7: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

92.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `system`, `system_extra`, `filebased_common`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `traits`.

93 The “platformbased” registry protocol

The `platformbased` registry is based on another active module working as a name server. It accepts queries for publishing and locating other active module instances. The address of this registry active moduel is given by the `--reg-addr` option.

93.0.1 Example name server

See `examples/nameserver/nameserver.pl` for an example implementation of a name server. In this example the addresses of active modules are saved by the name server in a persistent database (subdirectory `actmod_db` of the directory where you start it). This allows to restart the server right away if it dies (since it saves its state). This directory should be cleaned up regularly of addresses of active modules which are no more active. To do this, stop the server —by killing it (its pid is in `PATH/FILE`), and restart it after cleaning up the files in the above mentioned directory.

The name server has to be compiled as an active module itself:

```
ciaoc nameserver
```

It has to be started in the server machine before the application and its active modules are compiled.

93.1 Usage and interface

- **Library usage:**
  `:- use_module(library(actmod/regp_platformbased)).`

93.2 Documentation on multifiles

$curr_mod/1: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

fnct.decl_at_mod/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

fnct.prop/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$fnct_stub_rename/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$handle_stream/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$current_msg/3: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

gsusp.guard/4: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

gsusp.run/2: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

async.ftypes/2: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

async.run/2: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

$fiber_susp_hook/2: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

transient.decl/1: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

actmod_publish.save_addr/6: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

actmod_locate.remote_address/4: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

actmod_locate.cleanup_actI/2: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

$dist_addr_retry/1: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

$dmod_reg_protocol/2: PREDICATE

No further documentation available for this predicate. The predicate is ***multifile*****.**

$dmod_src/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_prop/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$local_actmod/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$static_named_actRef/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_proxy/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$actmod_exe/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.collect/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.dec/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.enc/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.prepare_query/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.join_answers/7: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

93.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `fibers_rt`, `actmod_rt`, `regp_platformserver`, `lists`, `regp_filebased`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `actmod`, `fibers`, `traits`, `assertions`, `assertions/assertions_basic`.

94 The "webbased" registry protocol

The `webbased` registry is also based on another active module working as a name server. The address of the registry must be specified in an application file named `webbased_common.pl` (see section below).

An application using a name server for active modules must have a file named `webbased_common.pl` that specifies where the name server resides. It must have the `URL` and the path which corresponds to that `URL` in the file system of the server machine (the one that hosts the `URL`) of the file that will hold the name server address.

The current distribution provides a file `webbased_common.pl` that can be used (after proper setting of its contents) for a server of active modules for a whole installation. Alternatively, particular servers for each application can be set up (see below).

The example implementation `examples/nameserver/nameserver.pl` can also be used for this registry protocol.

To set up a name server edit `webbased_common.pl` to change its contents appropriately as described above (`URL` and corresponding complete file path). Then recompile this library module:

```
ciaoc -c webbased_common
```

94.1 Usage and interface

- **Library usage:**
  `:- use_module(library(actmod/regp_webbased)).`

94.2 Documentation on multifiles

$curr_mod/1: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

fnct.decl_at_mod/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

fnct.prop/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$fnct_stub_rename/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$handle_stream/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$current_msg/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

gsusp.guard/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

gsusp.run/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

async.ftypes/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

async.run/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$fiber_susp_hook/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

transient.decl/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_publish.save_addr/6: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_locate.remote_address/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_locate.cleanup_actI/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dist_addr_retry/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_reg_protocol/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_src/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$dmod_prop/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$local_actmod/1: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$static_named_actRef/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$dmod_proxy/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$actmod_exe/3: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.collect/3: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.dec/4: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.enc/3: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.prepare_query/4: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.join_answers/7: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

94.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `fibers_rt`, `actmod_rt`, `regp_webserver`, `regp_filebased`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `actmod`, `fibers`, `traits`, `assertions`, `assertions/assertions_basic`.

95 Active module processes

This module provides predicates to create and handle active module instances. Note that each instance is associated with its own computational resources.

Active module instance run on their own fibers (optionally in a new process) for an `DMod`.

95.1 Usage and interface

- **Library usage:**
  `:- use_module(library(actmod/actmod_process)).`
- **Exports:**
  - *Predicates:*
    `actmod_spawn/3`, `actmod_kill/2`, `actmod_terminate/2`, `actmod_join/1`, `actmod_compile_all/0`, `actmod_compile/1`, `actmod_check_bin/1`, `actmod_load_dynmod/1`.
  - *Regular Types:*
    `actref/1`, `spawn_opt/1`.
  - *Multifiles:*
    `actmod_publish.save_addr/6`, `actmod_locate.remote_address/4`, `actmod_locate.cleanup_actI/2`, `$dist_addr_retry/1`, `$dmod_reg_protocol/2`, `$dmod_src/2`, `$dmod_prop/2`, `$local_actmod/1`, `$static_named_actRef/2`, `$dmod_proxy/2`, `$actmod_exe/3`, `qprot.collect/3`, `qprot.dec/4`, `qprot.enc/3`, `qprot.prepare_query/4`, `qprot.join_answers/7`.

95.2 Documentation on exports

actref/1: REGTYPE

Usage: `actref(X)`

`X` is the reference of an active module instance

spawn_opt/1: REGTYPE

`spawn_opt(X)`

Options to control the location of the fiber:

- `child`: run on a new child OS process
- `daemon`: run on a new daemon (`https://en.wikipedia.org/wiki/Daemon_(computing)`) OS process (background processes that are their own session and group leaders, and that are dissociated from a controlling terminal)
- (default): run on this OS process

Options to specify the active module implementation:

- `exec(ExecPath,ExecArgs)`: path for executable and arguments (containing the distributed runtime)
- `dynmod(Mod)`: module spec (for spawning using a generic `ciaosh` runtime)
- `libexec`: like `exec/1`, using binary created by `actmod_compile/1` at `build/libexec`

- `binexec(Name)`: use the binary `Name` created at `build/bin` (passing `--actmod` argument at process startup)

Usage: `spawn_opt(X)`

`X` is an spawning option

actmod_spawn/3: PREDICATE

Usage: `actmod_spawn(DMod,Opts,ActRef)`

Create a new active module instance of `DMod`. `Opts` is a list of options and `ActRef` the active module instance reference.

- *The following properties should hold at call time:*

`DMod` is currently a term which is not a free variable.	(`term_typing:nonvar/1`)
`Opts` is currently a term which is not a free variable.	(`term_typing:nonvar/1`)
`ActRef` is a free variable.	(`term_typing:var/1`)
`DMod` is an atom.	(`basic_props:atm/1`)
`Opts` is a list of `spawn_opt`s.	(`basic_props:list/2`)
`ActRef` is the reference of an active module instance	(`actmod_process:actref/1`)

actmod_kill/2: PREDICATE

Usage: `actmod_kill(ActRef,Msg)`

Sends SIGKILL signal to the active module instance `ActRef`. If not registered as a child process, it is considered to be a daemon. Return message in `Msg`.

actmod_terminate/2: PREDICATE

Usage: `actmod_terminate(ActRef,Msg)`

Sends SIGTERM signal to the active module instance `ActRef`. If not registered as a child process, it is considered to be a daemon. Return message in `Msg`.

actmod_join/1: PREDICATE

Usage: `actmod_join(ActRef)`

Block until the active module instance `ActRef` terminates

actmod_compile_all/0: PREDICATE

Usage:

Compile all imported active modules (only for `libexec`)

actmod_compile/1: PREDICATE

Usage: `actmod_compile(DMod)`

Ensure that the (imported) active module `DMod` is compiled (only for `libexec`)

actmod_check_bin/1: PREDICATE

Usage: `actmod_check_bin(DMod)`

Check a binary for the (imported) active module `DMod` exists (only for `libexec`)

actmod_load_dynmod/1: PREDICATE

No further documentation available for this predicate.

95.3 Documentation on multifiles

actmod_publish.save_addr/6: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_locate.remote_address/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_locate.cleanup_actI/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dist_addr_retry/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_reg_protocol/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_src/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_prop/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$local_actmod/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$static_named_actRef/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_proxy/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$actmod_exe/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.collect/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.dec/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.enc/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.prepare_query/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.join_answers/7: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

95.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `datafacts_rt`, `system`, `system_extra`, `process`, `actmod_rt`, `actmod_dist`, `pathnames`, `lists`, `config_common`, `ciaoc_aux`, `actmod_holder`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `fsyntax`, `hiord`, `isomodes`, `regtypes`, `datafacts`, `traits`.

96 actmod_rt (library)

96.1 Usage and interface

- **Library usage:**
  `:- use_module(library(actmod/actmod_rt)).`
- **Exports:**
  - *Predicates:*
    `actref_send/2`, `$actmod_start_main/1`, `$actmod_start_nohalt/1`, `named_actRef/2`, `singleton_actRef/2`, `actI_init_named/2`, `actI_send_cast/3`, `actI_send_call/3`, `actI_receive_response/2`, `actmod_call/1`, `actmod_cast/1`, `$actmod_call/1`, `get_actI/2`, `force_set_actref/1`, `actmod_get_self/1`, `actmod_get_self_mod/1`, `dist_log/1`.
  - *Multifiles:*
    `actmod_publish.save_addr/6`, `actmod_locate.remote_address/4`, `actmod_locate.cleanup_actI/2`, `$dist_addr_retry/1`, `$dmod_reg_protocol/2`, `$dmod_src/2`, `$dmod_prop/2`, `$local_actmod/1`, `$static_named_actRef/2`, `$dmod_proxy/2`, `$actmod_exe/3`, `qprot.collect/3`, `qprot.dec/4`, `qprot.enc/3`, `qprot.prepare_query/4`, `qprot.join_answers/7`, `$curr_mod/1`, `fnct.decl_at_mod/2`, `fnct.prop/2`, `$fnct_stub_rename/2`, `$handle_stream/2`, `$current_msg/3`, `gsusp.guard/4`, `gsusp.run/2`, `async.ftypes/2`, `async.run/2`, `$fiber_susp_hook/2`, `transient.decl/1`.

96.2 Documentation on exports

actref_send/2: PREDICATE
No further documentation available for this predicate.

$actmod_start_main/1: PREDICATE
No further documentation available for this predicate.

$actmod_start_nohalt/1: PREDICATE
No further documentation available for this predicate.

named_actRef/2: PREDICATE
No further documentation available for this predicate.

singleton_actRef/2: PREDICATE
No further documentation available for this predicate.

actI_init_named/2: PREDICATE
No further documentation available for this predicate.

actI_send_cast/3: PREDICATE
No further documentation available for this predicate.

actI_send_call/3: PREDICATE
No further documentation available for this predicate.

actI_receive_response/2: PREDICATE
No further documentation available for this predicate.

actmod_call/1: PREDICATE
No further documentation available for this predicate.

actmod_cast/1: PREDICATE
No further documentation available for this predicate.

$actmod_call/1: PREDICATE
No further documentation available for this predicate.

get_actI/2: PREDICATE
No further documentation available for this predicate.

force_set_actref/1: PREDICATE
No further documentation available for this predicate.

actmod_get_self/1: PREDICATE
No further documentation available for this predicate.

actmod_get_self_mod/1: PREDICATE
No further documentation available for this predicate.

dist_log/1: PREDICATE
No further documentation available for this predicate.

96.3 Documentation on multifiles

actmod_publish.save_addr/6: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_locate.remote_address/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

actmod_locate.cleanup_actI/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dist_addr_retry/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_reg_protocol/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_src/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_prop/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$local_actmod/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$static_named_actRef/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$dmod_proxy/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$actmod_exe/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

qprot.collect/3: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.dec/4: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.enc/3: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.prepare_query/4: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

qprot.join_answers/7: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$curr_mod/1: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

fnct.decl_at_mod/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

fnct.prop/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$fnct_stub_rename/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$handle_stream/2: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

$current_msg/3: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

gsusp.guard/4: PREDICATE
No further documentation available for this predicate. The predicate is *multifile*.

gsusp.run/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

async.ftypes/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

async.run/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

$fiber_susp_hook/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

transient.decl/1: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

96.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `datafacts_rt`, `fibers_rt`, `actmod_dist`, `lists`, `fibers_data`, `aggregates`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `hiord`, `fsyntax`, `datafacts`, `traits`.

97 Constraint programming over rationals

Author(s): Christian Holzbaur, Daniel Cabeza, Samir Genaim (Meta-programming predicates).

Warning: This package is currently being adapted to the new characteristics of the Ciao module system. This new version works right now with limitations, but it is under further development at the moment. Use with (lots of) caution.

97.1 Usage and interface

- **Library usage:**

  `:- use_package(clpq).`

  or

  `:- module(...,...,[clpq]).`
- **New operators defined:**

  `.=./2` [**700,xfx**], `.<>./2` [**700,xfx**], `.<./2` [**700,xfx**], `.=<./2` [**700,xfx**], `.>./2` [**700,xfx**], `.>=./2` [**700,xfx**].
- **Implicit imports:**
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

97.2 Other information

97.2.1 Some CLP(Q) examples

(Other examples can be found in the source and library directories.)

- **'Reversible' Fibonacci (clpq):**

```
:- module(fib_q, [fib/2], []).
:- use_package(clpq).

fib(X,Y):- X .=. 0, Y .=. 0.
fib(X,Y):- X .=. 1, Y .=. 1.
fib(N,F) :-
    N .>. 1,
    N1 .=. N - 1,
    N2 .=. N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F .=. F1+F2.
```

- **Matrix multiplication (clpq):**

```
:- use_package(clpq).

mmultiply([],_,[]).
```

```
mmultiply([V0|Rest], V1, [Result|Others]):-
        mmultiply(Rest, V1, Others),
        multiply(V1,V0,Result).

multiply([],_,[]).
multiply([V0|Rest], V1, [Result|Others]):-
        multiply(Rest, V1, Others),
        vmul(V0,V1,Result).

vmul([],[],0).
vmul([H1|T1], [H2|T2], Result):-
    vmul(T1,T2, Newresult),
    Result .=. H1*H2+Newresult.

matrix(1,[[1,2,3,4,5],[4,0,-1,5,6],[7,1,-2,8,9],[-1,0,1,3,2],[1,5,-3,2,4]]).
matrix(2,[[3,2,1,0,-1],[-2,1,3,0,2],[1,2,0,-1,5],[1,3,2,4,5],[-5,1,4,2,2]]).

%% Call with: ?- go(M).

go(M):-
    matrix(1,M1),
    matrix(2,M2),
    mmultiply(M1, M, M2).
```

- **Queens (clpq):**

```
:- use_package(clpq).

:- use_module(library(lists), [member/2]).

queens(N, Qs) :- constrain_values(N, N, Qs), place_queens(N, Qs).

constrain_values(0, _N, []).
constrain_values(N, Range, [X|Xs]) :-
    N .>. 0, X .>. 0, X .=<. Range,
    N1 .=. N - 1,
    constrain_values(N1, Range, Xs), no_attack(Xs, X, 1).

no_attack([], _Queen, _Nb).
no_attack([Y|Ys], Queen, Nb) :-
    Queen .<>. Y+Nb,
    Queen .<>. Y-Nb,
    Nb1 .=. Nb + 1,
    no_attack(Ys, Queen, Nb1).

place_queens(0, _).
place_queens(N, Q) :-
    N > 0, member(N, Q), N1 is N-1, place_queens(N1, Q).
```

97.2.2 Meta-programming with CLP(Q)

The implementation of CLP(Q) in Ciao compiles the constraints in the program to a sequence of calls to the underlying constraints solver (at compile-time). This results in efficient implementation, since the structure of the constraints is processed only at compile-time, but requires the constraints to be known at static time which can be a limitation for metaprogramming-based applications such as static program analyzers. For example, the call:

```
?- X=(A+B), Y=(C-D), X .>. Y.

no
```

fails because `X .>. Y` is translated first to a sequence of calls that require (when they invoked) `X` and `Y` to be either numbers or free variables. To overcome this limitation, you can use `clpq_meta/1` which delays the translation of the constraints from compile-time to run-time (i.e., when `clpq_meta/1` is called), For example:

```
?- X=(A+B),Y=(C-D), clpq_meta([X .>. Y]).

X = A+B,
Y = C-D,
C.<.D+A+B ?
```

The argument of `clpq_meta/1` accepts a goal or lists of goals, where each goal is limited to conjunctions, disjunctions, or CLP(Q) constraints. Other operations on constraints which are extensively used in meta-programming, in particular in static program analysis, are *projection* and *entailment check*. The projection operation restricts the constraints (that are available in the store) to a given set of variables and turns the answer into terms. You can use the multifile predicate `dump_constraints/3` for that purpose:

```
?- A .>. C, C .>. B, dump_constraints([A,B],[X,Y],Cs).

Cs = [X.>.Y],
C.>.B,
C.<.A ?

?- C=(B+D), clpq_meta([A .>. C, D .>. 0]), dump_constraints([A,B],[X,Y],Cs).

C = B+D,
Cs = [Y.<.X],
D.<. -B+A,
D.>.0 ?
```

The *entailment check* is used to check if a list of constrains is entailed by the store. You can use the predicate `clpq_entailed/1` for that purpose:

```
?- A .>. C, C .>. B, B .>. D, clpq_entailed([ A .>. B, A .>. D]).

B.>.D,
C.>.B,
C.<.A ?

yes

?- A .>=. B, clpq_entailed([ A .>. B ]).
```

```
no
```

98 Constraint programming over reals

Author(s): Christian Holzbaur, Daniel Cabeza, Samir Genaim (Meta-programming predicates).

Warning: This package is currently being adapted to the new characteristics of the Ciao module system. This new version now works right now to some extent, but it under further development at the moment. Use with (lots of) caution.

98.1 Usage and interface

- **Library usage:**
  `:- use_package(clpr).`
  or
  `:- module(...,...,[clpr]).`
- **New operators defined:**
  `.=./2` [**700,xfx**], `.<>./2` [**700,xfx**], `.<./2` [**700,xfx**], `.=<./2` [**700,xfx**], `.>./2` [**700,xfx**], `.>=./2` [**700,xfx**].
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

98.2 Other information

98.2.1 Some CLP(R) examples

(Other examples can be found in the source and library directories.)

- **'Reversible' Fibonacci (clpr):**

```
:- module(_, [fib/2], []).
:- use_package(clpr).

fib(X,Y):- X .=. 0, Y .=. 0.
fib(X,Y):- X .=. 1, Y .=. 1.
fib(N,F) :-
    N .>. 1,
    N1 .=. N - 1,
    N2 .=. N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F .=. F1+F2.
```

- **Dirichlet problem for Laplace's equation (clpr):**

```
%
```

```
% Solve the Dirichlet problem for Laplace's equation using
% Leibman's five-point finit-differenc approximation.
% The goal ?- go1 is a normal example, while the goal ?- go2
% shows output constraints for a small region where the boundary conditions
% are not specified.
%
:- use_package(clpq).
:- use_module(library(format)).

laplace([_, _]).
laplace([H1, H2, H3|T]):-
    laplace_vec(H1, H2, H3),
    laplace([H2, H3|T]).

laplace_vec([_, _], [_, _], [_, _]).
laplace_vec([_TL, T, TR|T1], [ML, M, MR|T2], [_BL, B, BR|T3]):-
    B + T + ML + MR - 4 * M .=. 0,
    laplace_vec([T, TR|T1], [M, MR|T2], [B, BR|T3]).

printmat([]).
printmat([H|T]):-
    printvec(H),
    printmat(T).

printvec([]):- nl.
printvec([H|T]):-
    printrat(H),
    printvec(T).

printrat(rat(N,D)) :- !,
    X is N/D,
    format(" ~2f",X).
printrat(N) :-
    X is N*100,
    format(" ~2d",X).

go1:-
    X =  [
      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, _, _, _, _, _, _, _, _, _, 100],
    [100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100]
    ],
    laplace(X),
```

```
    printmat(X).

% Answer:
% 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
% 100.00 51.11 32.52 24.56 21.11 20.12 21.11 24.56 32.52 51.11 100.00
% 100.00 71.91 54.41 44.63 39.74 38.26 39.74 44.63 54.41 71.91 100.00
% 100.00 82.12 68.59 59.80 54.97 53.44 54.97 59.80 68.59 82.12 100.00
% 100.00 87.97 78.03 71.00 66.90 65.56 66.90 71.00 78.03 87.97 100.00
% 100.00 91.71 84.58 79.28 76.07 75.00 76.07 79.28 84.58 91.71 100.00
% 100.00 94.30 89.29 85.47 83.10 82.30 83.10 85.47 89.29 94.30 100.00
% 100.00 96.20 92.82 90.20 88.56 88.00 88.56 90.20 92.82 96.20 100.00
% 100.00 97.67 95.59 93.96 92.93 92.58 92.93 93.96 95.59 97.67 100.00
% 100.00 98.89 97.90 97.12 96.63 96.46 96.63 97.12 97.90 98.89 100.00
% 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0

go2([B31, M32, M33, B34, B42, B43, B12, B13, B21, M22, M23, B24]) :-
    laplace([
    [_B11, B12, B13, _B14],
    [B21, M22, M23, B24],
    [B31, M32, M33, B34],
    [_B41, B42, B43, _B44]
    ]).


% Answer:
%
% B34.=. -4*M22+B12+B21+4*M33-B43,
% M23.=.4*M22-M32-B12-B21,
% B31.=. -M22+4*M32-M33-B42,
% B24.=.15*M22-4*M32-4*B12-4*B21-M33-B13 ?
```

98.2.2 Meta-programming with CLP(R)

see ⟨undefined⟩ [Meta-programming with CLP(Q)], page ⟨undefined⟩

99 Constraint programming over finite domains (new)

Author(s): Emilio Jesús Gallego Arias, Rémy Haemmerlé, Jose F. Morales.

This package extends Ciao with constraints over finite domains (FD). The solver is an instance of the Constraint Logic Programming (CLP) scheme as introduced by Jaffar and Lassez [JL87]. It uses classical propagation techniques as described in Van Hentenryck's book [Van89] and Diaz's clp(FD) implementation [CD96].

The package provides predicates for checking consistency of FD constraints. A FD is a small subset of integers, and FD constraints are relations over integer. Hence only integer or variables are allowed in such constraints. FD variables (i.e., variables that occur in an FD constraint) get associated with a domain either explicitly declared by the program or implicitly imposed by the solver. As soon as variables get an empty domain the computation fails, hence forcing backtracking.

The package defines basic operators, automatically imports basic constraints and enumerating predicates form the module `clpfd_rt`, and provides high-level ⟨undefined⟩ [Meta-Constraints], page ⟨undefined⟩ through a transparent compilation process.

99.1 Completeness Considerations

For efficiency reason, the solver is not complete on non-ground constraints, in the sense that it may not be able to determine that a set of constraints is actually satisfiable. In such cases, the system silently succeeds. To ensure full completeness, the programmer may use the `labeling/2` predicate that uses an automatic backtracking search to find ground solutions for a list of FD variables. Labeling is complete, always terminates, and yields no redundant solutions. See an example of use of labeling in the following ⟨undefined⟩ [Example], page ⟨undefined⟩.

On success, the top-level will display the domain associated with each FD variable remaining free in the query. This domain should not be understood as values permitted for the corresponding variable, but only as values not excluded by the incomplete propagation mechanism of the solver. Note that the answer output by the top-level is by itself incomplete as the remaining constraints are not showed.

99.2 Meta-Constraints

There are five meta-constraints, namely `#=/2` (constraint equal), `#\=/2` (constraint not equal), `#</2` (constraint less than), `#=</2` (constraint less or equal), `#>/2` (constraint greater than), and `#>=/2` (constraint greater or equal). These meta-constraints are defined over arithmetic expressions with FD variables (see regular type `fd_expr/1` in module `clpfd_rt`.). Such constraints are "meta" in the sens that their arguments are interpreted at compile-time and all variables occurring free in the arguments will be implicitly constrained to take integer values only. In particular, note that variables constrained in such a way would not be unifiable with complex FD expressions. For instance, the call:

```
X + Y #> Z.
```

is not equivalent to the call:

```
A = X + Y, A #> Z.
```

While the first call succeeds, the second one will throw an exception to indicate that `A` cannot be unified with the non-integer term `X + Y`. It is possible to view meta-constraints as a convenient way to define an infinite number of FD constraints. For instance `A #> Z` and `X + Y #> Z` can be considered respectively as binary and ternary constraints over FD variables.

It is possible to delay interpretation of meta-constraints at call time by explicitly prefixing the call with `clpfd_rt`. For instance, the following call will not throw any exception:

```
A = X + Y, clfd_rt:(A #> Z).
```

99.3 Example

The problem is to put N queens on an NxN chessboard so that there is no pair of queens threatening each other. Each variable is a queen. Each queen has a designated row. The problem is to find a different column for each one.

The main constraint of the problem is that no queen threaten another. This is encoded by the `diff/3` predicate and should hold for any pair of queens.

The main call is `queens(N, L, Lab)` which looks for a solution `L` for the `N` queens problem using labeling `Lab`. Observe the call to `labeling/2` at the end of definition of `queens/3`, which tries to find a solution for the problem.

99.4 Usage and interface

- **Library usage:**

  `:- use_package(clpfd).`

  or

  `:- module(...,...,[clpfd]).`
- **New operators defined:**

  `in/2` [700,xfx], `#=/2` [700,xfx], `#\=/2` [700,xfx], `#</2` [700,xfx], `#=</2` [700,xfx], `#>/2` [700,xfx], `#>=/2` [700,xfx], `../2` [550,xfx].
- **Implicit imports:**
  - *System library modules:*

    `clpfd_rt`.
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

100 Finite domain solver runtime

Author(s): Emilio Jesús Gallego Arias, Rémy Haemmerlé.

This module provides Finite Domain (FD) constraints and enumerating predicates for FD variables. See `clpfd_doc` for more details about the Ciao FD solver.

This module provides two kinds of contraints: *basic constraints* **(such as `domain/3`, `all_different/1`, ...) that deal with FD variables or lists of FD variables, and** *meta-contraints* **(such as `#=/2`, `#=</2`, ...) that deal with arithmetic expressions over FD variables, called in the following FD expressions. Meta-constraints of this module behave as described in package `clpfd`, except that their arguments are interpreted at run time.**

100.1 Usage and interface

- **Library usage:**
  `:- use_module(library(clpfd/clpfd_rt)).`
- **Exports:**
  - *Predicates:*
    `in/2`, `#=/2`, `#\=/2`, `#</2`, `#=</2`, `#>/2`, `#>=/2`, `domain/3`, `in/2`, `all_different/1`, `labeling/2`, `indomain/1`, `label/1`, `labeling/2`, `minimize/2`, `maximize/2`, `wrapper/2`.
  - *Regular Types:*
    `fdvar/1`, `fd_range_expr/1`, `fd_expr/1`.
  - *Multifiles:*
    `attr_rt:unify_hook/3`, `attr_rt:attribute_goals/4`.

100.2 Documentation on exports

in/2: PREDICATE

Usage: `in(Var,Range)`

Constrains `Var` to take its value in the domain described by `Range`.

- *The following properties should hold at call time:*
  `Var` is a variable or an integer. (`clpfd_rt:fdvar/1`)
  `Range` is a range expression. (`clpfd_rt:fd_range_expr/1`)

fdvar/1: REGTYPE

Usage: `fdvar(X)`

`X` is a variable or an integer.

fd_range_expr/1: REGTYPE

A term denoting a range expression:

```
fd_range_expr(I) :-
    integer(I).
fd_range_expr(..(Min,Max)) :-
    integer(Min),
    integer(Max).
fd_range_expr(A\/B) :-
    fd_range_expr(A),
    fd_range_expr(B).
```

an integer stands for a singleton range, `Min..Max` for the closed interval from `Min` to `Max`, and `A\/B` for the union of ranges `A` and `B`. Range expressions are used by the `in/2` predciate.

Usage: `fd_range_expr(Expr)`

`Expr` **is a range expression.**

fd_expr/1: REGTYPE

A term denoting an arithmetic expression over FD variables:

```
fd_expr(Var) :-
    var(Var).
fd_expr(I*Exp) :-
    int(I),
    fd_expr(Exp).
fd_expr(Exp*I) :-
    int(I),
    fd_expr(Exp).
fd_expr(Exp1+Exp2) :-
    fd_expr(Exp1),
    fd_expr(Exp2).
fd_expr(Exp1-Exp2) :-
    fd_expr(Exp1),
    fd_expr(Exp2).
fd_expr(-Exp) :-
    fd_expr(Exp).
```

FD expressions are used by meta-constraints.

Usage: `fd_expr(Expr)`

`Expr` **is an FD expression.**

#=/2: PREDICATE

Meta-constraint "equal".

Usage: `#=(A,B)`

Constrains the interpretation of `A` to be equal to the interpretation of `B`.

– *The following properties should hold at call time:*

`A` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

`B` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

#\=/2: PREDICATE

Meta-constraint "not equal".

Usage: `#\=(A,B)`

Constrains the interpretation of `A` to be different from the interpretation of `B`

– *The following properties should hold at call time:*

`A` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

`B` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

#</2: PREDICATE

Meta-constraint "smaller than".

Usage: `#<(A,B)`

Constrains the interpretation of `A` to be smaller than the interpretation of `B`.

– *The following properties should hold at call time:*

`A` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

`B` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

#=</2: PREDICATE

Meta-constraint "smaller or equal".

Usage: `#=<(A,B)`

Constrains `A` to be smaller or equal to `B`.

– *The following properties should hold at call time:*

`A` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

`B` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

#>/2: PREDICATE

Meta-constraint "greater than".

Usage: `#>(A,B)`

Constrains the interpretation of `A` to be greater than the interpretation of `B`.

– *The following properties should hold at call time:*

`A` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

`B` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

#>=/2: PREDICATE

Meta-constraint "greater or equal".

Usage: `#>=(A,B)`

Constrains the interpretation of `A` to be greater or equal than the interpretation of `B`.

– *The following properties should hold at call time:*

`A` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

`B` **is an FD expression.** (`clpfd_rt:fd_expr/1`)

domain/3: PREDICATE

Usage: `domain(Vars,Min,Max)`

Constrains each element of `Vars` to take its value between `Min` and `Max` (included). This predicate is generally used to set the initial domain of an interval

– *The following properties should hold at call time:*

`Vars` is a list of `fdvars`. (`basic_props:list/2`)

`Min` is an integer. (`basic_props:int/1`)

`Max` is an integer. (`basic_props:int/1`)

in/2: PREDICATE

Usage: `in(Var,Range)`

Constrains `Var` to take its value in the domain described by `Range`.

– *The following properties should hold at call time:*

`Var` is a variable or an integer. (`clpfd_rt:fdvar/1`)

`Range` is a range expression. (`clpfd_rt:fd_range_expr/1`)

all_different/1: PREDICATE

Usage: `all_different(Vars)`

Constrains all elements in `Vars` to take distinct values. This is equivalent to posting an inequality constraint for each pair of variables. This constraint is triggered when a variable becomes ground, removing its value from the domain of the other variables.

– *The following properties should hold at call time:*

`Vars` is a list of `fdvars`. (`basic_props:list/2`)

labeling/2: PREDICATE

Usage: `labeling(Options,Vars)`

Assigns a value to each variable in `Vars` according to the labeling options given by `Options`. This predicate is re-executable on backtracking.

The different options are :

- `[]`: the leftmost variables is selected first. Its values are enumerating from the smallest to the greatest.
- `[ff]`: the variable with the smallest number of elements in its domain is selected first. Its values are then enumerated from the smallest to the greatest.
- `[step]`: the variable with the smallest number of elements in its domain is selected first. The minimal value of the domainis assigned, on bactracking the value is pruned form the domain and a new variable is selected.

– *The following properties should hold at call time:*

`Vars` is a list of `fdvars`. (`basic_props:list/2`)

indomain/1: PREDICATE

No further documentation available for this predicate.

label/1: PREDICATE

No further documentation available for this predicate.

labeling/2: PREDICATE

Usage: `labeling(Options,Vars)`

Assigns a value to each variable in `Vars` according to the labeling options given by `Options`. This predicate is re-executable on backtracking.

The different options are :

- `[]`: **the leftmost variables is selected first. Its values are enumerating from the smallest to the greatest.**
- `[ff]`: **the variable with the smallest number of elements in its domain is selected first. Its values are then enumerated from the smallest to the greatest.**
- `[step]`: **the variable with the smallest number of elements in its domain is selected first. The minimal value of the domainis assigned, on bactracking the value is pruned form the domain and a new variable is selected.**

– *The following properties should hold at call time:*

`Vars` **is a list of** `fdvars`. (`basic_props:list/2`)

minimize/2: PREDICATE

No further documentation available for this predicate. *Meta-predicate* **with arguments:** `minimize(goal,?)`.

maximize/2: PREDICATE

No further documentation available for this predicate. *Meta-predicate* **with arguments:** `maximize(goal,?)`.

wrapper/2: PREDICATE

No further documentation available for this predicate.

100.3 Documentation on multifiles

attr_rt:unify_hook/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

attr_rt:attribute_goals/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

100.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  clpfd_debug_rt, fd_term, fd_constraints, fd_labeling, fd_optim, attr_rt.
- *Packages:*
  prelude, initial, condcomp, assertions, assertions/assertions_basic, regtypes, fsyntax, dcg, clpfd/clpfd_debug, clpfd/clpfd_options, attr.

101 Tabling execution

Author(s): Pablo Chico de Guzmán Huerta, Joaquín Arias, The Ciao Development Team.

Stability: [beta] Most of the functionality is there but it is still missing some testing and/or verification.

This library package allows the evaluation of predicates using tabled resolution (*tabling*). Tabling is an alternative execution strategy for logic programs that records calls and their answers in order to reuse them in future calls. It improves Prolog declarativity and can improve efficiency by avoiding repeated computations [TS86, War92]. Tabling is guaranteed to terminate when Prolog programs have the *bounded term-depth property*. Some examples of the use of tabling can be found at the end of this section.

101.1 Tabling

Adding a `table/1` declaration to a predicate makes the compiler and run-time system distinguish the first occurrence of a tabled goal (the *generator*) and subsequent calls which are identical up to variable renaming (the *consumers*). The generator applies resolution using the program clauses to derive answers for the goal. Consumers *suspend* the current execution path (using implementation-dependent means) and move to a different branch. When such an alternative branch finally succeeds, the answer generated for the initial query is inserted in a table associated with the original goal. This makes it possible to reactivate suspended calls and to continue execution at the point where it was stopped. Thus, consumers do not recompute those calls, but obtain instead the answers from the table where they have been previously inserted by the producer.

Predicates not marked as tabled are executed following standard SLD resolution, hopefully with (minimal or no) overhead due to the availability of tabling in the system.

Our current version of tabling only supports *local* evaluation and *variant tabling* [**RRR96**]. Local evaluation computes all the answers of a tabled predicate before returning any of them. Note that a call to a tabled predicate will never return if this call has infinite answers. Variant tabling considers two calls/answers to be the same only when they are identical under variable renaming.

By declaring a predicate as tabled, a program translation is performed to abstract the use of internal tabling primitives for the user. Our tabling implementation technique follows the $Theta$ -CHAT approach [DS99] which does not require major changes in the compiler or run-time system.

101.2 Tabled Constraint Logic Programming

The TCLP implementation allows the combination of tabling with constraints. The initial implementation described in [dGCHS12] has been modified by a modular implementation, called Mod TCLP, which is described in [AC19a]. By using the Mod TCLP interface of a constraint solver, e.g., `:- use_package(t_clpq)`, the tabling engine uses the entailment check provided by the constraint solver to detect more particular calls / answers.

The TCLP interface of a constraint solver must implement the following interface:

```
call_domain_projection/2
answer_domain_projection/2
call_store_projection/3
answer_store_projection/3
call_entail/2
answer_check_entail/3
apply_answer/2
```

as an API for the tablingq engine. Some examples of TCLP interfaces are `different_constraints`, `t_clpq`, and `t_clpr` libraries.

Some examples of the use of TCLP can be found at the end of this section (with some examples of the TCLP interface).

101.3 Some TCLP applications

The current implementation of Mod TCLP has been used to develop more complex applications:

- The Implementation of an Abstract Interpretation Algorithm described in [AC19b]:

  Abstract interpretation requires a fixpoint computation. PLAI is a fixpoint algorithm implemented by the abstract interpreter of CiaoPP, an analyzer and optimizer suite for logic programs, part of the Ciao development environment. In this paper, we adapt the existing PLAI implementation in CiaoPP using tabled constraint logic programming. The tabling engine is used to compute the fixpoint and the constraint engine computes the LUB of the abstract substitutions of different clauses. That provides, on one hand, much simpler code since the fixpoint computation is taken care of by the underlying tabling machinery, and, in most cases, performance gains, since some crucial operations (such as branch switching and resumption) are executed by the tabling engine. Determining that the fixpoint has been reached uses semantic equivalence, e.g., whether syntactically different representations of an abstract substitution actually refer to the same element in the abstract domain. This is delegated to the abstract domain operations, transparently to the analyzer. As a result, the tabling analyzer can reuse answers in more cases than if syntactical equality were used to detect repeated calls, and better performance, even taking into account the additional cost associated to these checks, is achieved. The implementation presented is based on the TCLP framework available in Ciao Prolog. It is one-third the size of the initial fixpoint implementation we started with, and its performance has been evaluated by analysing several programs with different abstract domains.

- Incremental Evaluation of Lattice-Based Aggregates in Logic Programming described in [AC19c] and available as a bundle, `:- use_package(tclp_aggregates)`, in the current distribution of Ciao:

  Aggregates are used to compute single pieces of information from separate data items, such as records in a database or answers to a query to a logic program. The maximum and minimum are well-known examples of aggregates. The computation of aggregates in Prolog or variant-based tabling can loop even if the aggregate at hand can be finitely determined. When answer subsumption or mode-directed tabling is used, termination improves, but the behavior observed in existing proposals is not consistent. We present a framework to incrementally compute aggregates for elements in a lattice. We use the entailment and join relations of the lattice to define (and compute) aggregates and decide whether some atom is compatible with (entails) the aggregate. The semantics of the aggregates defined in this way is consistent with the LFP semantics of tabling with constraints. Our implementation is based on the TCLP framework available in Ciao Prolog, and improves its termination properties w.r.t. similar approaches. Defining aggregates that do not fit into the lattice structure is possible, but some properties guaranteed by the lattice may not hold. However, the flexibility provided by this possibility justifies its inclusion. We validate our design with several examples and we evaluate their performance.

101.4 Usage and interface

- **Library usage:**

  The `TABLED_EXECUTION` flag must be set to `yes` during system configuration in order to compile the engine with support for the execution of goals with tabling. It is set to `yes` by default.
- **New declarations defined:**

  `table/1`.
- **Implicit imports:**
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

101.5 Documentation on new declarations

table/1: DECLARATION

It declares a tabled predicate.

101.6 Other information

101.6.1 Some examples using Tabling

We now illustrate some of the uses of the package of tabling through examples. The following example defines a simple predicte `path(X,Y)` which returns the transitive closuer of `edge/2` without entering loops:

```
:- use_package(tabling).
:- table path/2.

path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).

edge(a,b).
edge(b,a).
```

Other examples can be found in the source and library directories and in [dG12].

101.6.2 Some examples using Tabled Constraint Logic Programming

We now illustrate some of the uses of the package of t_clpq through examples. The following example defines a simple predicte `fibonacci(N,F)` which returns the fibonacci number `F` given the index `N` and returns the index `N` given the fibonacci number `F`:

```
:- use_package(tabling).
:- use_package(t_clpq).
:- table fibonacci/2.

fibonacci(0, 0).
fibonacci(1, 1).
fibonacci(N, F) :-
```

```
    N .>=. 2,
    N1 .=. N - 1,
    N2 .=. N - 2,
    F1 .>=. 0,
    F2 .>=. 0,
    F .=. F1 + F2,
    fibonacci(N1, F1),
    fibonacci(N2, F2).
```

Other examples can be found in the source and library directories and in [AC19a].

101.6.3 Some examples of TCLP interface

We now illustrate the implementation of a TCLP interface to link a constraint solver wiht the tabling engine through examples. The following example defines the interface of `clpq` **wiht the tabling engine:**

```
:- use_package(clpq).
:- use_module(library(clpq/clpq_dump), [clpqr_dump_constraints/3]).
:- active tclp.

call_domain_projection(Vars, st(Vars,_)).
call_entail(st(Vars,_), st(FGen-ProjGen)) :-
       Vars = FGen, clpq_entailed(ProjGen).
call_store_projection(_, st(Vars,_), st(F,Proj)) :-
       clpqr_dump_constraints(Vars, F , Proj).

answer_domain_projection(Vars, st(F,Proj)) :-
       clpqr_dump_constraints(Vars, F, Proj).
answer_check_entail(st(F,_), st(FAns,ProjAns), 1) :-
       F = FAns, clpq_entailed(ProjAns), !.
answer_check_entail(st(F,Proj), st(FAns,ProjAns), -1) :-
       F = FAns, clpq_meta(ProjAns), clpq_entailed(Proj).
answer_store_projection(_, St, St).

apply_answer(Vars, st(FAns,ProjAns)) :-
       Vars = FAns, clpq_meta(ProjAns).
```

Other examples can be found in the source and library directories and in [AC19a].

102 Attributed variables

Author(s): Rémy Haemmerlé, Christian Holzbaur, Daniel Cabeza, Manuel Carro.

This package implements attributed variables in the style of Holzbaur [Hol90]. It provides a way to associate to variables one or several arbitrary terms called attributes. By allowing the user to redefine the unification of attributed variables, this extension makes possible the design of coroutining facilities (see subsection ⟨undefined⟩ [Example], page ⟨undefined⟩) and clean interfaces between Prolog and constraints solvers.

Attributes are private to module and each variable can have at most one attribute in each module. Attributes are handled by predicate provided by `attr_rt` module.

`attr` package imports automatically attributes variables manipulation predicates, `put_attr_local/2`, `get_attr_local/2`, and `del_attr_local/2` from module `'attr/attr_rt'`, and set up the following hooks:

- `attr_unify_hook(AttValue, VarValue).`

  Hook that must be defined in the module using package `attr`. It is called after the attributed variable of that module has been unified with a non-var term, possibly another attributed variable. `AttValue` is the attribute that was associated to the variable in this module and `VarValue` is the new value of the variable. Normally this predicate fails to veto binding the variable to `VarValue`, forcing backtracking to undo the binding. If `VarValue` is another attributed variable the hook often combines the two attribute and associates the combined attribute with `VarValue` using `attr_rt:put_attr_local/2`.

- `attribute_goal(Var, S0, S).`

  This optional hook, if it is defined, is used by `attr_rt:copy_term/3` to project attributes of that module to residual goals, and by the toplevel to obtain residual goals after executing a query. The predicate is supposed to unified `S0` with a different list containing the residual goals and which have `S` as tail. For the sake of simplicity, it can be defined using `dcg`. (See DCG non-terminal `attribute_goal(Var)` in example below.)

- `attr_portray_hook(Attribute, Var).`

  Called by `write_term/2` for each attribute associate to the variable `Var` if the option option attributes(portray) is in effect. If the hook succeeds the variable is considered to be printed otherwise `Module =...` is printed to indicate the existence of an attribute defined in module `Module`. At call time `Attribute` is the actual value of the attribute associate to the variables `Var`.

102.1 Example

In the following example we give an implementation of `freeze/2`. We name it `myfreeze/2` in order to avoid a name clash with the built-in predicate of the same name. The code is available in the module `library(attr/example/myfreeze)`

```
:- module(myfreeze, [myfreeze/2], [attr, dcg, hiord]).

:- use_module(engine(io_basic)).

:- meta_predicate(myfreeze(?, goal)).

myfreeze(X, Goal) :-
    ( nonvar(X) ->
        call(Goal)
    ; get_attr_local(X, Fb) ->
        meta_conj(Fb, Goal, C),
```

```
        put_attr_local(X, C)   % rescue conjunction
    ; put_attr_local(X, Goal)
    ).

:- if(defined(optim_comp)).
attr_unify_hook(Fa, Other) :-
    ( nonvar(Other) ->
        '$trust_metatype'(Fa, goal),
        call(Fa)
    ; get_attr_local(Other, Fb) ->
        meta_conj(Fa, Fb, C),
        put_attr_local(Other, C)  % rescue conjunction
    ; put_attr_local(Other, Fa)   % rescue conjunction
    ).
:- else.
attr_unify_hook(Fa, Other) :-
    ( nonvar(Other) ->
        call(Fa)
    ; get_attr_local(Other, Fb) ->
        meta_conj(Fa, Fb, C),
        put_attr_local(Other, C)  % rescue conjunction
    ; put_attr_local(Other, Fa)   % rescue conjunction
    ).
:- endif.

attribute_goals(X) -->
    [myfreeze:myfreeze(X, G)],
    {get_attr_local(X, G)}.

attr_portray_hook(G, Var):-
    display(Var),
    display('<-myfrozen('),
    display(G),
    display(')').

% A (meta) conjunction of two goals
meta_conj('$:'(Fa), '$:'(Fb), '$:'('basiccontrol:,'(Fa, Fb))).
```

102.2 Usage and interface

- **Library usage:**
  `:- use_package(attr).`
  or
  `:- module(...,...,[attr]).`
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

103 Attributed variables runtime

Author(s): Rémy Haemmerlé, Christian Holzbaur, Daniel Cabeza, Manuel Carro.

This module provides a means to handle attributed variables. Note that attributes are private to the module from which the predicates implemented by `attr_rt` module are called. See package `attr` for more details about attributed variables.

103.1 Usage and interface

- **Library usage:**
  `:- use_module(library(attr/attr_rt)).`
- **Exports:**
  - *Predicates:*
    `attvar/1`, `put_attr_local/2`, `put_attr/3`, `get_attr_local/2`, `get_attr/3`, `del_attr_local/1`, `del_attr/2`, `copy_term/3`.

103.2 Documentation on exports

attvar/1: PREDICATE

Usage: `attvar(Var)`

Succeeds if Term is an attributed variable

put_attr_local/2: PREDICATE

Usage: `put_attr_local(Var,Value)`

If `Var` is a variable or attributed variable, set its attribute to `Value`. If an attribute is already associated with `Var`, the old value is replaced. Backtracking will restore the old value (i.e., an attribute is a mutable term. See also library `mutables`). This predicate raises a representation error if `Var` is not a variable and a type error if `Module` is not an atom.

put_attr/3: PREDICATE

No further documentation available for this predicate.

get_attr_local/2: PREDICATE

Usage: `get_attr_local(Var,Value)`

Request the current value for the attribute associocted to `Var`. If `Var` is not an attributed variable or the named attribute is not associated to `Var` this predicate fails silently.

get_attr/3: PREDICATE

No further documentation available for this predicate.

del_attr_local/1: PREDICATE

Usage: `del_attr_local(Var)`

If `Var` has an attribute, deletes it, otherwise succeeds without side-effect.

del_attr/2: PREDICATE

No further documentation available for this predicate.

copy_term/3: PREDICATE

Usage: `copy_term(Term,Copy,Gs)`

Creates a regular term `Copy` as a copy of `Term` (without any attributes), and a list `Gs` of goals that when executed reinstate all attributes onto Copy. The nonterminal `attribute_goal/1`, as defined in the modules the attributes stem from, is used to convert attributes to lists of goals.

103.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`, `format`, `sets`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `modes`, `fsyntax`, `dcg`, `datafacts`.

104 Low-level attributed variables

Author(s): Christian Holzbaur, Daniel Cabeza, Manuel Carro.

Direct usage of this module is discouraged. Prefer package `attr`.

These predicates allow the manipulation of *attributed variables*. Attributes are special terms which are attached to a (free) variable, and are hidden from the normal Prolog computation. They can only be treated by using the predicates below.

104.1 Usage and interface

- **Library usage:**
  `:- use_module(engine(attributes)).`
- **Exports:**
  - *Predicates:*
    `attach_attribute/2`, `get_attribute/2`, `update_attribute/2`, `detach_attribute/1`.
  - *Multifiles:*
    `verify_attribute/2`, `combine_attributes/2`.

104.2 Documentation on exports

attach_attribute/2: PREDICATE

Usage: `attach_attribute(Var,Attr)`

Attach attribute `Attr` to `Var`.

- *The following properties should hold at call time:*
  `Var` is a free variable. (`term_typing:var/1`)
  `Attr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
- *The following properties hold upon exit:*
  `Var` is a free variable. (`term_typing:var/1`)
  `Attr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

get_attribute/2: PREDICATE

Usage: `get_attribute(Var,Attr)`

Unify `Attr` with the attribute of `Var`, or fail if `Var` has no attribute.

- *The following properties should hold at call time:*
  `Var` is a free variable. (`term_typing:var/1`)
- *The following properties hold upon exit:*
  `Attr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

update_attribute/2: PREDICATE

Usage: `update_attribute(Var,Attr)`

Change the attribute of attributed variable `Var` to `Attr`.

- *The following properties should hold at call time:*
  `Var` **is a free variable.** (`term_typing:var/1`)
  `Attr` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties hold upon exit:*
  `Var` **is a free variable.** (`term_typing:var/1`)
  `Attr` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

detach_attribute/1: PREDICATE

Usage: `detach_attribute(Var)`

Take out the attribute from the attributed variable `Var`.

- *The following properties should hold at call time:*
  `Var` **is a free variable.** (`term_typing:var/1`)
- *The following properties hold upon exit:*
  `Var` **is a free variable.** (`term_typing:var/1`)

104.3 Documentation on multifiles

verify_attribute/2: PREDICATE

Usage: `verify_attribute(Attr,Term)`

A user defined predicate. **This predicate is called when an attributed variable with attribute `Attr` is about to be unified with the non-variable term `Term`. The user should define this predicate (as multifile) in the modules implementing special unification.**

- *The following properties should hold at call time:*
  `Attr` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Term` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties hold upon exit:*
  `Attr` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Term` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

The predicate is *multifile***.**

combine_attributes/2: PREDICATE

Usage: `combine_attributes(Var1,Var2)`

A user defined predicate. **This predicate is called when two attributed variables with attributes `Var1` and `Var2` are about to be unified. The user should define this predicate (as multifile) in the modules implementing special unification.**

- *The following properties should hold at call time:*
  `Var1` **is a free variable.** (`term_typing:var/1`)
  `Var2` **is a free variable.** (`term_typing:var/1`)
- *The following properties hold upon exit:*
  `Var1` **is a free variable.** (`term_typing:var/1`)
  `Var2` **is a free variable.** (`term_typing:var/1`)

The predicate is *multifile***.**

104.4 Documentation on imports

This module has the following direct dependencies:

– *Packages:*
  prelude, initial, condcomp, assertions, assertions/assertions_basic, nortchecks.

104.5 Other information

Note that combine_attributes/2 **and** verify_attribute/2 **are not called with the attributed variables involved, but with the corresponding attributes instead. The reasons are:**

- **There are simple applications which only refer to the attributes.**
- **If the application wants to refer to the attributed variables themselves, they can be made part the attribute term. The implementation of** freeze/2 **utilizes this technique. Note that this does not lead to cyclic structures, as the connection between an attributed variable and its attribute is invisible to the pure parts of the Prolog implementation.**
- **If attributed variables were passed as arguments, the user code would have to refer to the attributes through an extra call to** get_attribute/2**.**
- **As the/one attribute is the first argument to each of the two predicates, indexing applies. Note that attributed variables themselves look like variables to the indexing mechanism.**

However, future improvements may change or extend the interface to attributed variables in order to provide a richer and more expressive interface.

For customized output of attributed variables, please refer to the documentation of the predicate portray_attribute/2**.**

105 C Foreign Language interface

Author(s): Jose F. Morales, Manuel Carro.

Ciao includes a high-level, flexible way to interface C and Prolog, based on the use of assertions to declare the expected types and modes of the arguments of a Prolog predicate, and which C files contain the corresponding code. To this end, the user provides:

- A set of C files, or a precompiled shared library,
- A Ciao module defining which predicates are implemented in the C files and the types and modes of their arguments, and
- an (optional) set of flags required for the compilation of the files.

The Ciao compiler analyzes the Prolog code written by the user and gathers this information in order to generate automatically C "glue" code implementing the data translation between Prolog and C, and to compile the C code into dynamically loadable C object files, which are linked automatically when needed.

105.1 Declaration of Foreign Sources

The compilation and dynamic linking of foreign sources is controled by the following declarations:

- `use_foreign_source/1` specifies the foreign C (or C++) source file(s).
- `use_foreign_library/1` specifies the external library(ies) to be linked with the foreign sources.
- `use_foreign_gluecode_header/1` specifies custom header for the C gluecode file.
- `use_compiler/1` specifies the compiler to use.
- `extra_compiler_opts/1` specifies the additional compiler options.
- `use_linker/1` specifies the linker to use.
- `extra_linker_opts/1` specifies the additional linker options.

More details about these options can be found in ⟨undefined⟩ [Foreign Language Interface Properties], page ⟨undefined⟩.

The compilation of the foreign sources together with the glue code is decomposed into two phases:

- The Compilation phase: Each C (or C++) file `<SrcFile>` is compiled into an object file `<ObjFile>` by a command of the form:

  ```
  $ <Compiler> <CompilerOpts> -c <ExtraCompilerOptions> -o <ObjectFile> <SrcFile>
  ```

  where `<Compiler>` is the compiler command, `<CompilerOpts>` are some Ciao-specific compiler options, and `<ExtraCompilerOptions>` are the additional compiler options specified by the declaration(s) `extra_compiler_opts/1`.
- The Linking phase: All the object files `<ObjectFiles>` obtained by the compilation phase are linked together with the external libray(ies) into a dynamic library `<SOLib>` by a command of the form:

  ```
  $ <Linker> <CompilerOpts> -o <SOLib> <ObjectFiles> <ExternalLibs> <ExtraLinkerOptions>
  ```

  where `<Linker>` is the linker command, `<LinkerOpts>` are some Ciao-specific linker options, `<ExternalLibs>` are the foreign libraries in order they are declared, and `<ExtraLinkerOptions>` are the additional linker options specified by the declaration(s) `extra_linker_opts/1`.

Note that for the linking phase the order of libraries may be important. See https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html or https://sourceware.org/binutils/docs-2.24/ld/Options.html#Options for more details.

Once the foreign sources are compiled and linked together, the resulting library is dynamically loaded using the semantics of the `RTLD_LOCAL` flag of the **POSIX** `dlopen()` function (see the `dlopen` man page for additional details). In other words, the symbols defined in this dynamic library are not made available to resolve references in subsequently foreign loaded libraries. In particular the foreign sources cannot depend on previously loaded foreign sources.

105.2 Declaring Types and Modes

Each predicate implemented as a foreign C function must have accompanying declarations in the Ciao associated file stating the types and modes of the C function. A sample declaration for `prolog_predicate` which is implemented as `foreign_function_name` is:

```
:- trust pred prolog_predicate( m1(Arg1), ... mN(ArgN) )
     :: type1 * ... * typeN
      + ( foreign(foreign_function_name), returns(ArgR) ).
```

where

- `m1`, ..., `mN` and `type1`, ..., `typeN` are respectively the modes and types of the arguments (see ⟨undefined⟩ [Equivalence between Ciao and C modes], page ⟨undefined⟩ and ⟨undefined⟩ [Equivalence between Ciao and C types], page ⟨undefined⟩ for details).
- `foreign_function_name` is the name of the C function implementing `prolog_predicate/N`, and
- `returns(ArgR)` specifies that the result of this function is unified with argument `ArgR` of the Ciao predicate (which must be one of `Arg1` ... `ArgN`).

This notation can be simplified in several ways:

- If the name of the foreign function is the same as the name of the Ciao predicate, `foreign(foreign_function_name)` can be replaced by `foreign`.
- If the foreign function does not return anything (or if its value is ignored), then `returns(ArgR)` must be removed. Note that `returns` cannot be used without `foreign`.

A simplified, minimal form is thus:

```
:- trust pred prolog_predicate(m1(Arg1), ... mN(ArgN))
     :: type1 * ... * typeN + foreign.
```

105.3 Equivalence between Ciao and C types

The automatic translation between Ciao and C types is defined for some simple but useful Prolog types (like atoms and strings) and for a collection of reserved Prolog types (ctypes) that reflect C types.

The names (and meaning) of the types known for performing that translation are to be found in ⟨undefined⟩ [Foreign Language Interface Properties], page ⟨undefined⟩; they are also summarized below (Prolog types are listed first and the corresponding C types after them):

- `c_short`, `c_ushort`, `c_int`, `c_uint`, `c_long`, `c_ulong`, `c_uintptr`, `c_size`

  correspond to the C types:

  `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `uintptr_t`, `size_t`

  (be aware that the ranges of all types depend on the C data model, which differs in, e.g., 32 and 64-bit architectures).

- c_int8, c_uint8, c_int16, c_uint16, c_int32, c_uint32, c_int64, c_uint64 correspond to the C types:
  int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t.
- atm corresponds to C's 'char *' (with trailing zero).
- string corresponds to C's 'char *' (with trailing zero).
- X_list corresponds to C's 'T *' (an array of some ctype X with C type T, with associated length).
- address corresponds to C's 'void *'.

Strings, atoms, and lists of bytes are passed to (and from) C as dynamically (ciao_malloc) created arrays of characters (bytes). Those arrays are freed by Ciao upon return of the foreign function unless the property do_not_free/2 is specified (see examples below). This caters for the case in which the C files save in a private state (either by themselves, or by a library function being called by them) the values passed on from Prolog. The type X_list/1 requires an additional property, size_of/2, to indicate which argument represents its size.

Empty lists of bytes and integers are converted into C NULL pointers, and vice-versa. Empty strings ([]) and null atoms (") are converted into zero-length, zero-ended C strings (""). C NULL strings and empty arrays (i.e., arrays with zero length) are transformed into the empty list or the null atom ('').

Most of the work is performed by the predicates in the ⟨undefined⟩ [Foreign Language Interface Builder], page ⟨undefined⟩, which can be called explicitly by the user. Doing that is not usually needed, since the Ciao Compiler takes care of building glue code files an of compiling and linking whatever is necessary.

The translation to be performed is solely defined by the types of the arguments in the Ciao module (i.e., no inspection of the corresponding C file is done).

> An incorrect selection of types may generate wrong C prototypes. Those problems are silently ignored during compilation.

105.4 Equivalence between Ciao and C modes

The (prefix) +/1 ISO mode (or, equivalently, the in/1 mode) states that the corresponding Prolog argument is ground at the time of the call, and therefore it is an input argument in the C part; this groundness is automatically checked upon entry. The (prefix) -/1 ISO mode (or, equivalently, the go/1 mode) states that Prolog expects the C side to generate a (ground) value for that argument. Arguments with output mode should appear in C functions as pointers to the corresponding base type (as it is usual with C), i.e., an argument which is an integer generated by the C file, declared as:

```
:- trust pred get_int(go(ThisInt)) :: c_int + foreign.
```

or as:

```
:- trust pred get_int(-ThisInt) :: c_int + foreign.
```

should appear in the C code as

```
void get_int(int *thisint) {
    ....
}
```

Note the type of the (single) argument of the function. Besides, the return value of a function can always be used as an output argument, just by specifying to which Prolog arguments it corresponds, using the returns/1 property. The examples below illustrate this point, and the use of several assertions to guide the compilation process.

105.5 Custom access to Prolog from C

Automatic type conversion does not cover all the possible cases. When the automatic type conversion is not enough (or if the user, for any reason, does not want to go through the automatic conversion), it is possible to instruct Ciao not to make implicit type conversion. The strategy in that case is to pass the relevant argument(s) with a special type (a `ciao_term`) which can represent any term which can be built in Prolog. Operations to construct, traverse, and test this data abstraction from C are provided. The prototypes of these operations are placed on the `"ciao_prolog.h"` file, under the `include` subdirectory of the installation directory (the Ciao compiler knowns where it has been installed, and gives the C compiler the appropriate flags). This *non direct correspondence* mode is activated whenever a Ciao Prolog type unknown to the foreign interface (i.e., none of those in ⟨undefined⟩ [Foreign Language Interface Properties], page ⟨undefined⟩) or the type `any_term` (which is explicitly recognized by the foreign language interface) is found. The latter is preferred, as it is much more informative, and external tools, such as the the CiaoPP preprocessor, can take advantage of them.

105.5.1 Term construction

All term construction primitives return an argument of type `ciao_term`, which is the result of constructing a term. All Ciao Prolog terms can be built using the interface operations `ciao_var()`, `ciao_structure()`, `ciao_`*X*`()`, etc. There are, however, variants and specialized versions of these operations which can be freely intermixed. Using one version or another is a matter of taste and convenience. We list below the prototypes of the primitives in order of complexity.

Throughout this section, true, when referred to a boolean value, corresponds to the integer value `1`, and false corresponds to the integer value `0`, as is customary in C boolean expressions. These values are also available as the (predefined) constants `ciao_true` and `ciao_false`, both of type `ciao_bool`.

- `ciao_term ciao_var();`

  Returns a fresh, unbound variable.

- `ciao_term ciao_mk_`*X*`(T i);`

  Creates a term of ctype *X* from the value of `i`, of the corresponding C type `T`.

- `ciao_term ciao_put_number_chars(char *number_string);`

  It converts `number_string` (which must a string representing a syntactically valid number) into a `ciao_term`.

- `ciao_term ciao_atom(char *name);`

  Creates an atom whose printable name is given as a C string.

- `ciao_term ciao_structure_a(char *name, int arity, ciao_term *args);`

  Creates a structure with name `name` (i.e., the functor name), arity `arity` and the components of the array `args` as arguments: `args[0]` will be the first argument, `args[1]` the second, and so on. The `args` array itself is not needed after the term is created, and can thus be a variable local to a procedure. An atom can be represented as a 0-arity structure (with `ciao_structure(name, 0)`), and a list cell can be constructed using the `'.'/2` structure name. The `_a` suffix stands for *array*.

- `ciao_term ciao_structure(char *name, int arity, ...);`

  Similar to ciao_structure_a, but the C arguments after the arity are used to fill in the arguments of the structure.

- `ciao_term ciao_list(ciao_term head, ciao_term tail);`

  Creates a list from a `head` and a `tail`. It is equivalent to `ciao_structure(".", 2, head, tail)`.

- `ciao_term ciao_empty_list();`

  Creates an empty list. It is equivalent to `ciao_atom("[]")`.
- `ciao_term ciao_listn_a(int len, ciao_term *args);`

  Creates a list with 'len' elements from the array `args`. The *nth* element of the list (starting at 1) is `args[n-1]` (starting at zero).
- `ciao_term ciao_listn(size_t length, ...);`

  Like `ciao_listn_a()`, but the list elements appear explicitly as arguments in the call.
- `ciao_term ciao_dlist_a(size_t len, ciao_term *args, ciao_term base);`

  Like `ciao_listn_a`, but a difference list is created. `base` will be used as the tail of the list, instead of the empty list.
- `ciao_term ciao_dlist(size_t length, ...);`

  Similar to `ciao_dlist_a()` with a variable number of arguments. The last one is the tail of the list.
- `ciao_term ciao_copy_term(ciao_term src_term);`

  Returns a new copy of the `term`, with fresh variables (as `copy_term/2` does).

105.5.2 Testing the Type of a Term

A `ciao_term` can contain *any* Prolog term, and its implementation is opaque to the C code. Therefore the only way to know reliably what data is passed on is using explicit functions to test term types. Below, `ciao_bool` is a type defined in `"ciao_prolog.h"` which can take the values 1 (for true) and 0 (for false).

- `ciao_bool ciao_is_variable(ciao_term term);`

  Returns true if `term` is currently an uninstantiated variable.
- `ciao_bool ciao_is_number(ciao_term term);`

  Returns true if `term` is an integer (of any length) or a floating point number.
- `ciao_bool ciao_is_integer(ciao_term term);`

  Returns true if `term` is instantiated to an integer.
- `ciao_bool ciao_fits_in_X(ciao_term term);`

  Returns true if `term` is instantiated to a value representable by ctype `X` (e.g., 10000 is not representable by `c_int8` but it is for `c_int32`).
- `ciao_bool ciao_is_atom(ciao_term atom);`

  Returns true if `term` is an atom.
- `ciao_bool ciao_is_list(ciao_term term);`

  Returns true if `term` is a list (actually, a `cons` cell).
- `ciao_bool ciao_is_empty_list(ciao_term term);`

  Returns true if `term` is the atom which represents the empty list (i.e., `[]`).
- `ciao_bool ciao_is_structure(ciao_term term);`

  Returns true if `term` is a structure of any arity. This includes atoms (i.e., structures of arity zero) and lists, but excludes variables and numbers.

105.5.3 Term navigation

The functions below can be used to recover the value of a `ciao_term` into C variables, or to inspect Prolog structures.

- `int ciao_get_X(ciao_term term);`

  Converts `term` to the corresponding C type for the ctype `X`. For integer types, `ciao_is_integer(term)` must hold. For floating point, `ciao_is_number(term)` must hold.

- `char *ciao_get_number_chars(ciao_term term);`

  It converts `ciao_term` (which must be instantiated to a number) into a C string representing the number in the current radix. The string returned is a copy, which must (eventually) be explicitly deallocated by the user C code using the operation `ciao_free()`
- `char *ciao_atom_name(ciao_term atom);`

  Returns the name of the atom. The returned string *is the one internally used by Ciao*, and should not be deallocated, changed or altered in any form. The advantage of using it is that it is fast, as no data copying is needed.
- `char *ciao_atom_name_dup(ciao_term atom);`

  Obtains a copy of the name of the atom. The string can be modified, and the programmer has the responsibility of deallocating it after being used. Due to the copy, it is slower than calling `char *ciao_atom_name()`.
- `ciao_term ciao_list_head(ciao_term term);`

  Extracts the head of the list `term`. Requires `term` to be a list.
- `ciao_term ciao_list_tail(ciao_term term);`

  Extracts the tail of the list `term`. Requires `term` to be a list.
- `char *ciao_structure_name(ciao_term term);`

  Extracts the name of the structure `term`. Requires `term` to be a structure.
- `int ciao_structure_arity(ciao_term term);`

  Extracts the arity of the structure `term`.

  Requires `term` to be a structure.
- `ciao_term ciao_structure_arg(ciao_term term, int n);`

  Extracts the *nth* argument of the structure `term`. It behaves like `arg/3`, so the first argument has index 1. Requires `term` to be a structure.

105.5.4 Testing for Equality and Performing Unification

Variables of type `ciao_term` cannot be tested directly for equality: they are (currently) implemented as a sort of pointers which may be aliased (two different pointers may refer to the same object). The interface provides helper functions for testing term equality and to perform unification of terms.

- `ciao_bool ciao_unify(ciao_term x, ciao_term y);`

  Performs the unification of the terms `x` and `y`, and returns true if the unification was successful. This is equivalent to calling the (infix) Prolog predicate `=/2`. The bindings are trailed and undone on backtracking.
- `ciao_bool ciao_equal(ciao_term x, ciao_term y);`

  Performs equality testing of terms, and returns true if the test was successful. This is equivalent to calling the (infix) Prolog predicate `==/2`. Equality testing does not modify the terms compared.

105.5.5 Raising Exceptions

The following functions offer a way of throwing exceptions from C that can be caught in Prolog with `catch/3`. The term that reaches Prolog is exactly the same which was thrown by C. The execution flow is broken at the point where `ciao_raise_exception()` is executed, and it returns to Prolog.

- `void ciao_raise_exception(ciao_term ball);`

  Raises an exception an throws the term `ball`.

105.5.6 Creating and disposing of memory chunks

Memory to be used solely by the user C code can be reserved/disposed of using, e.g., the well-known `malloc()`/`free()` functions (or whatever other functions the user may have available). However, memory explicitly allocated by Ciao and passed to C code, or allocated by C code and passed on to Ciao (and subject to garbage collection by it) should be allotted and freed (when necessary) by using the functions:

- `void *ciao_malloc(int size);`
- `void ciao_free(void *pointer);`

whose behavior is similar to `malloc()`/`free()`, but which will cooordinate properly with Ciao's internal memory management.

105.5.7 Calling Prolog from C

It is also possible to make arbitrary calls to Prolog predicates from C. There are two basic ways of making a query, depending on whether only one solution is needed (or if the predicate to be called is known to generate only one solution), or if several solutions are required.

When only one solution is needed `ciao_commit_call` obtains it (the solution obtained will obviously be the first one) and discards the resources used for finding it:

- `ciao_bool ciao_commit_call(char *name, int arity, ...);`

  Makes a call to a predicate and returns true or false depending on whether the query has succedeed or not. In case of success, the (possibly) instantiated variables are reachable from C.
- `ciao_bool ciao_commit_call_term(ciao_term goal);`

  Like `ciao_commit_call()` but uses the previously built term `goal` as goal.

If more than one solution is needed, it is necessary to use the `ciao_query` operations. A consult begins with a `ciao_query_begin` which returns a `ciao_query` object. Whenever an additional solution is required, the `ciao_query_next` function can be called. The query ends by calling `ciao_query_end` and all pending search branches are pruned.

- `ciao_query *ciao_query_begin(char *name, int arity, ...);`

  The predicate with the given name, arity and arguments (similar to the `ciao_structure()` operation) is transformed into a `ciao_query` object which can be used to make the actual query.
- `ciao_query *ciao_query_begin_term(ciao_term goal);`

  Like ciao_query_begin but using the term `goal` instead.
- `ciao_bool ciao_query_ok(ciao_query *query);`

  Determines whether the query may have pending solutions. A false return value means that there are no more solutions; a true return value means that there are more possible solutions.
- `void ciao_query_next(ciao_query *query);`

  Ask for a new solution.
- `void ciao_query_end(ciao_query *query);`

  Ends the query and frees the used resources.

105.6 Examples

105.6.1 Mathematical functions

In this example, the standard mathematical library is accessed to provide the *sin*, *cos*, and *fabs* functions. Note that the library is specified simply as

```
:- use_foreign_library([m]).
```

The foreign interface adds the `-lm` at compile time. Note also how some additional options are added to optimize the compiled code (only glue code, in this case) and mathematics (only in the case of Linux in an Intel processor).

File *math.pl*:

```
:- module(math, [sin/2, cos/2, fabs/2], [foreign_interface]).

:- trust pred sin(in(X),go(Y)) :: c_double * c_double + (foreign,returns(Y)).
:- trust pred cos(in(X),go(Y)) :: c_double * c_double + (foreign,returns(Y)).
:- trust pred fabs(in(X),go(Y)) :: c_double * c_double + (foreign,returns(Y)).

:- extra_compiler_opts(['-O2']).
:- extra_compiler_opts('LINUXi686',['-ffast-math']).
:- extra_compiler_opts('LINUXx86_64',['-ffast-math']).
:- use_foreign_library('LINUXi686', m).
:- use_foreign_library('LINUXx86_64', m).
```

105.6.2 Addresses and C pointers

The `address` type designates any pointer, and provides a means to deal with C pointers in Prolog without interpreting them whatsoever. The C source file which implements the operations accessed from Prolog is declared with the

```
:- use_foreign_source(objects_c).
```

directive.

File *objects.pl*:

```
:- module(objects, [object/2, show_object/1], [foreign_interface]).

:- trust pred object(in(N),go(Object)) ::
    c_int * address + (foreign,returns(Object)).

:- trust pred show_object(in(Object)) ::
    address + foreign.

:- use_foreign_source(objects_c).
:- extra_compiler_opts(['-O2']).
```

File *objects_c.c*:

```
#include <stdio.h>

struct object {
  char *name;
  char *colour;
};

#define OBJECTS 3
```

```
struct object objects[OBJECTS] =
{ {"ring","golden"},
  {"table","brown"},
  {"bottle","green"} };

struct object *object(int n) {
  return &objects[n % OBJECTS];
}

void show_object(struct object *o) {
  printf("I show you a %s %s\n", o->colour, o->name);
}
```

105.6.3 Lists of bytes and arrays

A list of bytes (c.f., a list of ints) corresponds to a byte array in C. The length of the array is associated to that of the list using the property `size_of/2`. The returned array is freed by Ciao Prolog upon its recepction, unless the `do_not_free/1` property is specified (see later). Conversely, a list of natural numbers in the range 0 to 255 can be passed to C as an array.

File *byte_lists.pl***:**

```
:- module(byte_lists, [obtain_list/3, show_list/2], [foreign_interface]).

:- trust pred obtain_list(in(N),go(Length),go(List)) :: c_int * c_size * c_uint8_list
    + (foreign,size_of(List,Length)).
:- trust pred show_list(in(Length),in(List)) :: c_size * c_uint8_list
    + (foreign,size_of(List,Length)).

:- use_foreign_source(bytes_op).
```

File *bytes_op.c***:**

```
#include <stdlib.h>
#include <stdio.h>

void obtain_list(int n, size_t *l, unsigned char **s) {
  int i;
  if (n < 0) n = 0;
  *l = n;
  *s = (unsigned char *)malloc(*l);
  for (i = 0; i < *l; i++) {
    (*s)[i] = i;
  }
}

void show_list(size_t l, unsigned char *s) {
  if (s) {
    size_t n;
    printf("From C:");
    for (n = 0; n < l; n++) {
```

```
      printf(" %d", s[n]);
    }
    printf(".\n");
  } else {
    printf("From C: []\n");
  }
}
```

105.6.4 Lists of integers

File *int_lists.pl*:

```
:- module(int_lists, [obtain_list/3, show_list/2], [foreign_interface]).

:- trust pred obtain_list(in(N),go(Length),go(List)) :: c_size * c_size * c_int_list
    + (foreign,size_of(List,Length)).
:- trust pred show_list(in(Length),in(List)) :: c_size * c_int_list
    + (foreign,size_of(List,Length)).

:- use_foreign_source(ints_op).
```

File *ints_op.c*:

```
#include <stdlib.h>
#include <stdio.h>

void obtain_list(size_t n, size_t *l, int **s) {
  int i;
  *l = n;
  *s = (int *)malloc((*l) * sizeof(int));
  for (i = 0; i < *l; i++) {
    (*s)[i] = i;
  }
}

void show_list(size_t l, int *s) {
  if (s) {
    int n;
    printf("From C:");
    for (n = 0; n < l; n++) {
      printf(" %d", s[n]);
    }
    printf(".\n");
  } else {
    printf("From C: []\n");
  }
}
```

105.6.5 Strings and atoms

A C string can be seen as an array whose end is denoted by the trailing zero, and therefore stating its length is not needed. Two translations are possible into Ciao: as a Prolog string

(list of bytes, with no trailing zero) and as an atom. These are selected automatically just by choosing the corresponding type (look at the examples below).

Note how the `do_not_free/1` **property is specified in the** `a_string/1` **predicate: the string returned by C is static, and therefore it should not be freed by Prolog.**

File *strings_and_atoms.pl***:**

```
:- module(strings_and_atoms,
    [ lookup_string/2,
      lookup_atom/2,
      a_string/1,
      show_string/1,
      show_atom/1
    ],
    [foreign_interface]).

:- trust pred a_string(go(S)) ::
    string + (foreign(get_static_str),returns(S),do_not_free(S)).

:- trust pred lookup_string(in(N),go(S)) ::
    c_int * string + (foreign(get_str),returns(S)).
:- trust pred lookup_atom(in(N),go(S)) ::
    c_int * atm + (foreign(get_str),returns(S)).

:- trust pred show_string(in(S)) :: string + foreign(put_str).
:- trust pred show_atom(in(S)) :: atm + foreign(put_str).

:- use_foreign_source(str_op).
```

File *str_op.c***:**

```
#include <stdlib.h>
#include <stdio.h>

char *get_static_str(void) {
  return "this is a string Prolog should not free";
}

char *get_str(int n) {
  char *s;
  int size;
  int i;
  int c;
  if (n < 0) n = -n;
  size = (n%4) + 5;
  s = (char *)malloc(size+1);
  for (i = 0, c = ((i + n) % ('z' - 'a' + 1)) + 'a'; i < size; i++,c++) {
    if (c > 'z') c = 'a';
    s[i] = c;
  }
  s[i] = 0;
  return s;
}
```

```
void put_str(char *s) {
  if (s) {
    printf("From C: \"%s\"\n", s);
  } else {
    printf("From C: null\n");
  }
}
```

105.6.6 Arbitrary Terms

This example shows how data Prolog can be passed untouched to C code, and how it can be manipulated there.

File *any_term.pl*:

```
:- module(any_term,
    [custom_display_term/1,
     custom_create_term/2
    ],
    [foreign_interface]).

:- trust pred custom_display_term(in(X)) :: any_term + foreign.
:- trust pred custom_create_term(in(L), go(X)) :: c_int * any_term + (foreign,returns(X)).

:- use_foreign_source(any_term_c).
:- extra_compiler_opts(['-O2']).
```

File *any_term_c.c*:

```
#include <stdio.h>
#include <ciao_prolog.h>

ciao_term custom_create_term(int n) {
  ciao_term t;
  t = ciao_empty_list();
  while (n > 0) {
    t = ciao_list(ciao_mk_c_int(n), t);
    n--;
  }
  return t;
}

void custom_display_term(ciao_term term) {
  if (ciao_is_atom(term)) {
    printf("<atom name=\"%s\"/>", ciao_atom_name(term));
  } else if (ciao_is_structure(term)) {
    int i;
    int a;
    a = ciao_structure_arity(term);
    printf("<structure name=\"%s\" arity=\"%d\">", ciao_structure_name(term), a);
    for (i = 1; i <= a; i++) {
      printf("<argument number=\"%d\">", i);
      custom_display_term(ciao_structure_arg(term, i));
```

```
      printf("</argument>");
    }
    printf("</structure>");
  } else if (ciao_is_list(term)) {
    printf("<list>");
    printf("<head>");
    custom_display_term(ciao_list_head(term));
    printf("</head>");
    printf("<tail>");
    custom_display_term(ciao_list_tail(term));
    printf("</tail>");
    printf("</list>");
  } else if (ciao_is_empty_list(term)) {
    printf("<empty_list/>");
  } else if (ciao_is_integer(term)) {
    printf("<integer value=\"%d\"/>", ciao_get_c_int(term));
  } else if (ciao_is_number(term)) {
    printf("<float value=\"%f\"/>", ciao_get_c_float(term));
  } else {
    printf("<unknown/>");
  }
}
```

105.6.7 Exceptions

The following example defines a predicate in C that converts a list of codes into a number using `strtol()`. If this conversion fails, then a exception is raised.

File *exceptions_example.pl***:**

```
:- module(exceptions_example,
    [codes_to_number_c/2,
     safe_codes_to_number/2
    ],
    [foreign_interface]).

:- use_module(library(format)).

% If the string is not a number raises an exception.
:- trust pred codes_to_number_c(in(X), go(Y)) :: string * c_int + (foreign, returns(Y)).

safe_codes_to_number(X, Y) :-
    catch(codes_to_number_c(X, Y), Error, handle_exception(Error)).

handle_exception(Error) :- format("Exception caught ~w~n", [Error]).

:- use_foreign_source(exceptions_c).
:- extra_compiler_opts(['-O2']).
```

File *exceptions_c.c***:**

```
#include <string.h>
#include <stdlib.h>
```

```
#include <ciao_prolog.h>

int codes_to_number_c(char *s) {
  char *endptr;
  int n;
  n = strtol(s, &endptr, 10);
  if (endptr == NULL || *endptr != '\0') {
    ciao_raise_exception(ciao_structure("codes_to_number_exception",
                                        1,
                                        ciao_atom(s)));
  }
  return n;
}
```

105.6.8 Testing number types and using unbounded length integers

Unbounded length integers (and, in general, any number) can be converted to/from `ciao_terms` by using strings. The following examples show two possibilities: one which tries to be as smart as possible (checking whether numbers fit into a machine int or not), and being lazy and simpler –and probably slower.

File *bigints.pl*:

```
:- module(bigints,
    [
     make_smart_conversion/3, % Checks and uses convenient format
     force_string_conversion/2  % Passes around using strings
    ],
    [foreign_interface]).

:- trust pred make_smart_conversion_c(in(X), go(Y), go(How))
   :: any_term * any_term * any_term + foreign
   # "Given a number @var{X}, it is unified with @var{Y} by using the
   most specific internal representation (short integer, float, or
   long integer).  @var{How} returns how the conversion was done.  It
   behaves unpredictably if @var{X} is not a number.".

:- trust pred force_string_conversion_c(in(X), go(Y))
   :: any_term * any_term + foreign
   # "Given a number @var{X}, it is unified with @var{Y} by using the
   most general internal representation (a string of characters). It
   behaves unpredictably if @var{X} is not a number.".

:- use_foreign_source(bigints_c).

make_smart_conversion(A, B, C):-
    number(A),                              % Safety test
    make_smart_conversion_c(A, B, C).

force_string_conversion(A, B):-
    number(A),                              % Safety test
```

```
    force_string_conversion_c(A, B).
```

File *bigints_c.c*:

```
#include <ciao_prolog.h>

void make_smart_conversion_c(ciao_term  number_in,
                             ciao_term *number_out,
                             ciao_term *how_converted) {
  int    inter_int;
  double inter_float;
  char * inter_str;

  if (ciao_fits_in_c_int(number_in)) {/* Includes the case of being a float */
    inter_int = ciao_get_c_int(number_in);
    *number_out = ciao_mk_c_int(inter_int);
    *how_converted = ciao_atom("machine_integer");
  } else
    if (ciao_is_integer(number_in)) { /* Big number */
      inter_str   = ciao_get_number_chars(number_in);
      *number_out = ciao_put_number_chars(inter_str);
      ciao_free(inter_str);
      *how_converted = ciao_atom("string");
    } else { /* Must be a float */
      inter_float = ciao_get_c_double(number_in);
      *number_out = ciao_mk_c_double(inter_float);
      *how_converted = ciao_atom("float");
    }
}

void force_string_conversion_c(ciao_term  number_in,
                               ciao_term *number_out) {
  char *inter_str;
  inter_str  = ciao_get_number_chars(number_in);
  *number_out = ciao_put_number_chars(inter_str);
  ciao_free(inter_str);
}
```

105.6.9 Interfacing with C++

Ciao code can be interfaced easily with C++ using this interface. The basic idea is to write C functions (functions prefixed by 'extern "C"') within the C++ code to make the bridge between calls from Ciao to C++. Then, C++ objects can be cast as addresses. Because the foreign interface assumes that the foreign source is classical C, C++ source files should be declared with their extension.

File *cc_stack.pl*:

```
:- module(cc_stack, [cc_stack_new/1, cc_stack_size/2,
                     cc_stack_push/2, cc_stack_pop/1, cc_stack_top/2],
                    [foreign_interface, assertions]).

:- use_module(library(odd), [undo/1]).
```

```
:- trust pred ciao_stack_new(go(Stack)) :: address
    + (foreign, returns(Stack)).
:- trust pred ciao_stack_delete(in(_Stack)) :: address
    + foreign.
:- trust pred ciao_stack_size(in(_Stack), go(Size)) ::  (address * c_int)
    + (foreign, returns(Size)).
:- trust pred ciao_stack_push(in(_Stack), in(_Value)) :: (address * c_int)
    + foreign.
:- trust pred ciao_stack_pop(in(_Stack)) ::  address
    + foreign.
:- trust pred ciao_stack_top(in(_Stack), go(Value)) ::  (address * c_int)
    + (foreign, returns(Value)).

cc_stack_new(cc_stack(X)) :-
    ciao_stack_new(X),
    % stack are deallocated on backtrack.
    undo(ciao_stack_delete(X)).

cc_stack_size(cc_stack(X), Size):-
    ciao_stack_size(X, Size).

cc_stack_push(cc_stack(X), I):-
    ciao_stack_push(X, I).

cc_stack_pop(cc_stack(X)):-
    (
        ciao_stack_size(X, Size), Size > 0  ->
        ciao_stack_pop(X)
    ;
        throw(error(empty_cc_stack, cc_stack_pop/1-1))
    ).

cc_stack_top(cc_stack(X), Int):-
    (
        ciao_stack_size(X, Size), Size > 0  ->
        ciao_stack_top(X, Int)
    ;
        throw(error(empty_cc_stack, cc_stack_top/1-1))
    ).


:- use_foreign_library('stdc++').
:- use_foreign_source('cc_stack.cc').
```

File *cc_stack.cc***:**

```
#include <stack>
using namespace std;
```

```
typedef stack<int> ciao_stack;

extern "C" void *
ciao_stack_new(void)
{
  return (void*) new ciao_stack;
}

extern "C" void
ciao_stack_delete(void * S)
{
  delete ((ciao_stack *) S);
}

extern "C" int
ciao_stack_size(void * S)
{
  return  (((ciao_stack *) S)->size());
}

extern "C" void
ciao_stack_push(void * S, int v)
{
  ((ciao_stack *) S)->push(v);
}

extern "C" void
ciao_stack_pop(void * S)
{
  ((ciao_stack *) S)->pop();
}

extern "C" int
ciao_stack_top(void * S)
{
  return ((ciao_stack *) S)->top();
}
```

105.6.10 Embedding a Ciao engine into a C/C++ application

It is possible to include a Ciao engine (compiled as a static or dynamic library) into an existing C or C++ application. To do that it is necessary to call the `ciao_opts()` and `ciao_init()` functions to initialize the engine and some `ciao_load_qfile()` to load the necessary bytecode files. The structure of the C program is similar to:

```
#include <ciao_prolog.h>

int main(void) {
  ciao_opts("program_name", 0, NULL, 0, NULL, NULL);
  ciao_init(NULL);
  ciao_load_qfile("...");
  ...
```

```
    return 0;
  }
```

See the files at `foreign_interface/embedding_example/` **for a complete detailed example including a sample build script.**

105.7 Usage and interface

- **Library usage:**
  `:- use_package(foreign_interface).`
  or
  `:- module(...,...,[foreign_interface]).`
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

106 Foreign Language interface properties

Author(s): Jose F. Morales, Manuel Carro.

The C foreign language interface uses some properties to specify linking regimes, foreign files to be compiled, types of data available, memory allocation policies, etc.

106.1 Usage and interface

- **Library usage:**
  `:- use_module(library(foreign_interface/foreign_interface_properties)).`
- **Exports:**
  - *Properties:*
    `size_of/3`, `do_not_free/2`, `ttr/3`, `returns/2`, `needs_ciao_ctx/1`, `foreign/1`, `foreign/2`, `foreign_low/1`, `foreign_low/2`.
  - *Regular Types:*
    `any_term/1`, `address/1`, `null/1`, `c_short/1`, `c_int/1`, `c_long/1`, `c_ushort/1`, `c_uint/1`, `c_ulong/1`, `c_uintptr/1`, `c_size/1`, `c_float/1`, `c_double/1`, `c_int8/1`, `c_int8/1`, `c_int8/1`, `c_int16/1`, `c_int32/1`, `c_int64/1`, `c_uint8/1`, `c_uint16/1`, `c_uint32/1`, `c_uint64/1`, `c_uint8_list/1`, `c_int_list/1`, `c_double_list/1`.

106.2 Documentation on exports

any_term/1: REGTYPE

Usage: `any_term(X)`

`X` is any term. The foreign interface passes it to C functions as a general term.

address/1: REGTYPE

Usage: `address(Address)`

`Address` is a memory address.

null/1: REGTYPE

Usage: `null(Address)`

`Address` is a null address.

c_short/1: REGTYPE

Usage: `c_short(X)`

`X` is an integer in the range of C `short`.

c_int/1: REGTYPE

Usage: `c_int(X)`

`X` is an integer in the range of C `int`.

c_long/1: REGTYPE

Usage: `c_long(X)`

`X` is an integer in the range of C `long`.

c_ushort/1: REGTYPE

Usage: `c_ushort(X)`

`X` is an integer in the range of C `unsigned short`.

c_uint/1: REGTYPE

Usage: `c_uint(X)`

`X` is an integer in the range of C `unsigned int`.

c_ulong/1: REGTYPE

Usage: `c_ulong(X)`

`X` is an integer in the range of C `unsigned long`.

c_uintptr/1: REGTYPE

Usage: `c_uintptr(X)`

`X` is an integer in the range of C `uintptr_t`.

c_size/1: REGTYPE

Usage: `c_size(X)`

`X` is an integer in the range of C `size_t`.

c_float/1: REGTYPE

Usage: `c_float(X)`

`X` is an integer in the range of C `float`.

c_double/1: REGTYPE

Usage: `c_double(X)`

`X` is an integer in the range of C `double`.

c_int8/1: REGTYPE

Usage 1: `c_int8(X)`

`X` is an integer in the range of C `int8_t`.

Usage 2: `c_int8(X)`

`X` is an integer in the range of C `int8_t`.

Usage 3: `c_int8(X)`

`X` is an integer in the range of C `int8_t`.

c_int8/1: REGTYPE

Usage 1: `c_int8(X)`

`X` is an integer in the range of C `int8_t`.

Usage 2: `c_int8(X)`

`X` is an integer in the range of C `int8_t`.

Usage 3: `c_int8(X)`

`X` is an integer in the range of C `int8_t`.

c_int8/1: REGTYPE

Usage 1: `c_int8(X)`

`X` is an integer in the range of C `int8_t`.

Usage 2: `c_int8(X)`

`X` is an integer in the range of C `int8_t`.

Usage 3: `c_int8(X)`

`X` is an integer in the range of C `int8_t`.

c_int16/1: REGTYPE

Usage: `c_int16(X)`

`X` is an integer in the range of C `int16_t`.

c_int32/1: REGTYPE

Usage: `c_int32(X)`

`X` is an integer in the range of C `int32_t`.

c_int64/1: REGTYPE

Usage: `c_int64(X)`

`X` is an integer in the range of C `int64_t`.

c_uint8/1: REGTYPE

Usage: `c_uint8(X)`

`X` is an integer in the range of C `uint8_t`.

c_uint16/1: REGTYPE

Usage: `c_uint16(X)`

`X` is an integer in the range of C `uint16_t`.

c_uint32/1: REGTYPE

Usage: `c_uint32(X)`

`X` is an integer in the range of C `uint32_t`.

c_uint64/1: REGTYPE

Usage: `c_uint64(X)`

`X` is an integer in the range of C `uint64_t`.

c_uint8_list/1: REGTYPE

Usage: `c_uint8_list(List)`

`List` is a list of `c_uint8/1`.

c_int_list/1: REGTYPE

Usage: `c_int_list(List)`

`List` is a list of `c_int/1`.

c_double_list/1: REGTYPE

Usage: `c_double_list(List)`

`List` is a list of `c_double/1`.

size_of/3: PROPERTY

Usage: `size_of(Name,ListVar,SizeVar)`

For predicate `Name`, the size of the list argument `ListVar`, is given by the argument of type integer `SizeVar`.

do_not_free/2: PROPERTY

Usage: `do_not_free(Name,Var)`

For predicate `Name`, the C argument passed to (returned from) the foreign function will not be freed after calling the foreign function.

ttr/3: PROPERTY

Usage: `ttr(Name,Var,TTr)`

For predicate `Name`, the C argument will be translated using `TTr` as term translator.

returns/2: PROPERTY

Usage: `returns(Name,Var)`

The result of the foreign function that implements the Prolog predicate `Name` is unified with the Prolog variable `Var`. Cannot be used without `foreign/1` or `foreign/2`.

needs_ciao_ctx/1: PROPERTY

Usage: `needs_ciao_ctx(Name)`

The foreign function which implementes the predicate `Name` needs a `ciao_ctx` as its first argument.

foreign/1: PROPERTY

Usage: `foreign(Name)`

The Prolog predicate `Name` is implemented using the foreign function `Name`.

foreign/2: PROPERTY

Usage: `foreign(PrologName,ForeignName)`

The Prolog predicate `PrologName` is implemented using the foreign function `ForeignName`.

foreign_low/1: PROPERTY

Usage: `foreign_low(Name)`

The Prolog predicate `Name` is implemented using the function `Name`. The implementation is not a common C one, but it accesses directly the internal Ciao Prolog data structures and functions, and therefore no glue code is generated for it.

foreign_low/2: PROPERTY

Usage: `foreign_low(PrologName,ForeignName)`

The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied.

106.3 Documentation on internals

use_foreign_source/1: DECLARATION

Usage: :- `use_foreign_source(Files).`

`Files` is the (list of) foreign file(s) that will be linked with the glue-code file. If the file(s) do(es) not have extension, then the '.c' extension will be automatically added

- *The following properties should hold at call time:*

  `Files` **is an atom or a list of atoms.** (`basic_props:atm_or_atm_list/1`)

use_foreign_gluecode_header/1: DECLARATION

Usage: :- `use_foreign_gluecode_header(Files).`

`Files` is the (list of) alterantive headers that will be included in the glue-code file (instead of the default `ciao_gluecode.h`).

- *The following properties should hold at call time:*

  `Files` **is an atom or a list of atoms.** (`basic_props:atm_or_atm_list/1`)

use_foreign_library/1: DECLARATION

Usage: :- `use_foreign_library(Libs)`.

`Libs` **is the (list of) external library(es) needed to link the C files. Only the short name of the library (i.e., what would follow the** `-l` **in the linker is needed.**

– *The following properties should hold at call time:*

`Libs` **is an atom or a list of atoms.** (`basic_props:atm_or_atm_list/1`)

use_foreign_library/2: DECLARATION

Usage: :- `use_foreign_library(OsArch,Libs)`.

`Libs` **are the OS and architecture dependant libraries.**

– *The following properties should hold at call time:*

`OsArch` **is an atom.** (`basic_props:atm/1`)

`Libs` **is an atom or a list of atoms.** (`basic_props:atm_or_atm_list/1`)

extra_compiler_opts/1: DECLARATION

Usage: :- `extra_compiler_opts(Opts)`.

`Opts` **is the list of additional compiler options (e.g., optimization options) that will be used during the compilation.**

– *The following properties should hold at call time:*

`Opts` **is an atom or a list of atoms.** (`basic_props:atm_or_atm_list/1`)

extra_compiler_opts/2: DECLARATION

Usage: :- `extra_compiler_opts(OsArch,Opts)`.

`Opts` **are the OS and architecture dependant additional compiler options.**

– *The following properties should hold at call time:*

`OsArch` **is an atom.** (`basic_props:atm/1`)

`Opts` **is an atom or a list of atoms.** (`basic_props:atm_or_atm_list/1`)

use_compiler/1: DECLARATION

Usage: :- `use_compiler(Compiler)`.

`Compiler` **is the compiler to use in this file. When this option is used, the default (Ciao-provided) compiler options are not used; those specified in** `extra_compiler_options` **are used instead.**

– *The following properties should hold at call time:*

`Compiler` **is an atom.** (`basic_props:atm/1`)

use_compiler/2: DECLARATION

Usage: :- `use_compiler(OsArch,Compiler)`.

`Compiler` **is the compiler to use in this file when compiling for the architecture** `OsArch`**. The option management is the same as in** `use_compiler/2`**.**

– *The following properties should hold at call time:*

`OsArch` **is an atom.** (`basic_props:atm/1`)

`Compiler` **is an atom.** (`basic_props:atm/1`)

extra_linker_opts/1: DECLARATION

Usage: :- `extra_linker_opts(Opts)`.

`Opts` **is the list of additional linker options that will be used during the linkage.**

– *The following properties should hold at call time:*

`Opts` **is an atom or a list of atoms.** (`basic_props:atm_or_atm_list/1`)

extra_linker_opts/2: DECLARATION

Usage: :- `extra_linker_opts(OsArch,Opts)`.

`Opts` **are the OS and architecture dependant additional linker options.**

– *The following properties should hold at call time:*

`OsArch` **is an atom.** (`basic_props:atm/1`)

`Opts` **is an atom or a list of atoms.** (`basic_props:atm_or_atm_list/1`)

use_linker/1: DECLARATION

Usage: :- `use_linker(Linker)`.

`Linker` **is the linker to use in this file. When this option is used, the default (Ciao-provided) linker options are not used; those specified in** `extra_linker_options/1` **are used instead.**

– *The following properties should hold at call time:*

`Linker` **is an atom.** (`basic_props:atm/1`)

use_linker/2: DECLARATION

Usage: :- `use_linker(OsArch,Linker)`.

`Compiler` **is the linker to use in this file when compiling for the architecture** `OsArch`**. The option management is the same as in** `use_compiler/2`**.**

– *The following properties should hold at call time:*

`OsArch` **is an atom.** (`basic_props:atm/1`)

`Linker` **is an atom.** (`basic_props:atm/1`)

106.4 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

107 Utilities for on-demand compilation of foreign files

Author(s): Manuel Carro, Jose F. Morales.

This module provides two predicates which give the user information regarding how to compile external (C) files in order to link them with the Ciao engine at runtime.

These predicates are not intended to be called directly by the end-user. Instead, a tool or module whose aim is generating dynamically loadable files from source files should use the predicates in this file in order to find out what are the proper compiler and linker to use, and which options must be passed to them in the current architecture.

107.1 Usage and interface

- **Library usage:**
  `:- use_module(library(foreign_compilation)).`
- **Exports:**
  - *Predicates:*
    `compiler_and_opts/2`, `linker_and_opts/2`.

107.2 Documentation on exports

compiler_and_opts/2: PREDICATE

Usage: `compiler_and_opts(Compiler,Opts)`

`CC` is the C compiler used to compile foreign code (including gluecode), using options `Opts`.

- *Call and exit should be compatible with:*
  `Compiler` is an atom. (`basic_props:atm/1`)
  `Opts` is a list of `atm`s. (`basic_props:list/2`)

linker_and_opts/2: PREDICATE

Usage: `linker_and_opts(LD,Opts)`

`LD` is the linker program used to link foreign code, using options `Opts`.

- *Call and exit should be compatible with:*
  `LD` is an atom. (`basic_props:atm/1`)
  `Opts` is a list of `atm`s. (`basic_props:list/2`)

107.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `parse_shell_args`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

108 PART V - Compatibility

Author(s): The Ciao Development Team.

This part documents libraries which provide tighter compatibility with the *ISO-Prolog* standard (for features not covered in the basic Ciao libraries), as well as additional predicates and functionalities that, despite not being in the ISO standard, are present in popular Prolog implementations.

109 Classic Prolog

Author(s): Daniel Cabeza, Jose F. Morales, Manuel Hermenegildo.

This package loads the predicates and features that classical Prolog implementations provide by default (the 'built-ins'). This includes the ISO-Prolog predicates and features as well as others that are de-facto standard built-ins in most Prolog implementations. Consult the `classic_predicates` **module for the listing of the concrete predicates that are available and the Ciao libraries they come from (see pointer below). Apart from these, the features defined in Chapter 60 [Definite Clause Grammars], page 303 and Chapter 118 [Enabling operators at run-time], page 531 are also activated.**

The loading of this package can be controlled as described in the *Library usage* **below and in Chapter 25 [The module system], page 127.**

109.1 Usage and interface

- **Library usage:**

  All these modules, packages, and predicates are included by default in modules starting with a `module/2` **declaration or user files without a starting** `use_package/1` **declaration. In the Ciao shell, they are loaded by default when no** `~/.ciaorc` **exists.**

  Note that `:- module(`*modulename*`,`*exports*`)` **is equivalent to** `:- module(`*modulename*`,`*exports*`,[default])`.

  If you do not want these predicates/features loaded for a given file (e.g., for purity considerations, to aid analysis, to help making the executable smaller, etc.) you can ask for this explicitly using `:- module(`*modulename*`,`*exports*`,[])`. **The same can be achieved in a user file using** `:- use_package([])` **and in the top level by defining** `~/.ciaorc` **(which can then include whatever alternative prelude is preferred).**
- **New operators defined:**

  `-->/2` [**1200,xfx**], `|/2` [**1105,xfy**], `concurrent/1` [**1150,fx**], `data/1` [**1150,fx**], `dynamic/1` [**1150,fx**].
- **Implicit imports:**
  - *System library modules:*

    `operators`, `dcg_phrase_rt`, `datafacts_rt`, `dynamic_rt`, `classic_predicates`.
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `runtime_ops`, `dcg`, `dcg/dcg_phrase`, `dynamic`, `datafacts`.

110 Classic Prolog predicates

Author(s): Daniel Cabeza, Jose F. Morales, Manuel Hermenegildo.

This module reexports the predicates required by the classical Prolog package.

110.1 Usage and interface

- **Library usage:**
  `:- use_module(library(classic/classic_predicates)).`

110.2 Documentation on exports

op/3: (UNDOC_REEXPORT)
Imported from `operators` (see the corresponding documentation for details).

current_op/3: (UNDOC_REEXPORT)
Imported from `operators` (see the corresponding documentation for details).

member/2: (UNDOC_REEXPORT)
Imported from `lists` (see the corresponding documentation for details).

append/3: (UNDOC_REEXPORT)
Imported from `lists` (see the corresponding documentation for details).

delete/3: (UNDOC_REEXPORT)
Imported from `lists` (see the corresponding documentation for details).

select/3: (UNDOC_REEXPORT)
Imported from `lists` (see the corresponding documentation for details).

nth/3: (UNDOC_REEXPORT)
Imported from `lists` (see the corresponding documentation for details).

last/2: (UNDOC_REEXPORT)
Imported from `lists` (see the corresponding documentation for details).

reverse/2: (UNDOC_REEXPORT)
Imported from `lists` (see the corresponding documentation for details).

length/2: (UNDOC_REEXPORT)
Imported from `lists` (see the corresponding documentation for details).

use_module/1: (UNDOC_REEXPORT)
Imported from `compiler` (see the corresponding documentation for details).

use_module/2: (UNDOC_REEXPORT)
Imported from `compiler` (see the corresponding documentation for details).

ensure_loaded/1: (UNDOC_REEXPORT)
Imported from `compiler` (see the corresponding documentation for details).

^/2: (UNDOC_REEXPORT)
Imported from `aggregates` (see the corresponding documentation for details).

findnsols/5: (UNDOC_REEXPORT)
Imported from `aggregates` (see the corresponding documentation for details).

findnsols/4: (UNDOC_REEXPORT)
Imported from `aggregates` (see the corresponding documentation for details).

findall/4: (UNDOC_REEXPORT)
Imported from `aggregates` (see the corresponding documentation for details).

findall/3: (UNDOC_REEXPORT)
Imported from `aggregates` (see the corresponding documentation for details).

bagof/3: (UNDOC_REEXPORT)
Imported from `aggregates` (see the corresponding documentation for details).

setof/3: (UNDOC_REEXPORT)
Imported from `aggregates` (see the corresponding documentation for details).

read_option/1: (UNDOC_REEXPORT)
Imported from `read` (see the corresponding documentation for details).

second_prompt/2: (UNDOC_REEXPORT)
Imported from `read` (see the corresponding documentation for details).

read_top_level/3: (UNDOC_REEXPORT)
Imported from `read` (see the corresponding documentation for details).

read_term/3: (UNDOC_REEXPORT)
Imported from `read` (see the corresponding documentation for details).

read_term/2: (UNDOC_REEXPORT)
Imported from `read` (see the corresponding documentation for details).

read/2: (UNDOC_REEXPORT)
Imported from `read` (see the corresponding documentation for details).

read/1: (UNDOC_REEXPORT)
Imported from `read` (see the corresponding documentation for details).

write_attribute/1: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

printable_char/1: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

prettyvars/1: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

numbervars/3: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

portray_clause/1: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

portray_clause/2: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

printq/1: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

printq/2: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

print/1: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

print/2: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

write_canonical/1: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

write_canonical/2: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

writeq/1: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

writeq/2: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

write/1: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

write/2: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

write_option/1: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

write_term/2: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

write_term/3: (UNDOC_REEXPORT)
Imported from `write` (see the corresponding documentation for details).

put_char/2: (UNDOC_REEXPORT)
Imported from `iso_char` (see the corresponding documentation for details).

put_char/1: (UNDOC_REEXPORT)
Imported from `iso_char` (see the corresponding documentation for details).

peek_char/2: (UNDOC_REEXPORT)
Imported from `iso_char` (see the corresponding documentation for details).

peek_char/1: (UNDOC_REEXPORT)
Imported from `iso_char` (see the corresponding documentation for details).

get_char/2: (UNDOC_REEXPORT)
Imported from `iso_char` (see the corresponding documentation for details).

get_char/1: (UNDOC_REEXPORT)
Imported from `iso_char` (see the corresponding documentation for details).

char_codes/2: (UNDOC_REEXPORT)
Imported from `iso_char` (see the corresponding documentation for details).

number_chars/2: (UNDOC_REEXPORT)
Imported from `iso_char` (see the corresponding documentation for details).

atom_chars/2: (UNDOC_REEXPORT)
Imported from `iso_char` (see the corresponding documentation for details).

char_code/2: (UNDOC_REEXPORT)
Imported from `iso_char` (see the corresponding documentation for details).

unify_with_occurs_check/2: (UNDOC_REEXPORT)
Imported from `iso_misc` (see the corresponding documentation for details).

sub_atom/5: (UNDOC_REEXPORT)
Imported from `iso_misc` (see the corresponding documentation for details).

compound/1: (UNDOC_REEXPORT)
Imported from `iso_misc` (see the corresponding documentation for details).

call_det/2: (UNDOC_REEXPORT)
Imported from `iso_misc` (see the corresponding documentation for details).

forall/2: (UNDOC_REEXPORT)
Imported from `iso_misc` (see the corresponding documentation for details).

once/1: (UNDOC_REEXPORT)
Imported from `iso_misc` (see the corresponding documentation for details).

format_control/1: (UNDOC_REEXPORT)
Imported from `format` (see the corresponding documentation for details).

format_to_string/3: (UNDOC_REEXPORT)
Imported from `format` (see the corresponding documentation for details).

sformat/3: (UNDOC_REEXPORT)
Imported from `format` (see the corresponding documentation for details).

format/3: (UNDOC_REEXPORT)
Imported from `format` (see the corresponding documentation for details).

format/2: (UNDOC_REEXPORT)
Imported from `format` (see the corresponding documentation for details).

keypair/1: (UNDOC_REEXPORT)
Imported from `sort` (see the corresponding documentation for details).

keylist/1: (UNDOC_REEXPORT)
Imported from `sort` (see the corresponding documentation for details).

keysort/2: (UNDOC_REEXPORT)
Imported from `sort` (see the corresponding documentation for details).

sort/2: (UNDOC_REEXPORT)
Imported from `sort` (see the corresponding documentation for details).

between/3: (UNDOC_REEXPORT)
Imported from `between` (see the corresponding documentation for details).

find_executable/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

get_home/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

using_windows/0: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

cyg2win_a/3: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

no_swapslash/3: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

cyg2win/3: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

winpath_c/3: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

winpath/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

winpath/3: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

fd_close/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

fd_dup/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

system/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

system/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

shell/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

shell/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

shell/0: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

get_numcores/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

get_grnam/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

get_pwnam/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

get_gid/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

get_uid/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

get_pid/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

kill/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

wait/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

pause/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

dev_null/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

get_tmp_dir/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

system_error_report/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

make_directory/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

make_directory/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

rename_file/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

delete_file/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

delete_directory/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

set_exec_mode/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

chmod/3: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

chmod/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

fmode/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

touch/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

modif_time0/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

modif_time/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

file_properties/6: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

file_property/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

file_exists/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

file_exists/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

mktemp_in_tmp/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

mktemp/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

directory_files/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

cd/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

working_directory/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

umask/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

current_executable/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

current_host/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

extract_paths/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

current_env/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

del_env/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

set_env/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

setenvstr/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

getenvstr/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

copy_file/3: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

copy_file/2: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

copy_options/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

copy_option/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

datime/9: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

datime_struct/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

datime/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

now/1: (UNDOC_REEXPORT)
Imported from `system` (see the corresponding documentation for details).

close_file/1: (UNDOC_REEXPORT)
Imported from `dec10_io` (see the corresponding documentation for details).

told/0: (UNDOC_REEXPORT)
Imported from `dec10_io` (see the corresponding documentation for details).

telling/1: (UNDOC_REEXPORT)
Imported from `dec10_io` (see the corresponding documentation for details).

tell/1: (UNDOC_REEXPORT)
Imported from `dec10_io` (see the corresponding documentation for details).

seen/0: (UNDOC_REEXPORT)
Imported from `dec10_io` (see the corresponding documentation for details).

seeing/1: (UNDOC_REEXPORT)
Imported from `dec10_io` (see the corresponding documentation for details).

see/1: (UNDOC_REEXPORT)
Imported from `dec10_io` (see the corresponding documentation for details).

current_key/2: (UNDOC_REEXPORT)
Imported from `old_database` (see the corresponding documentation for details).

recorded/3: (UNDOC_REEXPORT)
Imported from `old_database` (see the corresponding documentation for details).

recordz/3: (UNDOC_REEXPORT)
Imported from `old_database` (see the corresponding documentation for details).

recorda/3: (UNDOC_REEXPORT)
Imported from `old_database` (see the corresponding documentation for details).

ttydisplay_string/1: (UNDOC_REEXPORT)
Imported from `ttyout` (see the corresponding documentation for details).

ttyskipeol/0: (UNDOC_REEXPORT)
Imported from `ttyout` (see the corresponding documentation for details).

ttydisplayq/1: (UNDOC_REEXPORT)
Imported from `ttyout` (see the corresponding documentation for details).

ttydisplay/1: (UNDOC_REEXPORT)

Imported from `ttyout` (see the corresponding documentation for details).

ttyflush/0: (UNDOC_REEXPORT)

Imported from `ttyout` (see the corresponding documentation for details).

ttytab/1: (UNDOC_REEXPORT)

Imported from `ttyout` (see the corresponding documentation for details).

ttyskip/1: (UNDOC_REEXPORT)

Imported from `ttyout` (see the corresponding documentation for details).

ttyput/1: (UNDOC_REEXPORT)

Imported from `ttyout` (see the corresponding documentation for details).

ttynl/0: (UNDOC_REEXPORT)

Imported from `ttyout` (see the corresponding documentation for details).

ttyget1/1: (UNDOC_REEXPORT)

Imported from `ttyout` (see the corresponding documentation for details).

ttyget/1: (UNDOC_REEXPORT)

Imported from `ttyout` (see the corresponding documentation for details).

110.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `aggregates`, `read`, `write`, `operators`, `iso_char`, `iso_misc`, `format`, `lists`, `sort`, `between`, `compiler`, `system`, `dec10_io`, `old_database`, `ttyout`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

111 Stricter ISO-Prolog package

Author(s): The Ciao Development Team.

This package enables stricter ISO-Prolog conformance by default in the modules that include it. In particular, it includes the modules and packages listed in the *Usage and interface* **box below. Note that this package is also included when loading the Chapter 109 [Classic Prolog], page 493 package (either explicitly or implicity by using a** `module/2` **declaration). On the other hand, using this package by itself, i.e.:** `:- module(..., ..., [iso_strict]).` *approximates* **a strictly conforming mode, i.e., a mode which rejects uses of non-ISO features (except for those that are built-in in Ciao, such as the module system).**

Also note that library predicates that correspond to those in the ISO-Prolog standard are marked accordingly throughout the manuals, and differences between the Ciao and the prescribed ISO-Prolog behaviours, if any, commented appropriately.

Strict compliance with ISO is still not complete: currently there are some minor deviations in, e.g., the treatment of characters, the syntax, some of the arithmetic functions, and part of the error system. In particular, the `char_conversion/2` **directive is not implemented, since Ciao does not (yet) have a character conversion table.**

However, as mentioned in ⟨undefined⟩ [ISO-Prolog compliance versus extensibility], page ⟨undefined⟩ in the introduction, the intention of the Ciao developers is to progressively complete ISO standard support, as well as adapt to the corrigenda and reasonable extensions of the standard as they appear.

111.1 Usage and interface

- **Library usage:**

  `:- use_package(iso_strict).`

  or

  `:- module(...,...,[iso_strict]).`
- **New operators defined:**

  `-->/2` [**1200,xfx**], `|/2` [**1105,xfy**], `concurrent/1` [**1150,fx**], `data/1` [**1150,fx**], `dynamic/1` [**1150,fx**].
- **Implicit imports:**
  - *System library modules:*

    `datafacts_rt`, `dynamic_rt`, `operators`, `aggregates`, `sort`, `iso_misc`, `terms_check`, `terms_vars`, `cyclic_terms`, `format`, `compiler`, `system`, `iso_incomplete`, `iso_char`, `read`, `write`.
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `dcg`, `dynamic`, `datafacts`, `hiord`, `runtime_ops`.

112 ISO Chars

Author(s): The Ciao Development Team, Daniel Cabeza, Edison Mera (documentation), Manuel Hermenegildo (minor mods).

This module provides some basic predicates according to the ISO specification of char manipulation.

112.1 Usage and interface

- **Library usage:**
  `:- use_module(library(iso_char)).`
- **Exports:**
  - *Predicates:*
    `char_code/2`, `atom_chars/2`, `number_chars/2`, `char_codes/2`, `get_char/1`, `get_char/2`, `peek_char/1`, `peek_char/2`, `put_char/1`, `put_char/2`.

112.2 Documentation on exports

char_code/2: PREDICATE

`char_code(Char,Code)`

Succeeds iff the character code of the one char atom `Char` is `Code`.

Usage 1: ⟨• ISO •⟩

- *Call and exit should be compatible with:*
  `Code` **is an integer which is a character code.** (`basic_props:character_code/1`)
- *The following properties should hold at call time:*
  `Char` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Code` **is an integer which is a character code.** (`basic_props:character_code/1`)

Usage 2: ⟨• ISO •⟩

- *The following properties should hold at call time:*
  `Char` **is a free variable.** (`term_typing:var/1`)
  `Code` **is an integer which is a character code.** (`basic_props:character_code/1`)
- *The following properties should hold upon exit:*
  `Char` **is an atom.** (`basic_props:atm/1`)

atom_chars/2: PREDICATE

`atom_chars(Atom,Chars)`

Succeeds iff `Chars` is a list whose elements are the one-char atoms whose names are the successive characters of the name of atom `Atom`

Usage 1: ⟨• ISO •⟩

- *Call and exit should be compatible with:*
  `Chars` **is a list of `atm`s.** (`basic_props:list/2`)

- *The following properties should hold at call time:*
  `Atom` is an atom. (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Chars` is a list of `atm`s. (`basic_props:list/2`)

Usage 2: ⟨• ISO •⟩

- *The following properties should hold at call time:*
  `Atom` is a free variable. (`term_typing:var/1`)
  `Chars` is a list of `atm`s. (`basic_props:list/2`)
- *The following properties should hold upon exit:*
  `Atom` is an atom. (`basic_props:atm/1`)

number_chars/2: PREDICATE

`number_chars(Number,Chars)`

Success iff `Chars` is a list whose elements are the one-char atoms corresponding to a character sequence of `Number` which could be output

Usage 1: ⟨• ISO •⟩

- *Call and exit should be compatible with:*
  `Chars` is a list of `atm`s. (`basic_props:list/2`)
- *The following properties should hold at call time:*
  `Number` is a number. (`basic_props:num/1`)
- *The following properties should hold upon exit:*
  `Chars` is a list of `atm`s. (`basic_props:list/2`)

Usage 2: ⟨• ISO •⟩

- *The following properties should hold at call time:*
  `Number` is a free variable. (`term_typing:var/1`)
  `Chars` is a list of `atm`s. (`basic_props:list/2`)
- *The following properties should hold upon exit:*
  `Number` is a number. (`basic_props:num/1`)

char_codes/2: PREDICATE

Usage 1:

- *Call and exit should be compatible with:*
  `Arg2` is a list of `character_code`s. (`basic_props:list/2`)
- *The following properties should hold at call time:*
  `Arg1` is a list of `atm`s. (`basic_props:list/2`)
- *The following properties should hold upon exit:*
  `Arg2` is a list of `character_code`s. (`basic_props:list/2`)

Usage 2:

- *The following properties should hold at call time:*
  `Arg1` is a free variable. (`term_typing:var/1`)
  `Arg2` is a list of `character_code`s. (`basic_props:list/2`)
- *The following properties should hold upon exit:*
  `Arg1` is a list of `atm`s. (`basic_props:list/2`)

get_char/1: PREDICATE

No further documentation available for this predicate.

get_char/2: PREDICATE

`get_char(Stream,Char)`

Is true iff `Char` unifies with the next character to be input from the target `Stream`.

Usage: ⟨• ISO •⟩

- *Call and exit should be compatible with:*
  `Char` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold upon exit:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
  `Char` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold globally:*
  `Stream` **is not further instantiated.** (`basic_props:not_further_inst/2`)

peek_char/1: PREDICATE

Usage: ⟨• ISO •⟩

Similar to `peek_code/1`, but using char instead of code.

- *Call and exit should be compatible with:*
  `Arg1` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Arg1` **is an atom.** (`basic_props:atm/1`)

peek_char/2: PREDICATE

Usage: ⟨• ISO •⟩

Similar to `peek_code/2`, but using char instead of code.

- *Call and exit should be compatible with:*
  `Arg2` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold at call time:*
  `Arg1` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold upon exit:*
  `Arg1` **is an open stream.** (`stream_basic:stream/1`)
  `Arg2` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold globally:*
  `Arg1` **is not further instantiated.** (`basic_props:not_further_inst/2`)

put_char/1: PREDICATE

Usage: ⟨ • ISO • ⟩

Similar to `put_code/1`**, but using char instead of code.**

- *The following properties should hold at call time:*

  `Arg1` **is an atom.** (`basic_props:atm/1`)

put_char/2: PREDICATE

Usage: ⟨ • ISO • ⟩

Similar to `put_code/2`**, but using char instead of code.**

- *The following properties should hold at call time:*

  `Arg1` **is an open stream.** (`stream_basic:stream/1`)

  `Arg2` **is an atom.** (`basic_props:atm/1`)

- *The following properties should hold upon exit:*

  `Arg1` **is an open stream.** (`stream_basic:stream/1`)

- *The following properties should hold globally:*

  `Arg1` **is not further instantiated.** (`basic_props:not_further_inst/2`)

112.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

113 Miscellaneous ISO Prolog predicates

Author(s): Daniel Cabeza, Jose F. Morales.

This module implements some miscellaneous ISO Prolog predicates.

113.1 Usage and interface

- **Library usage:**
  `:- use_module(library(iso_misc)).`
- **Exports:**
  - *Predicates:*
    `once/1`, `forall/2`, `call_det/2`, `compound/1`, `sub_atom/5`, `unify_with_occurs_check/2`.

113.2 Documentation on exports

once/1: PREDICATE

`once(G)`

Finds the first solution of goal `G` (if any). `once/1` behaves as `call/1`, except that no further solutions are explored on backtracking.

Usage: ⟨• ISO •⟩

- *The following properties should hold at call time:*
  `G` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `once(goal)`.

forall/2: PREDICATE

`forall(Generate,Test)`

`Test` **succeeds for all solutions to** `Generate`

Usage:

- *The following properties should hold at call time:*
  `Generate` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
  `Test` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Meta-predicate **with arguments:** `forall(goal,goal)`.

call_det/2: PREDICATE

`call_det(G,Det)`

Call goal `G` and unify `Det` with `true` if no additional choice-points has been created, or `false` otherwise

Usage:

- *Call and exit should be compatible with:*
  `Det` **is an atom.** (basic_props:atm/1)
- *The following properties should hold at call time:*
  `G` **is a term which represents a goal, i.e., an atom or a structure.** (basic_props:cgoal/1)
- *The following properties should hold upon exit:*
  `Det` **is an atom.** (basic_props:atm/1)

Meta-predicate **with arguments:** `call_det(goal,?)`.

compound/1: PREDICATE

`compound(T)`

`T` **is currently instantiated to a compound term.**

Usage: ⟨• ISO •⟩

- *Call and exit should be compatible with:*
  `T` **is any term.** (basic_props:term/1)
- *The following properties should hold upon exit:*
  `T` **is any term.** (basic_props:term/1)
  `T` **is a compound term.** (basic_props:struct/1)

sub_atom/5: PREDICATE

`sub_atom(Atom,Before,Length,After,Sub_atom)`

Is true iff atom `Atom` **can be broken into three pieces,** `AtomL`**,** `Sub_atom` **and** `AtomR` **such that** `Before` **is the number of characters of the name of** `AtomL`**,** `Length` **is the number of characters of the name of** `Sub_atom` **and** `After` **is the number of characters of the name of** `AtomR`

Usage: ⟨• ISO •⟩

- *Call and exit should be compatible with:*
  `Before` **is an integer.** (basic_props:int/1)
  `Length` **is an integer.** (basic_props:int/1)
  `After` **is an integer.** (basic_props:int/1)
  `Sub_atom` **is an atom.** (basic_props:atm/1)
- *The following properties should hold at call time:*
  `Atom` **is an atom.** (basic_props:atm/1)
- *The following properties should hold upon exit:*
  `Before` **is an integer.** (basic_props:int/1)
  `Length` **is an integer.** (basic_props:int/1)
  `After` **is an integer.** (basic_props:int/1)
  `Sub_atom` **is an atom.** (basic_props:atm/1)

unify_with_occurs_check/2: PREDICATE

`unify_with_occurs_check(X,Y)`

Attempts to compute and apply a most general unifier of the two terms `X` and `Y`. Is true iff `X` and `Y` are unifiable.

Usage: ⟨• ISO •⟩

- *Call and exit should be compatible with:*
  `X` **is any term.** (`basic_props:term/1`)
  `Y` **is any term.** (`basic_props:term/1`)
- *The following properties should hold upon exit:*
  `X` **is any term.** (`basic_props:term/1`)
  `Y` **is any term.** (`basic_props:term/1`)

113.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `between`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

114 ISO Prolog compatibility layer

Author(s): The Ciao Development Team.

This module provides some additional ISO Prolog predicates whose implementation is not yet complete.

114.1 Usage and interface

- **Library usage:**
  `:- use_module(library(iso_incomplete)).`
- **Exports:**
  - *Predicates:*
    `open/4`, `close/1`, `close/2`, `stream_property/2`.
  - *Regular Types:*
    `close_options/1`, `close_option/1`, `stream_prop/1`.

114.2 Documentation on exports

absolute_file_name/2: PREDICATE

No further documentation available for this predicate.

open/4: PREDICATE

Usage:
- *Call and exit should be compatible with:*
  `Arg3` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold at call time:*
  `Arg1` **is a source name.** (`stream_basic:sourcename/1`)
  `Arg2` **is an opening mode ('read', 'write' or 'append').** (`stream_basic:io_mode/1`)
  `Arg4` **is a list of options for** `open/4`**.** (`stream_basic:open_option_list/1`)
- *The following properties should hold upon exit:*
  `Arg3` **is an open stream.** (`stream_basic:stream/1`)

close/1: PREDICATE

Usage:
- *The following properties should hold at call time:*
  `Arg1` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold upon exit:*
  `Arg1` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold globally:*
  `Arg1` **is not further instantiated.** (`basic_props:not_further_inst/2`)

close/2: PREDICATE

Usage:

- *The following properties should hold at call time:*
  `Arg1` **is an open stream.** (`stream_basic:stream/1`)
  `Arg2` **is a list of** `close_option/1`**.** (`iso_incomplete:close_options/1`)
- *The following properties should hold upon exit:*
  `Arg1` **is an open stream.** (`stream_basic:stream/1`)
  `Arg2` **is a list of** `close_option/1`**.** (`iso_incomplete:close_options/1`)
- *The following properties should hold globally:*
  `Arg1` **is not further instantiated.** (`basic_props:not_further_inst/2`)
  `Arg2` **is not further instantiated.** (`basic_props:not_further_inst/2`)

close_options/1: REGTYPE

Usage: `close_options(L)`

`L` **is a list of** `close_option/1`**.**

close_option/1: REGTYPE

Usage: `close_option(O)`

`O` **is an option for close/2:**

```
close_option(force(true)).
close_option(force(false)).
```

.

stream_property/2: PREDICATE

Usage:

- *Call and exit should be compatible with:*
  `Arg1` **is an open stream.** (`stream_basic:stream/1`)
  `Arg2` **is a valid stream property:**

```
stream_prop(input).
stream_prop(output).
stream_prop(file_name(File)) :-
    atm(File).
stream_prop(mode(Mode)) :-
    atm(Mode).
stream_prop(alias(Alias)) :-
    atm(Alias).
```

(`iso_incomplete:stream_prop/1`)

- *The following properties should hold upon exit:*
  `Arg1` **is an open stream.** (`stream_basic:stream/1`)
  `Arg2` **is a valid stream property:**

```
stream_prop(input).
stream_prop(output).
stream_prop(file_name(File)) :-
    atm(File).
stream_prop(mode(Mode)) :-
    atm(Mode).
stream_prop(alias(Alias)) :-
    atm(Alias).
```

(iso_incomplete:stream_prop/1)

stream_prop/1: REGTYPE

Usage: stream_prop(P)

P **is a valid stream property:**

```
stream_prop(input).
stream_prop(output).
stream_prop(file_name(File)) :-
    atm(File).
stream_prop(mode(Mode)) :-
    atm(Mode).
stream_prop(alias(Alias)) :-
    atm(Alias).
```

set_input/1: PREDICATE

No further documentation available for this predicate.

set_output/1: PREDICATE

No further documentation available for this predicate.

at_end_of_stream/1: PREDICATE

No further documentation available for this predicate.

get_code/2: PREDICATE

No further documentation available for this predicate.

peek_code/2: PREDICATE

No further documentation available for this predicate.

put_code/2: PREDICATE

No further documentation available for this predicate.

nl/1: PREDICATE

No further documentation available for this predicate.

tab/2: PREDICATE

No further documentation available for this predicate.

get_byte/2: PREDICATE

No further documentation available for this predicate.

peek_byte/2: PREDICATE

No further documentation available for this predicate.

put_byte/2: PREDICATE

No further documentation available for this predicate.

display/2: PREDICATE

No further documentation available for this predicate.

displayq/2: PREDICATE

No further documentation available for this predicate.

get_char/2: PREDICATE

No further documentation available for this predicate.

peek_char/2: PREDICATE

No further documentation available for this predicate.

put_char/2: PREDICATE

No further documentation available for this predicate.

read/2: PREDICATE

No further documentation available for this predicate.

read_term/3: PREDICATE

No further documentation available for this predicate.

write_term/3: PREDICATE
No further documentation available for this predicate.

write/2: PREDICATE
No further documentation available for this predicate.

writeq/2: PREDICATE
No further documentation available for this predicate.

write_canonical/2: PREDICATE
No further documentation available for this predicate.

print/2: PREDICATE
No further documentation available for this predicate.

printq/2: PREDICATE
No further documentation available for this predicate.

portray_clause/2: PREDICATE
No further documentation available for this predicate.

114.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`, `iso_char`, `read`, `write`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `datafacts`.

115 DEC-10 Prolog file IO

This module implements the support for DEC-10 Prolog style file I/O.

115.1 Usage and interface

- **Library usage:**
  `:- use_module(library(dec10_io)).`
- **Exports:**
  - *Predicates:*
    `see/1`, `seeing/1`, `seen/0`, `tell/1`, `telling/1`, `told/0`, `close_file/1`.

115.2 Documentation on exports

see/1: PREDICATE

Usage: `see(File)`

- *The following properties should hold at call time:*
  `File` **is currently instantiated to an atom.** (`term_typing:atom/1`)

seeing/1: PREDICATE

Usage: `seeing(File)`

- *The following properties should hold upon exit:*
  `File` **is currently instantiated to an atom.** (`term_typing:atom/1`)

seen/0: PREDICATE

No further documentation available for this predicate.

tell/1: PREDICATE

Usage: `tell(File)`

- *The following properties should hold at call time:*
  `File` **is currently instantiated to an atom.** (`term_typing:atom/1`)

telling/1: PREDICATE

Usage: `telling(File)`

- *The following properties should hold upon exit:*
  `File` **is currently instantiated to an atom.** (`term_typing:atom/1`)

told/0: PREDICATE

No further documentation available for this predicate.

close_file/1: PREDICATE

No further documentation available for this predicate.

115.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `datafacts`.

116 Quintus-like internal database

Author(s): The Ciao Development Team.

The predicates described in this section were introduced in early implementations of Prolog to provide efficient means of performing operations on large quantities of data. The introduction of indexed dynamic predicates have rendered these predicates obsolete, and the sole purpose of providing them is to support existing code. Their use in new code is discouraged.

These predicates store arbitrary terms in the database without interfering with the clauses which make up the program. The terms which are stored in this way can subsequently be retrieved via the key on which they were stored. Many terms may be stored on the same key, and they can be individually accessed by pattern matching. Alternatively, access can be achieved via a special identifier which uniquely identifies each recorded term and which is returned when the term is stored.

116.1 Usage and interface

- **Library usage:**
  `:- use_module(library(old_database)).`
- **Exports:**
  - *Predicates:*
    `recorda/3`, `recordz/3`, `recorded/3`, `current_key/2`.

116.2 Documentation on exports

recorda/3: PREDICATE

`recorda(Key,Term,Ref)`

The term `Term` is recorded in the internal database as the first item for the key `Key`, where `Ref` is its implementation-defined identifier. The key must be given, and only its principal functor is significant. Any uninstantiated variables in the `Term` will be replaced by new private variables, along with copies of any subgoals blocked on these variables.

Usage: `recorda(Key,Term,Ref)`

- *The following properties should hold at call time:*
  `Key` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Ref` is a free variable. (`term_typing:var/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

recordz/3: PREDICATE

`recordz(Key,Term,Ref)`

Like `recorda/3`, except that the new term becomes the *last* item for the key `Key`.

Usage: `recordz(Key,Term,Ref)`

- *The following properties should hold at call time:*
  `Key` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Ref` is a free variable. (`term_typing:var/1`)

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

recorded/3: PREDICATE

`recorded(Key,Term,Ref)`

The internal database is searched for terms recorded under the key `Key`. These terms are successively unified with `Term` in the order they occur in the database. At the same time, `Ref` is unified with the implementation-defined identifier uniquely identifying the recorded item. If the key is instantiated to a compound term, only its principal functor is significant. If the key is uninstantiated, all terms in the database are successively unified with `Term` in the order they occur.

Usage: `recorded(Key,Term,Ref)`

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

current_key/2: PREDICATE

`current_key(KeyName,KeyTerm)`

`KeyTerm` is the most general form of the key for a currently recorded term, and `KeyName` is the name of that key. This predicate can be used to enumerate in undefined order all keys for currently recorded terms through backtracking.

Usage: `current_key(Name,Key)`

- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

116.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `datafacts`.

117 C-Prolog terminal I/O

Author(s): The Ciao Development Team.

This module provides emulation of C-Prolog terminal I/O.

117.1 Usage and interface

- **Library usage:**
  `:- use_module(library(ttyout)).`
- **Exports:**
  - *Predicates:*
    `ttyget/1`, `ttyget1/1`, `ttynl/0`, `ttyput/1`, `ttyskip/1`, `ttytab/1`, `ttyflush/0`, `ttydisplay/1`, `ttydisplayq/1`, `ttyskipeol/0`, `ttydisplay_string/1`.

117.2 Documentation on exports

ttyget/1: PREDICATE

Usage: `ttyget(X)`

- *The following properties should hold upon exit:*
  `X` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  DEPRECATED. (`basic_props:deprecated/1`)

ttyget1/1: PREDICATE

No further documentation available for this predicate.

ttynl/0: PREDICATE

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  DEPRECATED. (`basic_props:deprecated/1`)

ttyput/1: PREDICATE

Usage: `ttyput(X)`

- *The following properties should hold at call time:*
  `X` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  DEPRECATED. (`basic_props:deprecated/1`)

ttyskip/1: PREDICATE

No further documentation available for this predicate.

ttytab/1: PREDICATE

No further documentation available for this predicate.

ttyflush/0: PREDICATE

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

DEPRECATED. (`basic_props:deprecated/1`)

ttydisplay/1: PREDICATE

– *The following properties hold globally:*

DEPRECATED. (`basic_props:deprecated/1`)

ttydisplayq/1: PREDICATE

– *The following properties hold globally:*

DEPRECATED. (`basic_props:deprecated/1`)

ttyskipeol/0: PREDICATE

– *The following properties hold globally:*

DEPRECATED. (`basic_props:deprecated/1`)

ttydisplay_string/1: PREDICATE

– *The following properties hold globally:*

DEPRECATED. (`basic_props:deprecated/1`)

117.3 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

118 Enabling operators at run-time

Author(s): Daniel Cabeza.

This library package allows the use of the statically defined operators of a module for the reading performed at run-time by the program that uses the module. Simply by using this package the operator definitions appearing in the module are enabled during the execution of the program.

Use with care: this is a non-modular extension. It may alter the behaviour of other modules (performing read/write operations) not using the package. This may be improved in future versions of the package (see bugs entries below).

118.1 Usage and interface

- **Library usage:**
  `:- use_package(runtime_ops).`
  or
  `:- module(...,...,[runtime_ops]).`
- **Implicit imports:**
  - *System library modules:*
    `operators`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

119 Printing dynamic predicates

Author(s): The Ciao Development Team.

This module implements predicates to pretty print the contents of the dynamic predicate database to the output stream. These predicates are introduced for compatibility with legacy C-Prolog, as well as modern Prolog implementations. Note that the predicate representation may change due to internal code expansions.

119.1 Usage and interface

- **Library usage:**
  `:- use_module(library(listing)).`
- **Exports:**
  - *Predicates:*
    `listing/0`, `listing/1`.

119.2 Documentation on exports

listing/0: PREDICATE

Usage:

Show the definition of all the dynamic predicates using `portray_clause/1`.

listing/1: PREDICATE

Usage: `listing(Spec)`

Show the definition the specified predicate given by `Spec` (`F/A`) **using** `portray_clause/1`.

- *The following properties should hold at call time:*
  `Spec` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

119.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `write`, `parse_spec`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

120 PART VI - Data structures and algorithms

Author(s): The Ciao Development Team.

This part includes some useful data structures and algorithms, used frequently in programs or in the Ciao system itself.

121 List processing

Author(s): The Ciao Development Team.

This module provides a set of predicates for list processing.

121.1 Usage and interface

- **Library usage:**
  `:- use_module(library(lists)).`
- **Exports:**
  - *Predicates:*
    `nonsingle/1`, `append/3`, `reverse/2`, `reverse/3`, `delete/3`, `delete_non_ground/3`, `select/3`, `length/2`, `nth/3`, `add_after/4`, `add_before/4`, `dlist/3`, `list_concat/2`, `list_insert/2`, `insert_last/3`, `contains_ro/2`, `contains1/2`, `nocontainsx/2`, `last/2`, `list_lookup/3`, `list_lookup/4`, `intset_insert/3`, `intset_delete/3`, `intset_in/2`, `intset_sequence/3`, `intersection/3`, `union/3`, `difference/3`, `equal_lists/2`, `list_to_list_of_lists/2`, `powerset/2`, `cross_product/2`, `sequence_to_list/2`.
  - *Properties:*
    `sublist/2`, `subordlist/2`.
  - *Regular Types:*
    `list1/2`, `list_of_lists/1`.

121.2 Documentation on exports

nonsingle/1: PREDICATE

Usage: `nonsingle(X)`

`X` **is not a singleton.**

append/3: PREDICATE

Usage: `append(Xs,Ys,Zs)`

- *Call and exit should be compatible with:*
  `Xs` **is a list.** (`basic_props:list/1`)
  `Ys` **is a list.** (`basic_props:list/1`)
  `Zs` **is a list.** (`basic_props:list/1`)

Other properties:

`append(Xs,Ys,Zs)`

`Zs` **is** `Ys` **appended to** `Xs`**.**

- *The following properties should hold upon exit:*
  `Xs` **is a list.** (`basic_props:list/1`)

`append(Xs,Ys,Zs)`

- *If the following properties hold at call time:*
  Xs **is a list.** (basic_props:list/1)
  Ys **is a list.** (basic_props:list/1)
  then the following properties should hold upon exit:
  Zs **is a list.** (basic_props:list/1)

append(Xs,Ys,Zs)
- *If the following properties hold at call time:*
  Zs **is a list.** (basic_props:list/1)
  then the following properties should hold upon exit:
  Xs **is a list.** (basic_props:list/1)
  Ys **is a list.** (basic_props:list/1)

append(Xs,Ys,Zs)
- *The following properties should hold upon exit:*
  The sharing pattern for the variables [Xs,Ys,Zs] **is** [[Xs,Ys,Zs],[Xs,Zs],[Ys,Zs]]. (native_props:mshare/2)

append(Xs,Ys,Zs)
- *If the following properties hold at call time:*
  Xs **is currently ground (it contains no variables).** (term_typing:ground/1)
  Ys **is currently ground (it contains no variables).** (term_typing:ground/1)
  then the following properties should hold upon exit:
  Zs **is currently ground (it contains no variables).** (term_typing:ground/1)

append(Xs,Ys,Zs)
- *If the following properties hold at call time:*
  Zs **is currently ground (it contains no variables).** (term_typing:ground/1)
  then the following properties should hold upon exit:
  Xs **is currently ground (it contains no variables).** (term_typing:ground/1)
  Ys **is currently ground (it contains no variables).** (term_typing:ground/1)

append(Xs,Ys,Zs)
- *The following properties hold globally:*
  append(Xs,Ys,Zs) **is side-effect** free**.** (basic_props:sideff/2)

append(Xs,Ys,Zs)
- *If the following properties hold at call time:*
  Xs **is a list.** (basic_props:list/1)
  then the following properties hold globally:
  append(Xs,Ys,Zs) **is evaluable at compile-time.** (basic_props:eval/1)

append(Xs,Ys,Zs)
- *If the following properties hold at call time:*
  Zs **is a list.** (basic_props:list/1)
  then the following properties hold globally:
  append(Xs,Ys,Zs) **is evaluable at compile-time.** (basic_props:eval/1)

reverse/2: PREDICATE

Usage: `reverse(Xs,Ys)`

Reverses the order of elements in `Xs`.

- *The following properties should hold at call time:*
  `Xs` **is a list.** (`basic_props:list/1`)
  `Ys` **is any term.** (`basic_props:term/1`)
- *The following properties should hold upon exit:*
  `Xs` **is a list.** (`basic_props:list/1`)
  `Ys` **is a list.** (`basic_props:list/1`)

Other properties:

- *The following properties hold globally:*
  `reverse(Arg1,Arg2)` **is side-effect** `free`**.** (`basic_props:sideff/2`)

`reverse(Xs,_Ys)`

- *If the following properties hold at call time:*
  `Xs` **is a list.** (`basic_props:list/1`)
  then the following properties hold globally:
  `reverse(Xs,_Ys)` **is evaluable at compile-time.** (`basic_props:eval/1`)

reverse/3: PREDICATE

Usage: `reverse(A,B,C)`

Reverse the order of elements in `A`, and append it with `B`.

Other properties:

- *The following properties hold globally:*
  `reverse(Arg1,Arg2,Arg3)` **is side-effect** `free`**.** (`basic_props:sideff/2`)

`reverse(Xs,Ys,Zs)`

- *If the following properties hold at call time:*
  `Xs` **is a list.** (`basic_props:list/1`)
  `Ys` **is a list.** (`basic_props:list/1`)
  then the following properties hold globally:
  `reverse(Xs,Ys,Zs)` **is evaluable at compile-time.** (`basic_props:eval/1`)

delete/3: PREDICATE

Usage: `delete(L1,E,L2)`

`L2` **is** `L1` **without the occurrences of** `E`**.**

- *The following properties should hold upon exit:*
  `L1` **is a list.** (`basic_props:list/1`)
  `L2` **is a list.** (`basic_props:list/1`)

Other properties:

- *The following properties hold globally:*
  `delete(Arg1,Arg2,Arg3)` **is side-effect** `free`**.** (`basic_props:sideff/2`)

`delete(L1,E,L2)`

– *If the following properties hold at call time:*
  `L1` **is currently ground (it contains no variables).** (`term_typing:ground/1`)
  `L2` **is currently ground (it contains no variables).** (`term_typing:ground/1`)
  then the following properties hold globally:
  `delete(L1,E,L2)` **is evaluable at compile-time.** (`basic_props:eval/1`)

delete_non_ground/3: PREDICATE

Usage: `delete_non_ground(L1,E,L2)`

`L2` **is** `L1` **without the occurrences of** `E`. `E` **can be a nonground term so that all the elements in** `L1` **it unifies with will be deleted**

– *The following properties should hold upon exit:*
  `L1` **is a list.** (`basic_props:list/1`)
  `L2` **is a list.** (`basic_props:list/1`)

Other properties:

– *The following properties hold globally:*
  `delete_non_ground(Arg1,Arg2,Arg3)` **is side-effect** `true`. (`basic_props:sideff/2`)

`delete_non_ground(L1,E,L2)`

– *If the following properties hold at call time:*
  `L1` **is currently ground (it contains no variables).** (`term_typing:ground/1`)
  `L2` **is currently ground (it contains no variables).** (`term_typing:ground/1`)
  then the following properties hold globally:
  `delete_non_ground(L1,E,L2)` **is evaluable at compile-time.** (`basic_props:eval/1`)

select/3: PREDICATE

Usage: `select(X,Xs,Ys)`

`Xs` **and** `Ys` **have the same elements except for one occurrence of** `X`.

Other properties:

– *The following properties hold globally:*
  `select(Arg1,Arg2,Arg3)` **is side-effect** `free`. (`basic_props:sideff/2`)

`select(X,Xs,Ys)`

– *If the following properties hold at call time:*
  `X` **is currently ground (it contains no variables).** (`term_typing:ground/1`)
  `Xs` **is currently ground (it contains no variables).** (`term_typing:ground/1`)
  then the following properties hold globally:
  `select(X,Xs,Ys)` **is evaluable at compile-time.** (`basic_props:eval/1`)

length/2: PREDICATE

Usage 1: `length(L,N)`

Computes the length of `L`.

- *The following properties should hold at call time:*
  `L` is a list. (`basic_props:list/1`)
  `N` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `L` is a list. (`basic_props:list/1`)
  `N` is an integer. (`basic_props:int/1`)

Usage 2: `length(L,N)`

Outputs `L` of length `N`.

- *The following properties should hold at call time:*
  `L` is a free variable. (`term_typing:var/1`)
  `N` is an integer. (`basic_props:int/1`)
- *The following properties should hold upon exit:*
  `L` is a list. (`basic_props:list/1`)
  `N` is an integer. (`basic_props:int/1`)

Usage 3: `length(L,N)`

Checks that `L` is of length `N`.

- *The following properties should hold at call time:*
  `L` is a list. (`basic_props:list/1`)
  `N` is an integer. (`basic_props:int/1`)
- *The following properties should hold upon exit:*
  `L` is a list. (`basic_props:list/1`)
  `N` is an integer. (`basic_props:int/1`)

Other properties:

`length(A,B)`

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
- *The following properties hold globally:*
  `length(Arg1,Arg2)` **is side-effect** `free`**.** (`basic_props:sideff/2`)

`length(L,N)`

- *If the following properties hold at call time:*
  `L` is a list. (`basic_props:list/1`)
  then the following properties hold globally:
  `length(L,N)` **is evaluable at compile-time.** (`basic_props:eval/1`)

`length(L,N)`

- *If the following properties hold at call time:*
  `N` is currently instantiated to an integer. (`term_typing:integer/1`)
  then the following properties hold globally:
  `length(L,N)` **is evaluable at compile-time.** (`basic_props:eval/1`)

nth/3: PREDICATE

`nth(N,List,Elem)`

`N` is the position in `List` of `Elem`. `N` counts from one.

Usage 1:

Unifies `Elem` and the `N`th element of `List`.

- *Call and exit should be compatible with:*
  `List` **is a list.** (`basic_props:list/1`)
  `Elem` **is any term.** (`basic_props:term/1`)
- *The following properties should hold at call time:*
  `N` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold upon exit:*
  `List` **is a list.** (`basic_props:list/1`)
  `Elem` **is any term.** (`basic_props:term/1`)

Usage 2:

Finds the positions where `Elem` is in `List`. Positions are found in ascending order.

- *Call and exit should be compatible with:*
  `List` **is a list.** (`basic_props:list/1`)
  `Elem` **is any term.** (`basic_props:term/1`)
- *The following properties should hold at call time:*
  `N` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `N` **is an integer.** (`basic_props:int/1`)
  `List` **is a list.** (`basic_props:list/1`)
  `Elem` **is any term.** (`basic_props:term/1`)

Other properties:

- *The following properties hold globally:*
  `nth(N,List,Elem)` **is side-effect** `free`. (`basic_props:sideff/2`)

`nth(N,L,E)`

- *If the following properties hold at call time:*
  `N` **is currently instantiated to an integer.** (`term_typing:integer/1`)
  then the following properties hold globally:
  `nth(N,L,E)` **is evaluable at compile-time.** (`basic_props:eval/1`)

`nth(N,L,E)`

- *If the following properties hold at call time:*
  `L` **is a list.** (`basic_props:list/1`)
  then the following properties hold globally:
  `nth(N,L,E)` **is evaluable at compile-time.** (`basic_props:eval/1`)

`nth(N,L,E)`

- *If the following properties hold at call time:*
  `N` **is an integer.** (`basic_props:int/1`)
  `L` **is a list.** (`basic_props:list/1`)
  then the following properties hold globally:
  All calls of the form `nth(N,L,E)` are deterministic. (`native_props:is_det/1`)

add_after/4: PREDICATE

Usage: `add_after(L0,E0,E,L)`

Adds element `E` after element `E0` (or at end) to list `L0` returning in `L` the new list (uses term comparison).

- *The following properties should hold at call time:*

  `L0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `E0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `E` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `L` **is a free variable.** (`term_typing:var/1`)

add_before/4: PREDICATE

Usage: `add_before(L0,E0,E,L)`

Adds element E before element E0 (or at start) to list L0 returning in L the new list (uses term comparison).

- *The following properties should hold at call time:*

  `L0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `E0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `E` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `L` **is a free variable.** (`term_typing:var/1`)

list1/2: REGTYPE

Usage: `list1(T,X)`

`X` **is a list of** `Y`**s of at least one element.**

Meta-predicate **with arguments:** `list1(pred(1),?)`.

dlist/3: PREDICATE

Usage: `dlist(List,DList,Tail)`

`List` **is the result of removing** `Tail` **from the end of** `DList` **(makes a difference list from a list).**

list_concat/2: PREDICATE

Usage: `list_concat(LL,L)`

`L` **is the concatenation of all the lists in** `LL`**.**

- *The following properties should hold at call time:*

  `LL` **is a list of** `list`**s.** (`basic_props:list/2`)

- *The following properties should hold upon exit:*

  `L` **is a list.** (`basic_props:list/1`)

list_insert/2: PREDICATE

Usage: `list_insert(List,Term)`

Adds `Term` to the end of `List` if there is no element in `List` identical to `Term`.

- *The following properties should hold at call time:*

  `Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

insert_last/3: PREDICATE

Usage: `insert_last(L0,E,L)`

Adds element `E` at end of list `L0` returning `L`.

- *The following properties should hold at call time:*

  `L0` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `E` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

contains_ro/2: PREDICATE

Usage:

Impure membership (does not instantiate a variable in its first argument.

contains1/2: PREDICATE

Usage:

First membership.

nocontainsx/2: PREDICATE

Usage: `nocontainsx(L,X)`

`X` is not identical to any element of `L`.

last/2: PREDICATE

Usage: `last(L,X)`

`X` is the last element of list `L`.

list_lookup/3: PREDICATE

Usage: `list_lookup(List,Key,Value)`

Same as `list_lookup/4`, but use `-/2` as functor.

list_lookup/4: PREDICATE

Usage: `list_lookup(List,Functor,Key,Value)`

Look up `Functor(Key,Value)` pair in variable ended key-value pair list `L` or else add it at the end.

intset_insert/3: PREDICATE

Usage: `intset_insert(A,B,Set)`

Insert the element `B` in the ordered set of numbers `A`.

intset_delete/3: PREDICATE

Usage: `intset_delete(A,B,Set)`

Delete from the ordered set `A` the element `B`.

intset_in/2: PREDICATE

Usage: `intset_in(E,Set)`

Succeds iff `E` is element of `Set`

intset_sequence/3: PREDICATE

Usage: `intset_sequence(N,L1,L2)`

Generates an ordered set of numbers from 0 to `N`-1, and append it to `L1`.

intersection/3: PREDICATE

Usage: `intersection(List1,List2,List)`

`List` **has the elements which are both in** `List1` **and** `List2`.

- *The following properties should hold at call time:*
  - `List1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `List2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `List1` **is a list.** (`basic_props:list/1`)
  - `List2` **is a list.** (`basic_props:list/1`)
- *The following properties should hold upon exit:*
  - `List` **is a list.** (`basic_props:list/1`)

union/3: PREDICATE

Usage: `union(List1,List2,List)`

`List` **has the elements which are in** `List1` **followed by the elements which are in** `List2` **but not in** `List1`.

- *The following properties should hold at call time:*
  - `List1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `List2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `List` **is a free variable.** (`term_typing:var/1`)
  - `List1` **is a list.** (`basic_props:list/1`)
  - `List2` **is a list.** (`basic_props:list/1`)
- *The following properties should hold upon exit:*
  - `List` **is a list.** (`basic_props:list/1`)

difference/3: PREDICATE

Usage: `difference(List1,List2,List)`

`List` **has the elements which are in** `List1` **but not in** `List2`.

- *The following properties should hold at call time:*
  `List1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `List2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `List` **is a free variable.** (`term_typing:var/1`)
  `List1` **is a list.** (`basic_props:list/1`)
  `List2` **is a list.** (`basic_props:list/1`)
- *The following properties should hold upon exit:*
  `List` **is a list.** (`basic_props:list/1`)

sublist/2: PROPERTY

Usage: `sublist(List1,List2)`

`List2` **contains all the elements of** `List1`.

- *If the following properties should hold at call time:*
  `List2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

subordlist/2: PROPERTY

Usage: `subordlist(List1,List2)`

`List2` **contains all the elements of** `List1` **in the same order.**

- *If the following properties should hold at call time:*
  `List2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

equal_lists/2: PREDICATE

Usage: `equal_lists(List1,List2)`

`List1` **has all the elements of** `List2`**, and vice versa.**

- *The following properties should hold at call time:*
  `List1` **is a list.** (`basic_props:list/1`)
  `List2` **is a list.** (`basic_props:list/1`)

list_to_list_of_lists/2: PREDICATE

Usage 1: `list_to_list_of_lists(List,LList)`

- *The following properties should hold at call time:*
  `List` **is a list.** (`basic_props:list/1`)
- *The following properties should hold upon exit:*
  `lists:list_of_lists(LList)` (`lists:list_of_lists/1`)

Usage 2: `list_to_list_of_lists(List,LList)`

`LList` **is the list of one element lists with elements of** `List`.

- *The following properties should hold at call time:*
  `lists:list_of_lists(LList)` (`lists:list_of_lists/1`)
- *The following properties should hold upon exit:*
  `List` **is a list.** (`basic_props:list/1`)

powerset/2: PREDICATE

Usage: `powerset(List,LList)`

`LList` is the powerset of `List`, i.e., the list of all lists which have elements of `List`. If `List` is ordered, `LList` and all its elements are ordered.

- *The following properties should hold at call time:*

  `List` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `List` **is a list.** (`basic_props:list/1`)

- *The following properties should hold upon exit:*

  `lists:list_of_lists(LList)` (`lists:list_of_lists/1`)

cross_product/2: PREDICATE

Usage: `cross_product(LList,List)`

`List` is the cartesian product of the lists in `LList`, that is, the list of lists formed with one element of each list in `LList`, in the same order.

- *The following properties should hold at call time:*

  `LList` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `lists:list_of_lists(LList)` (`lists:list_of_lists/1`)

- *The following properties should hold upon exit:*

  `lists:list_of_lists(List)` (`lists:list_of_lists/1`)

sequence_to_list/2: PREDICATE

Usage: `sequence_to_list(Sequence,List)`

`List` is the list of all elements in the (comma-separated) sequence `Sequence`. The use of this predicate is reversible.

list_of_lists/1: REGTYPE

A regular type, defined as follows:

```
list_of_lists([]).
list_of_lists([L|Xs]) :-
    list(L),
    list_of_lists(Xs).
```

member/2: (UNDOC_REEXPORT)

Imported from `basic_props` (see the corresponding documentation for details).

member/2: (UNDOC_REEXPORT)

Imported from `basic_props` (see the corresponding documentation for details).

121.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  native_props.
- *Packages:*

  prelude, initial, condcomp, assertions, assertions/assertions_basic, isomodes, hiord, nativeprops.

122 Common higher-order predicates

Author(s): Daniel Cabeza, Manuel Carro, Edison Mera, Jose F. Morales.

This library implements a few basic higher-order predicates for reducing and transforming lists.

122.1 Usage and interface

- **Library usage:**
  `:- use_module(library(hiordlib)).`
- **Exports:**
  - *Predicates:*
    `foldl/4`, `foldl/5`, `foldl/6`, `foldl/7`, `foldl/8`, `foldr/4`, `minimum/3`, `filter/3`, `partition/4`, `maplist/2`, `maplist/3`, `maplist/4`, `maplist/5`, `maplist/6`.

122.2 Documentation on exports

foldl/4: PREDICATE

The left fold family `foldl/(n+3)` **is equivalent to:**

```
foldl(P, [X11, ..., X1m], ..., [Xn1, ..., Xnm], V0, V) :-
    P(X11, ..., Xn1, V0, V1),
    ...
    P(X1m, ..., Xnm, Vm_1, V).
```

where `P` **is a predicate of arity** `n+2`**, i.e., of type** `pred(n+2)`**.**

Usage: `foldl(P,Xs,V0,V)`

Reduces (fold) `Xs` **from the left applying** `P` **and using** `V0-V` **as accumulator.**

- *Call and exit should be compatible with:*
  `P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
  `Xs` **is a list of** `terms`**.** (`basic_props:list/2`)
  `V0` **is any term.** (`basic_props:term/1`)
  `V` **is any term.** (`basic_props:term/1`)
- *The following properties should hold at call time:*
  `P` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

Meta-predicate **with arguments:** `foldl(pred(3),?,?,?)`**.**

foldl/5: PREDICATE

Usage: `foldl(P,Xs,Ys,V0,V)`

Like `foldl/4` **but applied to successive tuples from** `Xs`, `Ys`**.**

- *Call and exit should be compatible with:*
  `P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

Xs is a list of terms. (basic_props:list/2)
Ys is a list of terms. (basic_props:list/2)
V0 is any term. (basic_props:term/1)
V is any term. (basic_props:term/1)
- *The following properties should hold at call time:*
P is currently a term which is not a free variable. (term_typing:nonvar/1)

Meta-predicate **with arguments:** `foldl(pred(4),?,?,?,?)`.

foldl/6: PREDICATE

Usage: `foldl(P,Xs,Ys,Zs,V0,V)`

Like `foldl/4` **but applied to successive tuples from** `Xs`, `Ys`, `Zs`.

- *Call and exit should be compatible with:*
P is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)
Xs is a list of terms. (basic_props:list/2)
Ys is a list of terms. (basic_props:list/2)
Zs is a list of terms. (basic_props:list/2)
V0 is any term. (basic_props:term/1)
V is any term. (basic_props:term/1)
- *The following properties should hold at call time:*
P is currently a term which is not a free variable. (term_typing:nonvar/1)

Meta-predicate **with arguments:** `foldl(pred(5),?,?,?,?,?)`.

foldl/7: PREDICATE

Usage: `foldl(P,Xs,Ys,Zs,Us,V0,V)`

Like `foldl/4` **but applied to successive tuples from** `Xs`, `Ys`, `Zs`, `Us`.

- *Call and exit should be compatible with:*
P is a term which represents a goal, i.e., an atom or a structure. (basic_props:cgoal/1)
Xs is a list of terms. (basic_props:list/2)
Ys is a list of terms. (basic_props:list/2)
Zs is a list of terms. (basic_props:list/2)
Us is a list of terms. (basic_props:list/2)
V0 is any term. (basic_props:term/1)
V is any term. (basic_props:term/1)
- *The following properties should hold at call time:*
P is currently a term which is not a free variable. (term_typing:nonvar/1)

Meta-predicate **with arguments:** `foldl(pred(6),?,?,?,?,?,?)`.

foldl/8: PREDICATE

Usage: `foldl(P,Xs,Ys,Zs,Us,Ws,V0,V)`

Like `foldl/4` but applied to successive tuples from `Xs`, `Ys`, `Zs`, `Us`, `Ws`.

- *Call and exit should be compatible with:*

  `P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

  `Xs` **is a list of** `terms`**.** (`basic_props:list/2`)

  `Ys` **is a list of** `terms`**.** (`basic_props:list/2`)

  `Zs` **is a list of** `terms`**.** (`basic_props:list/2`)

  `Us` **is a list of** `terms`**.** (`basic_props:list/2`)

  `Ws` **is a list of** `terms`**.** (`basic_props:list/2`)

  `V0` **is any term.** (`basic_props:term/1`)

  `V` **is any term.** (`basic_props:term/1`)

- *The following properties should hold at call time:*

  `P` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

Meta-predicate **with arguments:** `foldl(pred(7),?,?,?,?,?,?,?)`.

foldr/4: PREDICATE

The right fold family `foldr/(n+3)` is equivalent to:

```
foldr(P, [X11, ..., X1m], ..., [Xn1, ..., Xnm], V0, V) :-
    P(X1m, ..., Xnm, V0, V1),
    ...
    P(X11, ..., Xn1, Vm_1, V).
```

where `P` is a predicate of arity `n+2`, i.e., of type `pred(n+2)`.

Note that `foldr/(n+3)` is not tail recursive. When `P(...,?,?)` is a valid calling mode, it would be possible to reorder the calls as in:

```
foldr_tail(P, [X11, ..., X1m], ..., [Xn1, ..., Xnm], V0, V) :-
    P(X11, ..., Xn1, Vm_1, V),
    ...
    P(X1m, ..., Xnm, V0, V1).
```

which is exactly like `foldl/(n+3)` but with flipped accumulator arguments. See `foldl/(n+3)` examples.

Usage: `foldr(F,Xs,V0,V)`

Reduces (fold) `Xs` from the right applying `P` and using `V0-V` as accumulator.

- *Call and exit should be compatible with:*

  `F` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

  `Xs` **is a list of** `terms`**.** (`basic_props:list/2`)

  `V0` **is any term.** (`basic_props:term/1`)

  `V` **is any term.** (`basic_props:term/1`)

- *The following properties should hold at call time:*

  `F` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `V0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

Meta-predicate **with arguments:** `foldr(pred(3),?,?,?)`.

minimum/3: PREDICATE

Usage: `minimum(List,SmallerThan,Minimum)`

`Minimum` is the smaller in the nonempty list `List` according to the relation `SmallerThan`: `SmallerThan(X, Y)` succeeds iff X is smaller than Y.

– *The following properties should hold at call time:*

  `SmallerThan` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `List` is a list. (`basic_props:list/1`)

  `SmallerThan` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Minimum` is any term. (`basic_props:term/1`)

Meta-predicate **with arguments:** `minimum(?,pred(2),?)`.

filter/3: PREDICATE

Usage: `filter(P,Xs,Ys)`

`Ys` contains all elements `X` of `Xs` such that `P(X)` holds (preserving the order)

– *Call and exit should be compatible with:*

  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Xs` is a list. (`basic_props:list/1`)

  `Ys` is a list. (`basic_props:list/1`)

– *The following properties should hold at call time:*

  `P` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Xs` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

– *The following properties should hold globally:*

  All calls of the form `filter(P,Xs,Ys)` are deterministic. (`native_props:is_det/1`)

Meta-predicate **with arguments:** `filter(pred(1),?,?)`.

partition/4: PREDICATE

Usage: `partition(P,Xs,Ys,Zs)`

`Ys` contains all elements `X` of `Xs` such that `P(X)` holds, and `Zs` all that does not (preserving the order)

– *Call and exit should be compatible with:*

  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

  `Xs` is a list. (`basic_props:list/1`)

  `Ys` is a list. (`basic_props:list/1`)

  `Zs` is a list. (`basic_props:list/1`)

– *The following properties should hold at call time:*

  `P` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Xs` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Meta-predicate **with arguments:** `partition(pred(1),?,?,?)`.

maplist/2: PREDICATE

The map list family `maplist/(n+1)` is equivalent to:

```
maplist(P, [X11, ..., X1m], ..., [Xn1, ..., Xnm]) :-
    P(X11, ..., Xn1),
    ...
    P(X1m, ..., Xnm).
```

where `P` is a predicate of arity `n`, i.e., of type `pred(n)`.

Usage: `maplist(P,Xs)`

`P(X)` succeeds for each element `X` of `Xs`

- *Call and exit should be compatible with:*
  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `Xs` is a list. (`basic_props:list/1`)
- *The following properties should hold at call time:*
  `P` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Xs` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Meta-predicate with arguments: `maplist(pred(1),?)`.

maplist/3: PREDICATE

Usage: `maplist(P,Xs,Ys)`

Like `maplist/2` but applied to successive tuples from `Xs`, `Ys`.

- *Call and exit should be compatible with:*
  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `Xs` is a list. (`basic_props:list/1`)
  `Ys` is a list. (`basic_props:list/1`)
- *The following properties should hold at call time:*
  `P` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Xs` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Meta-predicate with arguments: `maplist(pred(2),?,?)`.

maplist/4: PREDICATE

Usage: `maplist(P,Xs,Ys,Zs)`

Like `maplist/2` but applied to successive tuples from `Xs`, `Ys`, `Zs`.

- *Call and exit should be compatible with:*
  `P` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `Xs` is a list. (`basic_props:list/1`)
  `Ys` is a list. (`basic_props:list/1`)
  `Zs` is a list. (`basic_props:list/1`)
- *The following properties should hold at call time:*
  `P` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Xs` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Meta-predicate with arguments: `maplist(pred(3),?,?,?)`.

maplist/5: PREDICATE

Usage: `maplist(P,Xs,Ys,Zs,Vs)`

Like `maplist/2` but applied to successive tuples from `Xs`, `Ys`, `Zs`, `Vs`.

– *Call and exit should be compatible with:*

`P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

`Xs` **is a list.** (`basic_props:list/1`)

`Ys` **is a list.** (`basic_props:list/1`)

`Zs` **is a list.** (`basic_props:list/1`)

`Vs` **is a list.** (`basic_props:list/1`)

– *The following properties should hold at call time:*

`P` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Xs` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

Meta-predicate **with arguments:** `maplist(pred(4),?,?,?,?)`.

maplist/6: PREDICATE

Usage: `maplist(P,Xs,Ys,Zs,Vs,Ws)`

Like `maplist/2` but applied to successive tuples from `Xs`, `Ys`, `Zs`, `Vs`, `Ws`.

– *Call and exit should be compatible with:*

`P` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

`Xs` **is a list.** (`basic_props:list/1`)

`Ys` **is a list.** (`basic_props:list/1`)

`Zs` **is a list.** (`basic_props:list/1`)

`Vs` **is a list.** (`basic_props:list/1`)

`Ws` **is a list.** (`basic_props:list/1`)

– *The following properties should hold at call time:*

`P` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Xs` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

Meta-predicate **with arguments:** `maplist(pred(5),?,?,?,?,?)`.

122.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`native_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `basicmodes`, `nativeprops`, `dcg`, `fsyntax`, `hiord`, `unittestdecls`.

123 Sorting lists

Author(s): Richard A. O'Keefe (original version), The Ciao Development Team (changes and modifications).

This module implements some sorting list predicates.

123.1 Usage and interface

- **Library usage:**
  `:- use_module(library(sort)).`
- **Exports:**
  - *Predicates:*
    `sort/2`, `keysort/2`.
  - *Regular Types:*
    `keylist/1`, `keypair/1`.

123.2 Documentation on exports

sort/2: PREDICATE

`sort(List1,List2)`

The elements of `List1` are sorted into the standard order (see Chapter 33 [Comparing terms], page 165) and any identical elements are merged, yielding `List2`. The time and space complexity of this operation is at worst `O(N lg N)` where `N` is the length of `List1`.

Usage:

`List2` **is the sorted list corresponding to** `List1`.

- *Call and exit should be compatible with:*
  `List2` **is a list.** (`basic_props:list/1`)
- *The following properties should hold at call time:*
  `List1` **is a list.** (`basic_props:list/1`)
- *The following properties should hold upon exit:*
  `List2` **is a list.** (`basic_props:list/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Other properties:

`sort(A,B)`

- *If the following properties hold at call time:*
  `A` **is a list.** (`basic_props:list/1`)
  then the following properties hold globally:
  `sort(A,B)` **is evaluable at compile-time.** (`basic_props:eval/1`)

`sort(A,B)`

- *The following properties hold globally:*
  `sort(A,B)` **is side-effect** `free`. (`basic_props:sideff/2`)

keysort/2: PREDICATE

`keysort(List1,List2)`

`List1` **is sorted into order according to the value of the *keys* of its elements, yielding the list `List2`. No merging takes place. This predicate is *stable*, i.e., if an element `A` occurs before another element `B` *with the same key* in the input, then `A` will occur before `B` also in the output. The time and space complexity of this operation is at worst `O(N lg N)` where `N` is the length of `List1`.**

Usage:

`List2` **is the (key-)sorted list corresponding to `List1`.**

- *Call and exit should be compatible with:*
  `List2` **is a list of pairs of the form `Key-Value`.** (`sort:keylist/1`)
- *The following properties should hold at call time:*
  `List1` **is a list of pairs of the form `Key-Value`.** (`sort:keylist/1`)
- *The following properties should hold upon exit:*
  `List2` **is a list of pairs of the form `Key-Value`.** (`sort:keylist/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

keylist/1: REGTYPE

Usage: `keylist(L)`

`L` **is a list of pairs of the form `Key-Value`.**

keypair/1: REGTYPE

Usage: `keypair(P)`

`P` **is a pair of the form `"K-_"`, where `K` is considered the *key*.**

123.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`.

124 Lists of lists

Author(s): The Ciao Development Team.

124.1 Usage and interface

- **Library usage:**
  `:- use_module(library(llists)).`
- **Exports:**
  - *Predicates:*
    `append/2`, `flatten/2`, `collect_singletons/2`, `transpose/2`.

124.2 Documentation on exports

append/2: PREDICATE

Usage:

Concatenates a list of lists into a list.

- *Call and exit should be compatible with:*
  `Arg1` **is a list of** `lists`. (`basic_props:list/2`)
  `Arg2` **is a list.** (`basic_props:list/1`)

flatten/2: PREDICATE

Usage:

Flattens out nested lists into a list.

- *Call and exit should be compatible with:*
  `Arg2` **is a list.** (`basic_props:list/1`)
- *The following properties should hold at call time:*
  `Arg1` **is any term.** (`basic_props:term/1`)
- *The following properties should hold upon exit:*
  `Arg2` **is a list.** (`basic_props:list/1`)

collect_singletons/2: PREDICATE

Usage:

Collects in a list the singletons lists appearing in a list of lists.

- *Call and exit should be compatible with:*
  `Arg2` **is a list.** (`basic_props:list/1`)
- *The following properties should hold at call time:*
  `Arg1` **is a list of** `lists`. (`basic_props:list/2`)
- *The following properties should hold upon exit:*
  `Arg2` **is a list.** (`basic_props:list/1`)

transpose/2: PREDICATE

Usage:

Transposes a list of lists, that is, viewing it as a matrix changes rows by columns.

- *Call and exit should be compatible with:*
  `Arg2` **is a list of** `lists`. (`basic_props:list/2`)
- *The following properties should hold at call time:*
  `Arg1` **is a list of** `lists`. (`basic_props:list/2`)
- *The following properties should hold upon exit:*
  `Arg2` **is a list of** `lists`. (`basic_props:list/2`)

124.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `lists`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

125 Identity lists

Author(s): Francisco Bueno.

The operations in this module handle lists by performing equality checks via identity instead of unification.

125.1 Usage and interface

- **Library usage:**
  `:- use_module(library(idlists)).`
- **Exports:**
  - *Predicates:*
    `member_0/2`, `memberchk/2`, `list_insert/2`, `add_after/4`, `add_before/4`, `delete/3`, `subtract/3`, `union_idlists/3`.

125.2 Documentation on exports

member_0/2: PREDICATE

`member_0(X,Xs)`

True iff memberchk/2 is true.

memberchk/2: PREDICATE

`memberchk(X,Xs)`

Checks that `X` is an element of (list) `Xs`.

list_insert/2: PREDICATE

Usage: `list_insert(List,Term)`

Adds `Term` to the end of (tail-opened) `List` if there is not an element in `List` identical to `Term`.

- *The following properties should hold at call time:*

  `List` is a free variable. (`term_typing:var/1`)

  `Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

add_after/4: PREDICATE

Usage: `add_after(L0,E0,E,L)`

Adds element `E` after the first element identical to `E0` (or at end) of list `L0`, returning in `L` the new list.

- *The following properties should hold at call time:*

  `L0` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `E0` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `E` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `L` is a free variable. (`term_typing:var/1`)

add_before/4: PREDICATE

Usage: `add_before(L0,E0,E,L)`

Adds element `E` before the first element identical to `E0` (or at start) of list `L0`, returning in `L` the new list.

- *The following properties should hold at call time:*
  - `L0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `E0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `E` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `L` **is a free variable.** (`term_typing:var/1`)

delete/3: PREDICATE

Usage: `delete(List,Element,Rest)`

`Rest` **has the same elements of** `List` **except for all the occurrences of elements identical to** `Element`**.**

- *The following properties should hold at call time:*
  - `List` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `Element` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `Rest` **is a free variable.** (`term_typing:var/1`)

subtract/3: PREDICATE

Usage: `subtract(Set,Set0,Difference)`

`Difference` **has the same elements of** `Set` **except those which have an identical occurrence in** `Set0`**.**

- *The following properties should hold at call time:*
  - `Set` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `Set0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `Difference` **is a free variable.** (`term_typing:var/1`)

union_idlists/3: PREDICATE

Usage: `union_idlists(List1,List2,List)`

`List` **has the elements which are in** `List1` **but are not identical to an element in** `List2` **followed by the elements in** `List2`**.**

- *The following properties should hold at call time:*
  - `List1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `List2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `List` **is a free variable.** (`term_typing:var/1`)

125.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

126 Lists of numbers

Author(s): The Ciao Development Team.

This module implements some kinds of lists of numbers.

126.1 Usage and interface

- **Library usage:**
  `:- use_module(library(numlists)).`
- **Exports:**
  - *Predicates:*
    `get_primes/2`, `sum_list/2`, `sum_list/3`, `sum_list_of_lists/2`, `sum_list_of_lists/3`.
  - *Regular Types:*
    `intlist/1`, `numlist/1`.

126.2 Documentation on exports

get_primes/2: PREDICATE

Usage: `get_primes(N,Primes)`

Computes the `N`th first prime numbers in ascending order.

- *The following properties should hold at call time:*
  `N` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold upon exit:*
  `Primes` **is a list of integers.** (`numlists:intlist/1`)

intlist/1: REGTYPE

Usage: `intlist(X)`

`X` **is a list of integers.**

numlist/1: REGTYPE

Usage: `numlist(X)`

`X` **is a list of numbers.**

sum_list/2: PREDICATE

Usage: `sum_list(List,N)`

`N` **is the total sum of the elements of `List`.**

- *The following properties should hold at call time:*
  `List` **is a list of numbers.** (`numlists:numlist/1`)
- *The following properties should hold upon exit:*
  `N` **is a number.** (`basic_props:num/1`)

sum_list/3: PREDICATE

Usage: `sum_list(List,N0,N)`

`N` **is the total sum of the elements of** `List` **plus** `N0`**.**

- *The following properties should hold at call time:*
  `List` **is a list of numbers.** (`numlists:numlist/1`)
  `N0` **is a number.** (`basic_props:num/1`)
- *The following properties should hold upon exit:*
  `N` **is a number.** (`basic_props:num/1`)

sum_list_of_lists/2: PREDICATE

Usage: `sum_list_of_lists(Lists,N)`

`N` **is the total sum of the elements of the lists of** `Lists`**.**

- *The following properties should hold at call time:*
  `List` **is a list of** `numlists`**.** (`basic_props:list/2`)
- *The following properties should hold upon exit:*
  `N` **is a number.** (`basic_props:num/1`)

sum_list_of_lists/3: PREDICATE

Usage: `sum_list_of_lists(Lists,N0,N)`

`N` **is the total sum of the elements of the lists of** `Lists` **plus** `N0`**.**

- *The following properties should hold at call time:*
  `List` **is a list of** `numlists`**.** (`basic_props:list/2`)
  `N0` **is a number.** (`basic_props:num/1`)
- *The following properties should hold upon exit:*
  `N` **is a number.** (`basic_props:num/1`)

126.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `lists`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

127 String processing

Author(s): Daniel Cabeza.

This module provides some string-related predicates, specially for including in grammars defining strings.

127.1 Usage and interface

- **Library usage:**
  `:- use_module(library(strings)).`
- **Exports:**
  - *Predicates:*
    `whitespace/2`, `whitespace0/2`, `string/3`.

127.2 Documentation on exports

whitespace/2: PREDICATE

`whitespace(String,Rest)`

In a grammar rule, as `whitespace/0`, represents whitespace (a positive number of space (32), tab (9), newline (10) or return (13) characters). Thus, `Rest` is a proper suffix of `String` with one or more whitespace characters removed. An example of use would be:

```
attrs([]) --> ""
attrs([N|Ns]) -->
    whitespace,
    attr(N),
    attrs(Ns).
```

Usage: `whitespace(S1,S2)`

- *The following properties should hold at call time:*
  `S1` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold upon exit:*
  `S2` is a string (a list of character codes). (`basic_props:string/1`)

whitespace0/2: PREDICATE

`whitespace0(String,Rest)`

In a grammar rule, as `whitespace0/0`, represents possible whitespace (any number of space (32), tab (9), newline (10) or return (13) characters). Thus, `Rest` is `String` or a proper suffix of `String` with one or more whitespace characters removed. An example of use would be:

```
assignment(N,V) -->
    variable_name(N), whitespace0, "=", whitespace0, value(V).
```

Usage: `whitespace0(S1,S2)`

- *The following properties should hold at call time:*
  `S1` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold upon exit:*
  `S2` is a string (a list of character codes). (`basic_props:string/1`)

string/3: PREDICATE

`string(String,Head,Tail)`

In a grammar rule, as `string/1`, represents literally `String`. An example of use would be:

```
double(A) -->
    string(A),
    string(A).
```

Usage 1:

- *Call and exit should be compatible with:*

  `String` **is a string (a list of character codes).** (`basic_props:string/1`)

  `Head` **is a string (a list of character codes).** (`basic_props:string/1`)

  `Tail` **is a string (a list of character codes).** (`basic_props:string/1`)

- *The following properties should hold upon exit:*

  `String` **is a string (a list of character codes).** (`basic_props:string/1`)

  `Head` **is a string (a list of character codes).** (`basic_props:string/1`)

  `Tail` **is a string (a list of character codes).** (`basic_props:string/1`)

Usage 2: `string(A,B,C)`

- *The following properties should hold at call time:*

  `C` **is a list.** (`basic_props:list/1`)

- *The following properties should hold upon exit:*

  `A` **is a list.** (`basic_props:list/1`)

  `B` **is a list.** (`basic_props:list/1`)

127.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

128 A fuzzy search and word metric library

Author(s): Isabel Garcia-Contreras.

The `fuzzy_search` **library provides a set of predicates which calculate distance between words implementing levenshtein** (`https://en.wikipedia.org/wiki/Levenshtein_distance`) **distance and damerau-levenshtein** (`https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance`) **distance.**

This is a simple example:

```
?- levenshtein_dist(append, apend, D).

D = 1 ?

yes
?- levenshtein_dist(append, appedn, D).

D = 2 ?

yes
?-

?- damerau_lev_dist(length, lenght, D).

D = 1 ?

yes
?-
```

128.1 Usage and interface

- **Library usage:**
  `:- use_module(library(fuzzy_search)).`
- **Exports:**
  - *Predicates:*
    `levenshtein_dist/3`, `damerau_lev_dist/3`, `fuzzy_search/5`.

128.2 Documentation on exports

levenshtein_dist/3: PREDICATE

Usage: `levenshtein_dist(Word1,Word2,Distance)`

Computes the levenshtein `Distance` **between** `Word1` **and** `Word2`

- *The following properties should hold at call time:*
  - `Word1` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  - `Word2` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  - `Distance` **is a free variable.** (`term_typing:var/1`)

- *The following properties should hold upon exit:*
  `Word1` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `Word2` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `Distance` **is a number.** (`basic_props:num/1`)

damerau_lev_dist/3: PREDICATE

Usage: `damerau_lev_dist(Word1,Word2,Distance)`

Computes the Damerau-Levenshtein `Distance` **between** `Word1` **and** `Word2`

- *The following properties should hold at call time:*
  `Word1` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `Word2` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `Distance` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Word1` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `Word2` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `Distance` **is a number.** (`basic_props:num/1`)

fuzzy_search/5: PREDICATE

Usage: `fuzzy_search(Word,Generator,Metric,MaxDistance,Suggestion)`

Proposes atoms generated by `Generator` **which have** `MaxDistance` **differences with** `Word`

- *The following properties should hold at call time:*
  `Word` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `Generator` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `Metric` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `MaxDistance` **is a number.** (`basic_props:num/1`)
  `Suggestion` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Word` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `Generator` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `Metric` **is currently instantiated to an atom.** (`term_typing:atom/1`)
  `MaxDistance` **is a number.** (`basic_props:num/1`)
  `Suggestion` **is currently instantiated to an atom.** (`term_typing:atom/1`)

Meta-predicate **with arguments:** `fuzzy_search(?,pred(1),pred(3),?,?)`.

128.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `lists`, `hiordlib`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `hiord`.

129 Diff algorithm

Author(s): Isabel Garcia-Contreras, Jose F. Morales.

This module implements Eugene Myers' Greedy Difference Algorithm described in 'An O(ND) Difference Algorithm and Its Variations' [Mye86].

The implementation is parametric in the comparison predicate. This predicate has to succeed if two elements are considered equal.

129.1 Complexity

This 'greedy' algorithm that computes the number differences has complexity O((N+M)D) in both time and space. With M and N the lenghts of the input lists and D the number of changes.

129.2 Pseudocode

This pseudocode is copied from the original article, where:

- `a` **and** `b` **are the input lists of lengths** `N` **and** `M`**,**
- `k = x - y` **and is used to label diagonals,**
- `V[k]` **contains the row index of the endpoint of a furthest reaching path in diagonal k.**

```
Constant MAX [0,M+N]

Var V: Array [-MAX.. MAX] of Integer

V[1] = 0

For D = 0 to MAX Do
   For k = -D to D in steps of 2 Do
      If k = -D or (k = D and V[k-1] < V[k+1] Then
    x = V[k+1]
      Else
    x = V[k-1] + 1
    y = x - k
      While (x < N, y < M and a(x+1) = b(y+1) Do
    (x,y) = (x+1,y+1)
      V[k] = x
      If (x >= N and y >= M) Then
        Length of an SES is D
        Stop

Length of an SES is greater than MAX
```

129.3 Example

```
?- diff([a,a,b,c], [b,c,d], '=', Diff).

Diff = [del(0,a),del(0,a),ins(2,d)] ?

yes
?-
```

129.4 Patch

The patch operation applies a list of changes expressed as `diff_item/1` to obtain a new list. Note that given two lists L1, L2, if their Diff is applied to L1, it will be obtained L2:

```
?- L1 = [a,a,b,c], L2 = [b,c,d],
   diff(L1, L2, '=', Diff),
   patch(L1, Diff, L2).


Diff = [del(0,a),del(0,a),ins(2,d)],
L1 = [a,a,b,c],
L2 = [b,c,d] ?
yes
?-
```

129.5 Usage and interface

- **Library usage:**
  `:- use_module(library(diff)).`
- **Exports:**
  - *Predicates:*
    `diff/4`, `patch/3`.
  - *Regular Types:*
    `diff_item/1`.

129.6 Documentation on exports

diff/4: PREDICATE

Usage: `diff(Ls1,Ls2,Compare,Diff)`

`Diff` **are the changes needed to transform** `Ls1` **into** `Ls2`.

- *The following properties should hold at call time:*
  `Ls1` **is a list.** (`basic_props:list/1`)
  `Ls2` **is a list.** (`basic_props:list/1`)
- *The following properties should hold upon exit:*
  `Diff` **is a list of** `diff_items`. (`basic_props:list/2`)

Meta-predicate **with arguments:** `diff(?,?,pred(2),?)`.

patch/3: PREDICATE

Usage: `patch(Ls,Diff,LNew)`

Apply a list of changes (`Diff`**) onto a** `Ls`.

- *The following properties should hold at call time:*
  `Ls` **is a list.** (`basic_props:list/1`)
  `diff_item` **is a list of** `Diffs`. (`basic_props:list/2`)

– *The following properties should hold upon exit:*
  `LNew` **is a list.** (`basic_props:list/1`)

diff_item/1: REGTYPE

`diff_item(X)`

`X` **is an insertion denoted with** `ins(Pos, Elem)` **meaning that** `Elem` **has to be inserted in position** `Pos`**, or** `del(Pos, Elem)` **meaning that** `Elem` **has to be removed from position** `Pos`**. It is defined as:**

```
diff_item(ins(P,_1)) :-
    nnegint(P).
diff_item(del(P,_1)) :-
    nnegint(P).
```

Usage: `diff_item(X)`

`X` **is a single edition in a sequence (insertion or deletion)**

129.7 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*
  `lists`.
– *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `hiord`, `regtypes`.

130 Dictionaries

Author(s): The Ciao Development Team.

This module provides predicates for implementing dictionaries. Such dictionaries are currently implemented as ordered binary trees of key-value pairs.

130.1 Usage and interface

- **Library usage:**
  `:- use_module(library(dict)).`
- **Exports:**
  - *Predicates:*
    `dictionary/5`, `dic_node/2`, `dic_lookup/3`, `dic_lookup/4`, `dic_get/3`, `dic_replace/4`.
  - *Regular Types:*
    `dictionary/1`, `old_or_new/1`, `non_empty_dictionary/1`.

130.2 Documentation on exports

dictionary/1: REGTYPE

Usage: `dictionary(D)`

`D` **is a dictionary.**

dictionary/5: PREDICATE

Usage: `dictionary(D,K,V,L,R)`

The dictionary node `D` **has key** `K`, **value** `V`, **left child** `L`, **and right child** `R`.

- *The following properties should hold upon exit:*
  `D` **is a non-empty dictionary.** (`dict:non_empty_dictionary/1`)

dic_node/2: PREDICATE

Usage: `dic_node(D,N)`

`N` **is a sub-dictionary of** `D`.

- *The following properties should hold at call time:*
  `D` **is a non-empty dictionary.** (`dict:non_empty_dictionary/1`)
- *The following properties should hold upon exit:*
  `N` **is a dictionary.** (`dict:dictionary/1`)

dic_lookup/3: PREDICATE

Usage: `dic_lookup(D,K,V)`

`D` **contains value** `V` **at key** `K`. **If it was not already in** `D` **it is added.**

- *The following properties should hold upon exit:*
  `D` **is a non-empty dictionary.** (`dict:non_empty_dictionary/1`)

dic_lookup/4: PREDICATE

Usage: `dic_lookup(D,K,V,O)`

Same as `dic_lookup(D,K,V)`. `O` indicates if it was already in `D` (`old`) or not (`new`).

– *The following properties should hold upon exit:*

`D` **is a non-empty dictionary.** (`dict:non_empty_dictionary/1`)

`dict:old_or_new(O)` (`dict:old_or_new/1`)

dic_get/3: PREDICATE

Usage: `dic_get(D,K,V)`

`D` **contains value `V` at key `K`. Fails if it is not already in `D`.**

– *The following properties should hold at call time:*

`D` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`D` **is a dictionary.** (`dict:dictionary/1`)

– *The following properties should hold upon exit:*

`D` **is a non-empty dictionary.** (`dict:non_empty_dictionary/1`)

dic_replace/4: PREDICATE

Usage: `dic_replace(D,K,V,D1)`

`D` **and `D1` are identical except for the element at key `K`, which in `D1` contains value `V`, whatever has (or whether it is) in `D`.**

– *The following properties should hold at call time:*

`D` **is a dictionary.** (`dict:dictionary/1`)

`D1` **is a dictionary.** (`dict:dictionary/1`)

– *The following properties should hold upon exit:*

`D` **is a dictionary.** (`dict:dictionary/1`)

`D1` **is a dictionary.** (`dict:dictionary/1`)

old_or_new/1: REGTYPE

A regular type, defined as follows:

```
old_or_new(old).
old_or_new(new).
```

non_empty_dictionary/1: REGTYPE

Usage: `non_empty_dictionary(D)`

`D` **is a non-empty dictionary.**

130.3 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`.

131 Variable name dictionaries

Author(s): Francisco Bueno, Edison Mera, Alejandro Serrano.

131.1 Usage and interface

- **Library usage:**
  `:- use_module(library(vndict)).`
- **Exports:**
  - *Predicates:*
    `create_dict/2`, `create_pretty_dict/2`, `complete_dict/3`, `complete_vars_dict/3`, `complete_dict_alpha/3`, `complete_vars_dict_alpha/3`, `prune_dict/3`, `sort_dict/2`, `dict2varnamesl/2`, `varnamesl2dict/2`, `find_name/4`, `prettyvars/2`, `rename/2`, `vars_names_dict/3`.
  - *Regular Types:*
    `null_dict/1`, `varnamedict/1`.

131.2 Documentation on exports

null_dict/1: REGTYPE

Usage: `null_dict(D)`

`D` **is an empty dictionary.**

create_dict/2: PREDICATE

Usage: `create_dict(Term,Dict)`

`Dict` **has names for all variables in** `Term`**.**

- *The following properties should hold at call time:*
  `Term` **is any term.** (`basic_props:term/1`)
- *The following properties should hold upon exit:*
  `Dict` **is a dictionary of variable names.** (`vndict:varnamedict/1`)

create_pretty_dict/2: PREDICATE

Usage: `create_pretty_dict(Term,Dict)`

`Dict` **has names for all variables in** `Term`**. The difference with** `create_dict/2` **is that prettier names are generated**

- *The following properties should hold at call time:*
  `Term` **is any term.** (`basic_props:term/1`)
- *The following properties should hold upon exit:*
  `Dict` **is a dictionary of variable names.** (`vndict:varnamedict/1`)

complete_dict/3: PREDICATE

Usage: `complete_dict(Dict,Term,NewDict)`

`NewDict` is `Dict` augmented with the variables of `Term` not yet in `Dict`.

– *The following properties should hold at call time:*

`Dict` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
`Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
`NewDict` is a free variable. (`term_typing:var/1`)

complete_vars_dict/3: PREDICATE

Usage: `complete_vars_dict(Dict,Vars,NewDict)`

`NewDict` is `Dict` augmented with the variables of the list `Vars` not yet in `Dict`.

– *The following properties should hold at call time:*

`Dict` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
`Vars` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
`NewDict` is a free variable. (`term_typing:var/1`)

complete_dict_alpha/3: PREDICATE

Usage: `complete_dict_alpha(Dict,Term,NewDict)`

`NewDict` is `Dict` augmented with the variables of `Term` not yet in `Dict` with alphabetical names.

– *The following properties should hold at call time:*

`Dict` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
`Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
`NewDict` is a free variable. (`term_typing:var/1`)

complete_vars_dict_alpha/3: PREDICATE

Usage: `complete_vars_dict_alpha(Dict,Vars,NewDict)`

`NewDict` is `Dict` augmented with the variables of the list `Vars` not yet in `Dict` with alphabetical names.

– *The following properties should hold at call time:*

`Dict` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
`Vars` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
`NewDict` is a free variable. (`term_typing:var/1`)

prune_dict/3: PREDICATE

Usage: `prune_dict(Term,Dict,NewDict)`

`NewDict` is `Dict` reduced to just the variables of `Term`.

– *The following properties should hold at call time:*

`Term` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
`Dict` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
`NewDict` is a free variable. (`term_typing:var/1`)

sort_dict/2: PREDICATE

Usage: `sort_dict(D,Dict)`

`D` is sorted into `Dict`.

- *The following properties should hold at call time:*
  `D` is a dictionary of variable names. (`vndict:varnamedict/1`)
- *The following properties should hold upon exit:*
  `Dict` is a dictionary of variable names. (`vndict:varnamedict/1`)

dict2varnamesl/2: PREDICATE

Usage: `dict2varnamesl(Dict,VNs)`

Translates `Dict` to `VNs`.

- *The following properties should hold at call time:*
  `Dict` is a dictionary of variable names. (`vndict:varnamedict/1`)
- *The following properties should hold upon exit:*
  `VNs` is a list of `Name=Var` pairs, where `Var` is a variable and `Name` its name. (`dict_types:varnamesl/1`)

varnamesl2dict/2: PREDICATE

Usage: `varnamesl2dict(VNs,Dict)`

Translates `VNs` to `Dict`.

- *The following properties should hold at call time:*
  `VNs` is a list of `Name=Var` pairs, where `Var` is a variable and `Name` its name. (`dict_types:varnamesl/1`)
- *The following properties should hold upon exit:*
  `Dict` is a dictionary of variable names. (`vndict:varnamedict/1`)

find_name/4: PREDICATE

Usage: `find_name(Vars,Names,V,Name)`

Given that `vars_names_dict(Dict,Vars,Names)` holds, it acts as `rename(V,Dict)`, but the name of `V` is given as `Name` instead of unified with it.

prettyvars/2: PREDICATE

Usage: `prettyvars(Term,Dict)`

Give names to the variables in the term `Term` using the dictionary `Dict`. Intended to replace `prettyvars/1` in those places where is possible to get the dictionary of variables.

- *Call and exit should be compatible with:*
  `Term` is any term. (`basic_props:term/1`)
  `Dict` is a list of `Name=Var` pairs, where `Var` is a variable and `Name` its name. (`dict_types:varnamesl/1`)
- *The following properties should hold at call time:*
  `Dict` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

rename/2: PREDICATE

Usage: `rename(Term,Dict)`

Unifies each variable in `Term` with its name in `Dict`. If no name is found, a new name is created.

– *The following properties should hold at call time:*

`Dict` **is a dictionary of variable names.** (`vndict:varnamedict/1`)

varnamedict/1: REGTYPE

Usage: `varnamedict(D)`

`D` **is a dictionary of variable names.**

vars_names_dict/3: PREDICATE

Usage: `vars_names_dict(Dict,Vars,Names)`

`Vars` **is a sorted list of variables, and `Names` is a list of their names, which correspond in the same order.**

– *Call and exit should be compatible with:*

`Dict` **is a dictionary of variable names.** (`vndict:varnamedict/1`)

`Vars` **is a list.** (`basic_props:list/1`)

`Names` **is a list.** (`basic_props:list/1`)

131.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`dict_types`, `lists`, `idlists`, `terms_vars`, `sets`, `sort`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `basicmodes`, `regtypes`.

132 Extendable arrays with logarithmic access time

Author(s): Lena Flood.

This module implements extendable arrays with logarithmic access time. It has been adapted from shared code written by David Warren and Fernando Pereira.

132.1 Usage and interface

- **Library usage:**
  `:- use_module(library(arrays)).`
- **Exports:**
  - *Predicates:*
    `new_array/1`, `is_array/1`, `aref/3`, `arefa/3`, `arefl/3`, `aset/4`, `array_to_list/2`.

132.2 Documentation on exports

new_array/1: PREDICATE

Usage: `new_array(Array)`

returns an empty new array `Array`**.**

- *The following properties should hold at call time:*
  `Array` **is a free variable.** (`term_typing:var/1`)

is_array/1: PREDICATE

Usage: `is_array(Array)`

`Array` **actually is an array.**

- *The following properties should hold at call time:*
  `Array` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

aref/3: PREDICATE

Usage: `aref(Index,Array,Element)`

unifies `Element` **to** `Array`[`Index`], **or fails if** `Array`[`Index`] **has not been set.**

- *The following properties should hold at call time:*
  `Index` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Array` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

arefa/3: PREDICATE

Usage: `arefa(Index,Array,Element)`

is as `aref/3`**, except that it unifies** `Element` **with a new array if** `Array`[`Index`] **is undefined. This is useful for multidimensional arrays implemented as arrays of arrays.**

- *The following properties should hold at call time:*
  `Index` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Array` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

arefl/3: PREDICATE

Usage: `arefl(Index,Array,Element)`

is as `aref/3`, except that `Element` appears as `[]` for undefined cells. Thus, `arefl(_,_,[])` always succeeds no matter what you give in the first or second args.

– *The following properties should hold at call time:*

`Index` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Array` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

aset/4: PREDICATE

Usage: `aset(Index,Array,Element,NewArray)`

unifies `NewArray` with the result of setting `Array`[`Index`] to `Element`.

– *The following properties should hold at call time:*

`Index` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Array` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`NewArray` **is a free variable.** (`term_typing:var/1`)

array_to_list/2: PREDICATE

Usage: `array_to_list(Array,List)`

returns a `List` of pairs Index-Element of all the elements of `Array` that have been set.

– *The following properties should hold at call time:*

`Array` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`List` **is a free variable.** (`term_typing:var/1`)

132.3 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

133 Association between key and value

Author(s): Pablo Chico, Manuel Carro.

This library implements a table. It takes its name from the classical "association lists". It allows storing a set of values and a key for each value, such that the values can later be accessed through these keys. These keys could not be ground terms, but they could not be instantiated later (so, this implementation unify with '==' instead of '='). The implementation uses a dynamically changing data structure for efficiency. When there are few elements the data structure used is a list of pairs. When the number of elements stored goes beyond some number, an AVL tree is used. There is a certain level of hysteresis so that no repeated data structure conversions occur when the number of elements is close to the threshold.

133.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assoc)).`
- **Exports:**
  - *Predicates:*
    `empty_assoc/1`, `assoc_to_list/2`, `is_assoc/1`, `min_assoc/3`, `max_assoc/3`, `gen_assoc/3`, `get_assoc/3`, `get_assoc/5`, `get_next_assoc/4`, `get_prev_assoc/4`, `list_to_assoc/2`, `ord_list_to_assoc/2`, `map_assoc/2`, `map_assoc/3`, `map/3`, `foldr/4`, `put_assoc/4`, `put_assoc/5`, `add_assoc/4`, `update_assoc/5`, `del_assoc/4`, `del_min_assoc/4`, `del_max_assoc/4`.

133.2 Documentation on exports

empty_assoc/1: PREDICATE

Usage 1: `empty_assoc(Assoc)`

True if `Assoc` is an empty `assoc_table`.

- *The following properties should hold at call time:*

  `Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)

Usage 2: `empty_assoc(Assoc)`

`Assoc` **is an empty** `assoc_table`**.**

- *The following properties should hold at call time:*

  `Assoc` **is a free variable.** (`term_typing:var/1`)

  `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)

assoc_to_list/2: PREDICATE

Usage: `assoc_to_list(Assoc,L)`

Transforms `Assoc` into `L` where each pair of `L` was a association in `Assoc`.

- *The following properties should hold at call time:*
  `Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `L` **is a free variable.** (`term_typing:var/1`)
  `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)
  `L` **is a ordered list of elements of the form** `key-value`**.** (`assoc:ord_pairs/1`)

is_assoc/1: PREDICATE

Usage: `is_assoc(Assoc)`

True if `Assoc` **is an** `assoc_table`**.**

- *The following properties should hold at call time:*
  `Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)

min_assoc/3: PREDICATE

Usage: `min_assoc(Assoc,Key,Value)`

`Key` **and** `Value` **are** `key` **and** `value` **of the element with the smallest** `key` **in** `Assoc`**.**

- *The following properties should hold at call time:*
  `Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Key` **is a free variable.** (`term_typing:var/1`)
  `Value` **is a free variable.** (`term_typing:var/1`)
  `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)
  `Key` **is a valid key in a** `assoc_table`**.** (`assoc:key/1`)
  `Value` **is a valid value in a** `assoc_table`**.** (`assoc:value/1`)

max_assoc/3: PREDICATE

Usage: `max_assoc(Assoc,Key,Value)`

`Key` **and** `Value` **are the** `key` **and** `value` **of the element with the largest** `key` **in** `Assoc`**.**

- *The following properties should hold at call time:*
  `Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Key` **is a free variable.** (`term_typing:var/1`)
  `Value` **is a free variable.** (`term_typing:var/1`)
  `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)
  `Key` **is a valid key in a** `assoc_table`**.** (`assoc:key/1`)
  `Value` **is a valid value in a** `assoc_table`**.** (`assoc:value/1`)

gen_assoc/3: PREDICATE

Usage 1: `gen_assoc(K,Assoc,V)`

Enumerate matching elements of Assoc in ascending order of their keys via backtracking.

- *The following properties should hold at call time:*

  `K` is a free variable. (`term_typing:var/1`)

  `Assoc` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `V` is a free variable. (`term_typing:var/1`)

  `K` is a valid key in a `assoc_table`. (`assoc:key/1`)

  `Assoc` is a associations beetwen keys and values. (`assoc:assoc_table/1`)

  `V` is a valid value in a `assoc_table`. (`assoc:value/1`)

Usage 2: `gen_assoc(K,Assoc,V)`

Enumerate matching elements of Assoc in ascending order of their keys via backtracking whose value is `V`.

- *The following properties should hold at call time:*

  `K` is a free variable. (`term_typing:var/1`)

  `Assoc` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `V` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `K` is a valid key in a `assoc_table`. (`assoc:key/1`)

  `Assoc` is a associations beetwen keys and values. (`assoc:assoc_table/1`)

  `V` is a valid value in a `assoc_table`. (`assoc:value/1`)

get_assoc/3: PREDICATE

Usage 1: `get_assoc(K,Assoc,V)`

True if `V` is the value associated to the key `K` in the assoc_table `Assoc`.

- *The following properties should hold at call time:*

  `K` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Assoc` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `V` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `K` is a valid key in a `assoc_table`. (`assoc:key/1`)

  `Assoc` is a associations beetwen keys and values. (`assoc:assoc_table/1`)

  `V` is a valid value in a `assoc_table`. (`assoc:value/1`)

Usage 2: `get_assoc(K,Assoc,V)`

`V` is the value associated to the key `K` in the assoc_table `Assoc`.

- *The following properties should hold at call time:*

  `K` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Assoc` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `V` is a free variable. (`term_typing:var/1`)

  `K` is a valid key in a `assoc_table`. (`assoc:key/1`)

  `Assoc` is a associations beetwen keys and values. (`assoc:assoc_table/1`)

  `V` is a valid value in a `assoc_table`. (`assoc:value/1`)

get_assoc/5: PREDICATE

Usage: `get_assoc(K,Assoc,Old,NewAssoc,New)`

`NewAssoc` is an `assoc_table` identical to `Assoc` except that the value associated with `Key` is `New` instead of `Old`.

– *The following properties should hold at call time:*
  - `K` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `Old` **is a free variable.** (`term_typing:var/1`)
  - `NewAssoc` **is a free variable.** (`term_typing:var/1`)
  - `New` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `K` **is a valid key in a** `assoc_table`**.** (`assoc:key/1`)
  - `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)
  - `Old` **is a valid value in a** `assoc_table`**.** (`assoc:value/1`)
  - `NewAssoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)
  - `New` **is a valid value in a** `assoc_table`**.** (`assoc:value/1`)

get_next_assoc/4: PREDICATE

Usage: `get_next_assoc(K,Assoc,NextK,NextV)`

`NextK` **and** `NextV` **are the next** `key` **and associated** `value` **after** `K` **in** `Assoc`**.**

– *The following properties should hold at call time:*
  - `K` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `NextK` **is a free variable.** (`term_typing:var/1`)
  - `NextV` **is a free variable.** (`term_typing:var/1`)
  - `K` **is a valid key in a** `assoc_table`**.** (`assoc:key/1`)
  - `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)
  - `NextK` **is a valid key in a** `assoc_table`**.** (`assoc:key/1`)
  - `NextV` **is a valid value in a** `assoc_table`**.** (`assoc:value/1`)

get_prev_assoc/4: PREDICATE

Usage: `get_prev_assoc(K,Assoc,PrevK,PrevV)`

`PrevK` **and** `PrevV` **are the previous** `key` **and associated** `value` **after** `K` **in** `Assoc`**.**

– *The following properties should hold at call time:*
  - `K` **is a valid key in a** `assoc_table`**.** (`assoc:key/1`)
  - `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)
  - `PrevK` **is a valid key in a** `assoc_table`**.** (`assoc:key/1`)
  - `PrevV` **is a valid value in a** `assoc_table`**.** (`assoc:value/1`)

list_to_assoc/2: PREDICATE

Usage: `list_to_assoc(L,Assoc)`

Transforms `L` **into** `Assoc` **where each pair of** `L` **will be a association in** `Assoc`**.**

– *The following properties should hold at call time:*
  - `L` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  - `Assoc` **is a free variable.** (`term_typing:var/1`)
  - `L` **is a list of elements of the form** `key-value`**.** (`assoc:pairs/1`)
  - `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)

ord_list_to_assoc/2: PREDICATE

Usage: `ord_list_to_assoc(L,Assoc)`

Transforms `L`, a list of pairs (using the functor `-/2`) sorted by its first element, into the table `Assoc` where each pair of `L` will become a association in `Assoc`.

- *The following properties should hold at call time:*

  `L` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Assoc` **is a free variable.** (`term_typing:var/1`)

  `L` **is a ordered list of elements of the form `key-value`.** (`assoc:ord_pairs/1`)

  `Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)

map_assoc/2: PREDICATE

Usage: `map_assoc(Pred,Assoc)`

`Assoc` is an association tree, and for each `Key`, if `Key` is associated with `Value` in `Assoc`, Pred(Value) is true.

- *The following properties should hold at call time:*

  `Pred` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

Meta-predicate **with arguments:** `map_assoc(pred(1),?)`.

map_assoc/3: PREDICATE

Usage: `map_assoc(Pred,Assoc,NewAssoc)`

`Assoc` and `NewAssoc` are association trees of the same shape, and for each `Key`, if `Key` is associated with `Old` in `Assoc` and with `new` in `NewAssoc`, Pred(Old,New) is true.

- *The following properties should hold at call time:*

  `Pred` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `NewAssoc` **is a free variable.** (`term_typing:var/1`)

Meta-predicate **with arguments:** `map_assoc(pred(2),?,?)`.

map/3: PREDICATE

Usage: `map(Pred,Assoc1,Assoc2)`

Applies `Pred` with arity 3 to each value of the assoc_table `Assoc1` obtaining the new assoc_table `Assoc2` in which only the values can have changed.

- *The following properties should hold at call time:*

  `Pred` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Assoc1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Assoc2` **is a free variable.** (`term_typing:var/1`)

Meta-predicate **with arguments:** `map(pred(3),?,?)`.

foldr/4: PREDICATE

Usage: `foldr(Pred,Assoc,DS,NDS)`

Applies `Pred` with arity 4 to each value of the assoc_table `Assoc`. If `Pred` is satisfied, it updates the data-structure DS. Otherwise it fails.

– *The following properties should hold at call time:*

`Pred` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
`Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
`DS` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
`NDS` **is a free variable.** (`term_typing:var/1`)

Meta-predicate **with arguments:** `foldr(pred(4),?,?,?)`.

put_assoc/4: PREDICATE

Usage: `put_assoc(K,Assoc,V,NewAssoc)`

The value `V` is inserted in `Assoc` associated to the key `K` and the result is `NewAssoc`. This can be used to insert and change associations.

– *The following properties should hold at call time:*

`K` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
`Assoc` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
`V` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
`NewAssoc` **is a free variable.** (`term_typing:var/1`)
`K` **is a valid key in a** `assoc_table`**.** (`assoc:key/1`)
`Assoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)
`V` **is a valid value in a** `assoc_table`**.** (`assoc:value/1`)
`NewAssoc` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)

put_assoc/5: PREDICATE

Usage: `put_assoc(K,Assoc1,V,Assoc2,Member)`

The value `V` is inserted in `Assoc1` associated to the key `K` and the result is `Assoc2`. If the key `K` doesn't belong to the `Assoc1` then `Member` is unified with no. Otherwise, `Assoc2` is the result of substituting the association `K-OldValue` by `K-V` and `Member` is unified with yes(OldValue).

– *The following properties should hold at call time:*

`K` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
`Assoc1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
`V` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
`Assoc2` **is a free variable.** (`term_typing:var/1`)
`Member` **is a free variable.** (`term_typing:var/1`)
`K` **is a valid key in a** `assoc_table`**.** (`assoc:key/1`)
`Assoc1` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)
`V` **is a valid value in a** `assoc_table`**.** (`assoc:value/1`)
`Assoc2` **is a associations beetwen keys and values.** (`assoc:assoc_table/1`)
`Member` **is no or yes(value).** (`assoc:is_member/1`)

add_assoc/4: PREDICATE

Usage: `add_assoc(K,Assoc1,V,Assoc2)`

This is similar to `put_value/5` but `Key` must not appear in `Assoc1` (`Member` in put_value/5 is known to be no). An error is thrown otherwise.

– *The following properties should hold at call time:*

`K` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`Assoc1` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`V` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`Assoc2` **is a free variable.**	(`term_typing:var/1`)
`K` **is a valid key in a** `assoc_table`**.**	(`assoc:key/1`)
`Assoc1` **is a associations beetwen keys and values.**	(`assoc:assoc_table/1`)
`V` **is a valid value in a** `assoc_table`**.**	(`assoc:value/1`)
`Assoc2` **is a associations beetwen keys and values.**	(`assoc:assoc_table/1`)

update_assoc/5: PREDICATE

Usage: `update_assoc(K,Assoc1,V,Assoc2,OldVar)`

This is similar to `put_assoc/5` but `Key` must not appear in `Assoc1` (`Member` in put_value/5 is known to be no). An error is thrown otherwise.

– *The following properties should hold at call time:*

`K` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`Assoc1` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`V` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`Assoc2` **is a free variable.**	(`term_typing:var/1`)
`OldVar` **is a free variable.**	(`term_typing:var/1`)
`K` **is a valid key in a** `assoc_table`**.**	(`assoc:key/1`)
`Assoc1` **is a associations beetwen keys and values.**	(`assoc:assoc_table/1`)
`V` **is a valid value in a** `assoc_table`**.**	(`assoc:value/1`)
`Assoc2` **is a associations beetwen keys and values.**	(`assoc:assoc_table/1`)
`OldVar` **is a valid value in a** `assoc_table`**.**	(`assoc:value/1`)

del_assoc/4: PREDICATE

Usage: `del_assoc(K,Assoc1,V,Assoc2)`

Delete in `Assoc1` the key `K` to give `Assoc2`. If the key `K` does not belong to the `Assoc1` then `Member` is unified with no and `Assoc1` and `Assoc2` are unified. Otherwise `Assoc2` is the result of deleting the key `K` and its associated `Value`, and `Member` is unified with yes(Value).

– *The following properties should hold at call time:*

`K` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`Assoc1` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`V` **is a free variable.**	(`term_typing:var/1`)
`Assoc2` **is a free variable.**	(`term_typing:var/1`)
`K` **is a valid key in a** `assoc_table`**.**	(`assoc:key/1`)
`Assoc1` **is a associations beetwen keys and values.**	(`assoc:assoc_table/1`)
`V` **is a valid value in a** `assoc_table`**.**	(`assoc:value/1`)
`Assoc2` **is a associations beetwen keys and values.**	(`assoc:assoc_table/1`)

del_min_assoc/4: PREDICATE

Usage: `del_min_assoc(Assoc,K,V,NewAssoc)`

`Assoc` and `NewAssoc` define the same finite function except that `Assoc` associates `K` with `V` and `NewAssoc` doesn't associate `K` with any value and `K` precedes all other keys in `Assoc`.

- *The following properties should hold at call time:*

`Assoc` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`K` **is a free variable.**	(`term_typing:var/1`)
`V` **is a free variable.**	(`term_typing:var/1`)
`NewAssoc` **is a free variable.**	(`term_typing:var/1`)
`Assoc` **is a associations beetwen keys and values.**	(`assoc:assoc_table/1`)
`K` **is a valid key in a** `assoc_table`**.**	(`assoc:key/1`)
`V` **is a valid value in a** `assoc_table`**.**	(`assoc:value/1`)
`NewAssoc` **is a associations beetwen keys and values.**	(`assoc:assoc_table/1`)

del_max_assoc/4: PREDICATE

Usage: `del_max_assoc(Assoc,K,V,NewAssoc)`

`Assoc` and `NewAssoc` define the same finite function except that `Assoc` associates `K` with `V` and `NewAssoc` doesn't associate `K` with any value and `K` is preceded by all other keys in `Assoc`.

- *The following properties should hold at call time:*

`Assoc` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`K` **is a free variable.**	(`term_typing:var/1`)
`V` **is a free variable.**	(`term_typing:var/1`)
`NewAssoc` **is a free variable.**	(`term_typing:var/1`)
`Assoc` **is a associations beetwen keys and values.**	(`assoc:assoc_table/1`)
`K` **is a valid key in a** `assoc_table`**.**	(`assoc:key/1`)
`V` **is a valid value in a** `assoc_table`**.**	(`assoc:value/1`)
`NewAssoc` **is a associations beetwen keys and values.**	(`assoc:assoc_table/1`)

133.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `hiordlib`, `lists`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `basicmodes`, `hiord`, `regtypes`.

134 Key-value lists

134.1 Usage and interface

- **Library usage:**
  `:- use_module(library(keys)).`
- **Exports:**
  - *Predicates:*
    `keys_and_values/3`, `keys_and_values/4`, `key_lookup/4`.

134.2 Documentation on exports

keys_and_values/3: PREDICATE

No further documentation available for this predicate.

keys_and_values/4: PREDICATE

No further documentation available for this predicate.

key_lookup/4: PREDICATE

No further documentation available for this predicate.

134.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

135 Graphs

Author(s): Francisco Bueno.

This module implements utilities for work with graphs

135.1 Usage and interface

- **Library usage:**
  `:- use_module(library(graphs)).`
- **Exports:**
  - *Predicates:*
    `dgraph_to_ugraph/2`, `dlgraph_to_lgraph/2`, `edges_to_ugraph/2`, `edges_to_lgraph/2`.
  - *Regular Types:*
    `dgraph/1`, `dlgraph/1`.

135.2 Documentation on exports

dgraph/1: REGTYPE

`dgraph(Graph)`

A directed graph is a term `graph(V,E)` where `V` is a list of vertices and `E` is a list of edges (none necessarily sorted). Edges are pairs of vertices which are directed, i.e., `(a,b)` represents `a->b`. Two vertices `a` and `b` are equal only if `a==b`.

Usage: `dgraph(Graph)`

`Graph` is a directed graph.

dlgraph/1: REGTYPE

`dlgraph(Graph)`

A labeled directed graph is a directed graph where edges are triples of the form `(a,l,b)` where `l` is the label of the edge `(a,b)`.

Usage: `dlgraph(Graph)`

`Graph` is a directed labeled graph.

dgraph_to_ugraph/2: PREDICATE

Usage: `dgraph_to_ugraph(Graph,UGraph)`

Converts `Graph` to `UGraph`.

- *The following properties should hold at call time:*
  `Graph` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `UGraph` is a free variable. (`term_typing:var/1`)
  `Graph` is a directed graph. (`graphs:dgraph/1`)
  `UGraph` is a free variable. (`term_typing:var/1`)

– *The following properties should hold upon exit:*
  `Graph` is a directed graph. (`graphs:dgraph/1`)
  `UGraph` is an ugraph. (`ugraphs:ugraph/1`)

dlgraph_to_lgraph/2: PREDICATE

Usage: `dlgraph_to_lgraph(Graph,LGraph)`

Converts `Edges` to `LGraph`.

– *The following properties should hold at call time:*
  `Graph` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `LGraph` is a free variable. (`term_typing:var/1`)
  `Graph` is a directed labeled graph. (`graphs:dlgraph/1`)
  `LGraph` is a free variable. (`term_typing:var/1`)
– *The following properties should hold upon exit:*
  `Graph` is a directed labeled graph. (`graphs:dlgraph/1`)
  `LGraph` is a labeled graph of `term` terms. (`lgraphs:lgraph/2`)

edges_to_ugraph/2: PREDICATE

Usage: `edges_to_ugraph(Edges,UGraph)`

Converts `Graph` to `UGraph`.

– *The following properties should hold at call time:*
  `Edges` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `UGraph` is a free variable. (`term_typing:var/1`)
  `Edges` is a list of `pairs`. (`basic_props:list/2`)
  `UGraph` is a free variable. (`term_typing:var/1`)
– *The following properties should hold upon exit:*
  `Edges` is a list of `pairs`. (`basic_props:list/2`)
  `UGraph` is an ugraph. (`ugraphs:ugraph/1`)

edges_to_lgraph/2: PREDICATE

Usage: `edges_to_lgraph(Edges,LGraph)`

Converts `Edges` to `LGraph`.

– *The following properties should hold at call time:*
  `Edges` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `LGraph` is a free variable. (`term_typing:var/1`)
  `Edges` is a list of `triples`. (`basic_props:list/2`)
  `LGraph` is a free variable. (`term_typing:var/1`)
– *The following properties should hold upon exit:*
  `Edges` is a list of `triples`. (`basic_props:list/2`)
  `LGraph` is a labeled graph of `term` terms. (`lgraphs:lgraph/2`)

135.3 Documentation on internals

pair/1: REGTYPE

Usage: `pair(P)`

`P` **is a pair** `(_,_)`.

triple/1: REGTYPE

Usage: `triple(P)`

`P` **is a triple** `(_,_,_)`.

135.4 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

  `sort`, `ugraphs`, `lgraphs`.

– *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `basicmodes`, `regtypes`.

136 Unweighted graph-processing utilities

Author(s): Richard A. O'Keefe (original shared code), Mats Carlsson (adapted from original code), Francisco Bueno (modifications), Manuel Carro (modifications).

An unweighted directed graph (ugraph) is represented as a list of (vertex-neighbors) pairs, where the pairs are in standard order (as produced by keysort with unique keys) and the neighbors of each vertex are also in standard order (as produced by sort), and every neighbor appears as a vertex even if it has no neighbors itself.

An undirected graph is represented as a directed graph where for each edge `(U,V)` there is a symmetric edge `(V,U)`.

An edge `(U,V)` is represented as the term `U-V`.

A vertex can be any term. Two vertices are distinct iff they are not identical (`==/2`).

A path is represented as a list of vertices. No vertex can appear twice in a path.

136.1 Usage and interface

- **Library usage:**
  `:- use_module(library(graphs/ugraphs)).`
- **Exports:**
  - *Predicates:*
    `vertices_edges_to_ugraph/3`, `neighbors/3`, `edges/2`, `del_edges/3`, `add_edges/3`, `vertices/2`, `del_vertices/3`, `add_vertices/3`, `transpose/2`, `rooted_subgraph/3`, `point_to/3`.
  - *Regular Types:*
    `ugraph/1`.

136.2 Documentation on exports

vertices_edges_to_ugraph/3: PREDICATE

No further documentation available for this predicate.

neighbors/3: PREDICATE

Usage: `neighbors(Vertex,Graph,Neighbors)`

Is true if `Vertex` is a vertex in `Graph` and `Neighbors` are its neighbors.

- *The following properties should hold at call time:*
  `Vertex` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Graph` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Neighbors` is a free variable. (`term_typing:var/1`)

edges/2: PREDICATE

Usage: `edges(Graph,Edges)`

Unifies `Edges` with the edges in `Graph`.

- *The following properties should hold at call time:*

  `Graph` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Edges` is a free variable. (`term_typing:var/1`)

del_edges/3: PREDICATE

Usage: `del_edges(Graph1,Edges,Graph2)`

Is true if `Graph2` is `Graph1` with `Edges` removed from it.

- *The following properties should hold at call time:*

  `Graph1` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Edges` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Graph2` is a free variable. (`term_typing:var/1`)

add_edges/3: PREDICATE

Usage: `add_edges(Graph1,Edges,Graph2)`

Is true if `Graph2` is `Graph1` with `Edges` and their 'to' and 'from' vertices added to it.

- *The following properties should hold at call time:*

  `Graph1` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Edges` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Graph2` is a free variable. (`term_typing:var/1`)

vertices/2: PREDICATE

Usage: `vertices(Graph,Vertices)`

Unifies `Vertices` with the vertices in `Graph`.

- *The following properties should hold at call time:*

  `Graph` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Vertices` is a free variable. (`term_typing:var/1`)

del_vertices/3: PREDICATE

Usage: `del_vertices(Graph1,Vertices,Graph2)`

Is true if `Graph2` is `Graph1` with `Vertices` and all edges to and from `Vertices` removed from it.

- *The following properties should hold at call time:*

  `Graph1` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Vertices` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Graph2` is a free variable. (`term_typing:var/1`)

add_vertices/3: PREDICATE

Usage: `add_vertices(Graph1,Vertices,Graph2)`

Is true if `Graph2` is `Graph1` with `Vertices` added to it.

- *The following properties should hold at call time:*
  `Graph1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Vertices` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Graph2` **is a free variable.** (`term_typing:var/1`)

transpose/2: PREDICATE

Usage: `transpose(Graph,Transpose)`

Is true if `Transpose` **is the graph computed by replacing each edge** `(u,v)` **in** `Graph` **by its symmetric edge** `(v,u)`**. It can only be used one way around. The cost is O(N^2).**

- *The following properties should hold at call time:*
  `Graph` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Transpose` **is a free variable.** (`term_typing:var/1`)

rooted_subgraph/3: PREDICATE

Usage: `rooted_subgraph(Graph,Sources,SubGraph)`

`SubGraph` **is the subgraph of** `Graph` **which is reachable from** `Sources`**.**

- *The following properties should hold at call time:*
  `Graph` **is an ugraph.** (`ugraphs:ugraph/1`)
  `Sources` **is a list.** (`basic_props:list/1`)
  `SubGraph` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `SubGraph` **is an ugraph.** (`ugraphs:ugraph/1`)

point_to/3: PREDICATE

Usage: `point_to(Vertex,Graph,Point_to)`

Is true if `Point_to` **is the list of nodes which go directly to** `Vertex` **in** `Graph`**.**

- *The following properties should hold at call time:*
  `Vertex` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Graph` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Point_to` **is a free variable.** (`term_typing:var/1`)

ugraph/1: REGTYPE

Usage: `ugraph(Graph)`

`Graph` **is an ugraph.**

136.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `sets`, `sort`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `regtypes`.

137 Labeled graph-processing utilities

Author(s): Francisco Bueno.

See the comments for the `ugraphs` library.

137.1 Usage and interface

- **Library usage:**
  `:- use_module(library(graphs/lgraphs)).`
- **Exports:**
  - *Predicates:*
    `vertices_edges_to_lgraph/3`.
  - *Regular Types:*
    `lgraph/2`.

137.2 Documentation on exports

lgraph/2: REGTYPE

Usage: `lgraph(Type,Graph)`

`Graph` **is a labeled graph of** `Type` **terms.**

Meta-predicate **with arguments:** `lgraph(pred(1),?)`.

vertices_edges_to_lgraph/3: PREDICATE

`vertices_edges_to_lgraph(Vertices0,Edges,Graph)`

This one is a copy of the same procedure in library(wgraphs) except for the definition of min/3 (ah! - the polimorphism!).

It would only be needed if there are multi-edges, i.e., several edges between the same two vertices.

137.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `sort`, `sets`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `regtypes`.

138 Queues

138.1 Usage and interface

- **Library usage:**
  `:- use_module(library(queues)).`
- **Exports:**
  - *Predicates:*
    `q_empty/1`, `q_insert/3`, `q_member/2`, `q_delete/3`.

138.2 Documentation on exports

q_empty/1: PREDICATE

No further documentation available for this predicate.

q_insert/3: PREDICATE

No further documentation available for this predicate.

q_member/2: PREDICATE

No further documentation available for this predicate.

q_delete/3: PREDICATE

No further documentation available for this predicate.

138.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

139 Set operations

Author(s): Lena Flood.

This module implements set operations. Sets are just ordered lists.

139.1 Usage and interface

- **Library usage:**
  `:- use_module(library(sets)).`
- **Exports:**
  - *Predicates:*
    `insert/3`, `ord_delete/3`, `ord_member/2`, `ord_test_member/3`, `ord_subtract/3`, `ord_intersection/3`, `ord_intersection_diff/4`, `ord_intersect/2`, `ord_subset/2`, `ord_subset_diff/3`, `ord_union/3`, `ord_union_diff/4`, `ord_union_symdiff/4`, `ord_union_change/3`, `merge/3`, `ord_disjoint/2`, `setproduct/3`.

139.2 Documentation on exports

insert/3: PREDICATE

Usage: `insert(Set1,Element,Set2)`

It is true when `Set2` is `Set1` with `Element` inserted in it, preserving the order.

- *The following properties should hold at call time:*
  `Set1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Element` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Set2` **is a free variable.** (`term_typing:var/1`)

ord_delete/3: PREDICATE

Usage: `ord_delete(Set0,X,Set)`

It succeeds if `Set` is `Set0` without element `X`.

- *The following properties should hold at call time:*
  `Set0` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `X` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Set` **is a free variable.** (`term_typing:var/1`)

ord_member/2: PREDICATE

Usage: `ord_member(X,Set)`

It succeeds if `X` is member of `Set`.

- *The following properties should hold at call time:*
  `X` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Set` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

ord_test_member/3: PREDICATE

Usage: `ord_test_member(Set,X,Result)`

If `X` is member of `Set` then `Result=yes`. Otherwise `Result=no`.

- *The following properties should hold at call time:*

  `Set` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `X` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Result` is a free variable. (`term_typing:var/1`)

ord_subtract/3: PREDICATE

Usage: `ord_subtract(Set1,Set2,Difference)`

It is true when `Difference` contains all and only the elements of `Set1` which are not also in `Set2`.

- *The following properties should hold at call time:*

  `Set1` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Set2` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

ord_intersection/3: PREDICATE

Usage: `ord_intersection(Set1,Set2,Intersection)`

It is true when `Intersection` is the ordered representation of `Set1` and `Set2`, provided that `Set1` and `Set2` are ordered lists.

- *The following properties should hold at call time:*

  `Set1` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Set2` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

ord_intersection_diff/4: PREDICATE

Usage: `ord_intersection_diff(Set1,Set2,Intersect,NotIntersect)`

`Intersect` contains those elements which are both in `Set1` and `Set2`, and `NotIntersect` those which are in `Set1` but not in `Set2`.

- *The following properties should hold at call time:*

  `Set1` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Set2` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Intersect` is a free variable. (`term_typing:var/1`)

  `NotIntersect` is a free variable. (`term_typing:var/1`)

ord_intersect/2: PREDICATE

Usage: `ord_intersect(Xs,Ys)`

Succeeds when the two ordered lists have at least one element in common.

- *The following properties should hold at call time:*

  `Xs` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Ys` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

ord_subset/2: PREDICATE

Usage: `ord_subset(Xs,Ys)`

Succeeds when every element of `Xs` appears in `Ys`.

– *The following properties should hold at call time:*

`Xs` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Ys` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

ord_subset_diff/3: PREDICATE

Usage: `ord_subset_diff(Set1,Set2,Difference)`

It succeeds when every element of `Set1` appears in `Set2` and `Difference` has the elements of `Set2` which are not in `Set1`.

– *The following properties should hold at call time:*

`Set1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Set2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Difference` **is a free variable.** (`term_typing:var/1`)

ord_union/3: PREDICATE

Usage: `ord_union(Set1,Set2,Union)`

It is true when `Union` is the union of `Set1` and `Set2`. When some element occurs in both sets, `Union` retains only one copy.

– *The following properties should hold at call time:*

`Set1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Set2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

ord_union_diff/4: PREDICATE

Usage: `ord_union_diff(Set1,Set2,Union,Difference)`

It succeeds when `Union` is the union of `Set1` and `Set2`, and `Difference` is `Set2` set-minus `Set1`.

– *The following properties should hold at call time:*

`Set1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Set2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Union` **is a free variable.** (`term_typing:var/1`)

`Difference` **is a free variable.** (`term_typing:var/1`)

ord_union_symdiff/4: PREDICATE

Usage: `ord_union_symdiff(Set1,Set2,Union,Diff)`

It is true when Diff is the symmetric difference of Set1 and Set2, and Union is the union of Set1 and Set2.

– *The following properties should hold at call time:*

`Set1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Set2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Union` **is a free variable.** (`term_typing:var/1`)

`Diff` **is a free variable.** (`term_typing:var/1`)

ord_union_change/3: PREDICATE

Usage: `ord_union_change(Set1,Set2,Union)`

`Union` **is the union of** `Set1` **and** `Set2` **and** `Union` **is different from** `Set2`.

– *The following properties should hold at call time:*

`Set1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Set2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Union` **is a free variable.** (`term_typing:var/1`)

merge/3: PREDICATE

Usage: `merge(Set1,Set2,Union)`

See `ord_union/3`.

– *The following properties should hold at call time:*

`Set1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Set2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

ord_disjoint/2: PREDICATE

Usage: `ord_disjoint(Set1,Set2)`

`Set1` **and** `Set2` **have no element in common.**

– *The following properties should hold at call time:*

`Set1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Set2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

setproduct/3: PREDICATE

Usage: `setproduct(Set1,Set2,Product)`

`Product` **has all two element sets such that one element is in** `Set1` **and the other in** `set2`**, except that if the same element belongs to both, then the corresponding one element set is in** `Product`.

– *The following properties should hold at call time:*

`Set1` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Set2` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Product` **is a free variable.** (`term_typing:var/1`)

139.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`sort`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `basicmodes`.

140 Bit-coded-set operations

Author(s): Ann Mulkers (first version), Francisco Bueno (port to Ciao).

Only finite sets and complements of finite sets.

140.1 Usage and interface

- **Library usage:**
  `:- use_module(library(bitcodesets)).`
- **Exports:**
  - *Predicates:*
    `bitset_empty/1`, `bitset_equal/2`, `bitset_member/2`, `not_bitset_member/2`, `bitset_member_SoS/2`, `bitset_subset/2`, `bitset_union/3`, `bitset_union_list/2`, `bitset_union_list_list_s/4`, `bitset_intersect/3`, `bitset_subtract/3`, `bitset_subtract_list/3`, `bitset_size/2`, `bitcode_to_listofbitcode/2`, `bitcode_to_set/2`, `bitcode_to_set/3`, `bitcode_to_set_array/2`, `int_to_bitcode/2`, `lbitcode_to_llist/2`, `lbitcode_to_llistS/2`, `llist_to_lbitcode/2`, `set_to_bitcode/2`.

140.2 Documentation on exports

bitset_empty/1: PREDICATE

No further documentation available for this predicate.

bitset_equal/2: PREDICATE

No further documentation available for this predicate.

bitset_member/2: PREDICATE

No further documentation available for this predicate.

not_bitset_member/2: PREDICATE

No further documentation available for this predicate.

bitset_member_SoS/2: PREDICATE

No further documentation available for this predicate.

bitset_subset/2: PREDICATE

No further documentation available for this predicate.

bitset_union/3: PREDICATE

No further documentation available for this predicate.

bitset_union_list/2: PREDICATE

No further documentation available for this predicate.

bitset_union_list_list_s/4: PREDICATE

No further documentation available for this predicate.

bitset_intersect/3: PREDICATE

No further documentation available for this predicate.

bitset_subtract/3: PREDICATE

No further documentation available for this predicate.

bitset_subtract_list/3: PREDICATE

No further documentation available for this predicate.

bitset_size/2: PREDICATE

No further documentation available for this predicate.

bitcode_to_listofbitcode/2: PREDICATE

No further documentation available for this predicate.

bitcode_to_set/2: PREDICATE

No further documentation available for this predicate.

bitcode_to_set/3: PREDICATE

No further documentation available for this predicate.

bitcode_to_set_array/2: PREDICATE

No further documentation available for this predicate.

int_to_bitcode/2: PREDICATE

No further documentation available for this predicate.

lbitcode_to_llist/2: PREDICATE

No further documentation available for this predicate.

lbitcode_to_llistS/2: PREDICATE

No further documentation available for this predicate.

llist_to_lbitcode/2: PREDICATE

No further documentation available for this predicate.

set_to_bitcode/2: PREDICATE

No further documentation available for this predicate.

140.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `sort`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

141 Lists of sets

141.1 Usage and interface

- **Library usage:**
  `:- use_module(library(lsets)).`
- **Exports:**
  - *Predicates:*
    `merge_list_of_lists/2`, `merge_lists/3`, `merge_each/3`, `ord_member_list_of_lists/2`, `delete_var_from_list_of_lists/4`, `setproduct_lists/4`, `ord_intersect_lists/2`, `ord_intersect_all/2`, `sort_list_of_lists/2`, `transitive_closure_lists/3`, `closure_under_union/2`, `ord_split_lists/4`, `ord_split_lists_from_list/4`, `split_lists_from_list/4`, `powerset_of_set_of_sets/2`.

141.2 Documentation on exports

merge_list_of_lists/2: PREDICATE
No further documentation available for this predicate.

merge_lists/3: PREDICATE
No further documentation available for this predicate.

merge_each/3: PREDICATE
No further documentation available for this predicate.

ord_member_list_of_lists/2: PREDICATE
No further documentation available for this predicate.

delete_var_from_list_of_lists/4: PREDICATE
No further documentation available for this predicate.

setproduct_lists/4: PREDICATE
No further documentation available for this predicate.

ord_intersect_lists/2: PREDICATE
No further documentation available for this predicate.

ord_intersect_all/2: PREDICATE

No further documentation available for this predicate.

sort_list_of_lists/2: PREDICATE

No further documentation available for this predicate.

transitive_closure_lists/3: PREDICATE

No further documentation available for this predicate.

closure_under_union/2: PREDICATE

No further documentation available for this predicate.

ord_split_lists/4: PREDICATE

No further documentation available for this predicate.

ord_split_lists_from_list/4: PREDICATE

No further documentation available for this predicate.

split_lists_from_list/4: PREDICATE

No further documentation available for this predicate.

powerset_of_set_of_sets/2: PREDICATE

No further documentation available for this predicate.

141.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `lists`, `sets`, `sort`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

142 PART VII - Standard libraries

Author(s): The Ciao Development Team.

This part documents the standard Ciao library. The predicates in this library are part of the core Ciao system. In particular, this library includes a large subset of the ISO Prolog standard predicates and also of the additional predicates normally provided by most classical Prolog systems, in addition to Ciao-specific extensions.

In general, these modules and predicates have to be imported explicitly as indicated in the corresponding *usage* section. However, if a `module/2` declaration is used (or in user files without a starting `use_package/1` declaration) many of them are loaded by default. See Chapter 109 [Classic Prolog], page 493 and Chapter 25 [The module system], page 127 for more details.

143 Stream handling and operations

Author(s): The Ciao Development Team.

This module reexports the predicates required for stream handling (`stream_basic`**) and input/output operations (** `io_basic`**). See the documentation of the reexported modules for further details.**

143.1 Usage and interface

- **Library usage:**
  `:- use_module(library(streams)).`

143.2 Documentation on exports

fixed_absolute_file_name/3: (UNDOC_REEXPORT)
Imported from `stream_basic` **(see the corresponding documentation for details).**

atm_or_int/1: (UNDOC_REEXPORT)
Imported from `stream_basic` **(see the corresponding documentation for details).**

io_mode/1: (UNDOC_REEXPORT)
Imported from `stream_basic` **(see the corresponding documentation for details).**

stream_alias/1: (UNDOC_REEXPORT)
Imported from `stream_basic` **(see the corresponding documentation for details).**

stream/1: (UNDOC_REEXPORT)
Imported from `stream_basic` **(see the corresponding documentation for details).**

sourcename/1: (UNDOC_REEXPORT)
Imported from `stream_basic` **(see the corresponding documentation for details).**

pipe/2: (UNDOC_REEXPORT)
Imported from `stream_basic` **(see the corresponding documentation for details).**

absolute_file_name/7: (UNDOC_REEXPORT)
Imported from `stream_basic` **(see the corresponding documentation for details).**

absolute_file_name/2: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

stream_code/2: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

current_stream/3: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

clearerr/1: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

flush_output/0: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

flush_output/1: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

line_position/2: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

line_count/2: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

character_count/2: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

current_output/1: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

set_output/1: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

current_input/1: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

set_input/1: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

close/1: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

open_option_list/1: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

open/4: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

open/3: (UNDOC_REEXPORT)
Imported from `stream_basic` (see the corresponding documentation for details).

at_end_of_stream/0: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

at_end_of_stream/1: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

displayq/1: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

displayq/2: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

display/1: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

display/2: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

at_end_of_stream/1: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

at_end_of_stream/0: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

put_byte/1: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

put_byte/2: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

peek_byte/1: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

peek_byte/2: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

get_byte/1: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

get_byte/2: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

getct1/2: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

getct/2: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

string_bytes/2: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

code_class/2: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

tab/1: (UNDOC_REEXPORT)
Imported from `io_basic` **(see the corresponding documentation for details).**

tab/2: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

nl/0: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

nl/1: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

put_code/1: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

put_code/2: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

skip_line/0: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

skip_line/1: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

skip_code/1: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

skip_code/2: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

peek_code/1: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

peek_code/2: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

get1_code/1: (UNDOC_REEXPORT)
Imported from `io_basic` (see the corresponding documentation for details).

get1_code/2: (UNDOC_REEXPORT)

Imported from `io_basic` (see the corresponding documentation for details).

get_code/1: (UNDOC_REEXPORT)

Imported from `io_basic` (see the corresponding documentation for details).

get_code/2: (UNDOC_REEXPORT)

Imported from `io_basic` (see the corresponding documentation for details).

143.3 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

144 Basic file/stream handling

Author(s): Daniel Cabeza, Mats Carlsson.

This module provides basic predicates for handling files and streams, in order to make input/output on them.

144.1 Usage and interface

- **Library usage:**

  `:- use_module(engine(stream_basic)).`
- **Exports:**
  - *Predicates:*

    `open/3`, `open/4`, `close/1`, `set_input/1`, `current_input/1`, `set_output/1`, `current_output/1`, `character_count/2`, `line_count/2`, `line_position/2`, `flush_output/1`, `flush_output/0`, `clearerr/1`, `current_stream/3`, `stream_code/2`, `absolute_file_name/2`, `absolute_file_name/7`, `pipe/2`, `fixed_absolute_file_name/3`.
  - *Regular Types:*

    `open_option_list/1`, `sourcename/1`, `stream/1`, `stream_alias/1`, `io_mode/1`, `atm_or_int/1`.
  - *Multifiles:*

    `file_search_path/2`, `library_directory/1`.

144.2 Documentation on exports

open/3: PREDICATE

`open(File,Mode,Stream)`

Open `File` with mode `Mode` and return in `Stream` the stream associated with the file. No extension is implicit in `File`.

Usage 1: ⟨• ISO •⟩

Normal use.

- *Call and exit should be compatible with:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold at call time:*

  `File` **is a source name.** (`stream_basic:sourcename/1`)

  `Mode` **is an opening mode ('read', 'write' or 'append').** (`stream_basic:io_mode/1`)
- *The following properties should hold upon exit:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Usage 2:

In the special case that `File` is an integer, it is assumed to be a file descriptor passed to Prolog from a foreign function call. The file descriptor is connected to a Prolog stream (invoking the C POSIX function `fdopen()`) which is unified with `Stream`.

- *Call and exit should be compatible with:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold at call time:*
  `File` **is an integer.** (`basic_props:int/1`)
  `Mode` **is an opening mode ('read', 'write' or 'append').** (`stream_basic:io_mode/1`)
- *The following properties should hold upon exit:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)

open/4: PREDICATE

`open(File,Mode,Stream,Options)`

Same as `open(File, Mode, Stream)` **with options** `Options`**. See the definition of** `open_option_list/1` **for details.**

Usage:

- *Call and exit should be compatible with:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold at call time:*
  `File` **is a source name.** (`stream_basic:sourcename/1`)
  `Mode` **is an opening mode ('read', 'write' or 'append').** (`stream_basic:io_mode/1`)
  `Options` **is a list of options for** `open/4`**.** (`stream_basic:open_option_list/1`)
- *The following properties should hold upon exit:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

open_option_list/1: REGTYPE

A list of options for `open/4`**, currently the meaningful options are:**

`lock` **Try to set an advisory lock for the file. If the open mode is** `read`**, the lock is a read (shared) lock, else it is a write (exclusive) lock. If the lock cannot be acquired, the call waits until it is released (but can fail in exceptional cases).**

`lock_nb` **Same as** `lock`**, but the call immediately fails if the lock cannot be acquired.**

`lock(Lock_Mode)`
Same as `lock`**, but specifying in** `Lock_Mode` **whether the lock is** `read` **(also** `shared`**) or** `write` **(also** `exclusive`**). This option has be included for compatibility with the SWI-Prolog locking options, because in general the type of lock should match the open mode as in the** `lock` **option.**

`lock_nb(Lock_Mode)`
Same as the previous option but with the `lock_nb` **behavior.**

All file locking is implemented via the POSIX function fcntl(). Please refer to its manual page for details.

Usage: `open_option_list(L)`

`L` **is a list of options for** `open/4`**.**

close/1: PREDICATE

`close(Stream)`

Close the stream `Stream`.

Usage: ⟨ • ISO • ⟩

- *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

set_input/1: PREDICATE

`set_input(Stream)`

Set the current input stream to `Stream`. A notion of *current input stream* is maintained by the system, so that input predicates with no explicit stream operate on the current input stream. Initially it is set to `user_input`.

Usage: ⟨ • ISO • ⟩

- *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

current_input/1: PREDICATE

`current_input(Stream)`

Unify `Stream` with the current input stream. In addition to the ISO behavior, stream aliases are allowed. This is useful for most applications checking whether a stream is the standard input or output.

Usage: ⟨ • ISO • ⟩

- *Calls should, and exit will be compatible with:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

set_output/1: PREDICATE

`set_output(Stream)`

Set the current output stream to `Stream`. A notion of *current output stream* is maintained by the system, so that output predicates with no explicit stream operate on the current output stream. Initially it is set to `user_output`.

Usage: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
– *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

current_output/1: PREDICATE

`current_output(Stream)`

Unify `Stream` with the current output stream. The same comment as for `current_input/1` applies.

Usage: ⟨• ISO •⟩

– *Calls should, and exit will be compatible with:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
– *The following properties hold upon exit:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
– *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

character_count/2: PREDICATE

`character_count(Stream,Count)`

`Count` **characters have been read from or written to** `Stream`**.**

Usage:

– *Calls should, and exit will be compatible with:*
  `Count` **is an integer.** (`basic_props:int/1`)
– *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
– *The following properties hold upon exit:*
  `Count` **is an integer.** (`basic_props:int/1`)
– *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

line_count/2: PREDICATE

`line_count(Stream,Count)`

`Count` **lines have been read from or written to** `Stream`**.**

Usage:

– *Calls should, and exit will be compatible with:*
  `Count` **is an integer.** (`basic_props:int/1`)
– *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
– *The following properties hold upon exit:*
  `Count` **is an integer.** (`basic_props:int/1`)
– *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

line_position/2: PREDICATE

`line_position(Stream,Count)`

`Count` characters have been read from or written to the current line of `Stream`.

Usage:

- *Calls should, and exit will be compatible with:*
  `Count` is an integer. (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Count` is an integer. (`basic_props:int/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

flush_output/1: PREDICATE

`flush_output(Stream)`

Flush any buffered data to output stream `Stream`.

Usage: ⟨• ISO •⟩

- *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

flush_output/0: PREDICATE

`flush_output`

Behaves like `current_output(S), flush_output(S)`

clearerr/1: PREDICATE

`clearerr(Stream)`

Clear the end-of-file and error indicators for input stream `Stream`.

Usage:

- *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Stream` is an open stream. (`stream_basic:stream/1`)

current_stream/3: PREDICATE

`current_stream(Filename,Mode,Stream)`

`Stream` is a stream which was opened in mode `Mode` and which is connected to the absolute file name `Filename` (an atom) or to the file descriptor `Filename` (an integer). This predicate can be used for enumerating all currently open streams through backtracking.

Usage:

- *Calls should, and exit will be compatible with:*
  `stream_basic:atm_or_int(Filename)` (`stream_basic:atm_or_int/1`)
  `Mode` **is an opening mode ('read', 'write' or 'append').** (`stream_basic:io_mode/1`)
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `stream_basic:atm_or_int(Filename)` (`stream_basic:atm_or_int/1`)
  `Mode` **is an opening mode ('read', 'write' or 'append').** (`stream_basic:io_mode/1`)
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

stream_code/2: PREDICATE

`stream_code(Stream,StreamCode)`

`StreamCode` **is the file descriptor (an integer) corresponding to the Prolog stream** `Stream`**.**

Usage 1:

- *Calls should, and exit will be compatible with:*
  `StreamCode` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `StreamCode` **is an integer.** (`basic_props:int/1`)

Usage 2:

- *The following properties should hold at call time:*
  `Stream` **is a free variable.** (`term_typing:var/1`)
  `StreamCode` **is an integer.** (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)

absolute_file_name/2: PREDICATE

`absolute_file_name(RelFileSpec,AbsFileSpec)`

If `RelFileSpec` **is an absolute pathname then do an absolute lookup. If** `RelFileSpec` **is a relative pathname then prefix the name with the name of the current directory and do an absolute lookup. If** `RelFileSpec` **is a path alias, perform the lookup following the path alias rules (see** `sourcename/1`**). In all cases: if a matching file with suffix** `.pl` **exists, then** `AbsFileSpec` **will be unified with this file. Failure to open a file normally causes an exception. The behaviour can be controlled by the** `fileerrors` **prolog flag.**

Usage: `absolute_file_name(RelFileSpec,AbsFileSpec)`

`AbsFileSpec` **is the absolute name (with full path) of** `RelFileSpec`**.**

- *Call and exit should be compatible with:*
  `RelFileSpec` **is a source name.** (`stream_basic:sourcename/1`)
  `AbsFileSpec` **is an atom.** (`basic_props:atm/1`)

- *The following properties should hold at call time:*

  `RelFileSpec` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `AbsFileSpec` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

absolute_file_name/7: PREDICATE

`absolute_file_name(Spec,Opt,Suffix,CurrDir,AbsFile,AbsBase,AbsDir)`

`AbsFile` is the absolute name (with full path) of `Spec`, which has an optional first suffix `Opt` and an optional second suffix `Suffix`, when the current directory is `CurrDir`. `AbsBase` is the same as `AbsFile`, but without the second suffix, and `AbsDir` is the absolute path of the directory where `AbsFile` is. The Ciao compiler invokes this predicate with `Opt='_opt'` and `Suffix='.pl'` when searching source files.

Usage:

- *The following properties should hold at call time:*

  `Spec` **is a source name.** (`stream_basic:sourcename/1`)

  `Opt` **is an atom.** (`basic_props:atm/1`)

  `Suffix` **is an atom.** (`basic_props:atm/1`)

  `CurrDir` **is an atom.** (`basic_props:atm/1`)

  `AbsFile` **is a free variable.** (`term_typing:var/1`)

  `AbsBase` **is a free variable.** (`term_typing:var/1`)

  `AbsDir` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*

  `AbsFile` **is an atom.** (`basic_props:atm/1`)

  `AbsBase` **is an atom.** (`basic_props:atm/1`)

  `AbsDir` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

pipe/2: PREDICATE

Usage: ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `Arg1` **is a free variable.** (`term_typing:var/1`)

  `Arg2` **is a free variable.** (`term_typing:var/1`)
- *The following properties hold upon exit:*

  `Arg1` **is an open stream.** (`stream_basic:stream/1`)

  `Arg2` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

sourcename/1: REGTYPE

A source name is a flexible way of referring to a concrete file. A source name is either a relative or absolute filename given as:

- an atom, or
- a unary functor (which represents a *path alias*, see below) applied to a *relative* path, the latter being given as an atom.

In all cases certain filename extensions (e.g., `.pl`) can be implicit. In the first form above, file names can be relative to the current directory. Also, file names beginning with `~` or `$` are treated specially. For example,

`'~/ciao/sample.pl'`
: is equivalent to `'/home/staff/herme/ciao/sample.pl'`, if `/home/staff/herme` is the user's home directory. (On POSIX systems, this is equivalent to `'$HOME/ciao/sample.pl'` as explained below.)

`'~bardo/prolog/sample.pl'`
: is equivalent to `'/home/bardo/prolog/sample.pl'`, if `/home/bardo` is `bardo`'s home directory.

`'$UTIL/sample.pl'`
: is equivalent to `'/usr/local/src/utilities/sample.pl'`, if `/usr/local/src/utilities` is the value of the environment variable `UTIL`.

The second form allows using path aliases. Such aliases allow refering to files not with absolute file system paths but with paths which are relative to predefined (or user-defined) abstract names. For example, given the path alias `myutils` which has been defined to refer to path `'/home/bardo/utilities'`, if that directory contains the file `stuff.pl` then the term `myutils(stuff)` in a `use_module/1` declaration would refer to the file `'/home/bardo/utilities/stuff.pl'` (the `.pl` extension is implicit in the `use_module/1` declaration). As a special case, if that directory contains a subdirectory named `stuff` which in turn contains the file `stuff.pl`, the same term would refer to the file `'/home/bardo/utilities/stuff/stuff.pl'`. If a path alias is related to several paths, all paths are scanned in sequence until a match is found. For information on predefined path aliases or how to define new path aliases, see `file_search_path/2`.

Usage: `sourcename(F)`

`F` is a source name.

stream/1: REGTYPE

Streams correspond to the file pointers used at the operating system level, and usually represent opened files. There are four special streams which correspond with the operating system standard streams:

`user_input`
: The standard input stream, i.e. the terminal, usually.

`user_output`
: The standard output stream, i.e. the terminal, usually.

`user_error`
: The standard error stream.

`user`
: The standard input or output stream, depending on context.

Usage: `stream(S)`

`S` is an open stream.

stream_alias/1: REGTYPE

Usage: `stream_alias(S)`

`S` is the alias of an open stream, i.e., an atom which represents a stream at Prolog level.

io_mode/1: REGTYPE

Can have the following values:

`read` Open the file for input.

`write` Open the file for output. The file is created if it does not already exist, the file will otherwise be truncated.

`append` Open the file for output. The file is created if it does not already exist, the file will otherwise be appended to.

Usage: `io_mode(M)`

`M` is an opening mode ('read', 'write' or 'append').

atm_or_int/1: REGTYPE

A regular type, defined as follows:

```
atm_or_int(X) :-
    atm(X).
atm_or_int(X) :-
    int(X).
```

fixed_absolute_file_name/3: PREDICATE

No further documentation available for this predicate.

144.3 Documentation on multifiles

file_search_path/2: PREDICATE

`file_search_path(Alias,Path)`

The path alias `Alias` is linked to path `Path`. Both arguments must be atoms. New facts (or clauses) of this predicate can be asserted to define new path aliases. Predefined path aliases in Ciao are:

`library` Initially points to all Ciao library paths. See `library_directory/1`.

`engine` The path of the Ciao engine builtins.

`.` The current path (`'.'`).

The predicate is *multifile*.

The predicate is of type *dynamic*.

`file_search_path(X,Y)`

– *The following properties hold upon exit:*

`X` is ground. (`basic_props:gnd/1`)

`Y` is ground. (`basic_props:gnd/1`)

library_directory/1: PREDICATE

`library_directory(Path)`

`Path` **is a library path (a path represented by the path alias** `library`**). More library paths can be defined by asserting new facts (or clauses) of this predicate. The predicate is** *multifile***.**

The predicate is of type *dynamic***.**

`library_directory(X)`

- *The following properties hold upon exit:*

  `X` **is ground.** (`basic_props:gnd/1`)

144.4 Documentation on imports

This module has the following direct dependencies:

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`.

145 Basic input/output stream operations

Author(s): Daniel Cabeza, Mats Carlsson, Jose F. Morales (minor).

This module provides the basic input/output operations on streams (byte/code input/output and canonical term output). Most predicates are provided in two versions: one that specifies the input or output stream as the first argument and a second which omits this argument and uses the current input or output stream.

From the ISO-Prolog predicates for character input/output, only the `_code` versions are provided, the `_char` versions are given by `library(iso_char)`.

145.1 Usage and interface

- **Library usage:**
  `:- use_module(engine(io_basic)).`
- **Exports:**
  - *Predicates:*
    `get_code/2`, `get_code/1`, `get1_code/2`, `get1_code/1`, `peek_code/2`, `peek_code/1`, `skip_code/2`, `skip_code/1`, `skip_line/1`, `skip_line/0`, `put_code/2`, `put_code/1`, `nl/1`, `nl/0`, `tab/2`, `tab/1`, `code_class/2`, `string_bytes/2`, `getct/2`, `getct1/2`, `get_byte/2`, `get_byte/1`, `peek_byte/2`, `peek_byte/1`, `put_byte/2`, `put_byte/1`, `at_end_of_stream/0`, `at_end_of_stream/1`, `display/2`, `display/1`, `displayq/2`, `displayq/1`, `at_end_of_stream/1`, `at_end_of_stream/0`.

145.2 Documentation on exports

get_code/2: PREDICATE

`get_code(Stream,Code)`

Reads from `Stream` the next character and unifies `Code` with its character code. At end of stream, unifies `Code` with the integer -1.

Usage: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Code` is an integer. (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Code` is an integer. (`basic_props:int/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

get_code/1: PREDICATE

`get_code(Code)`

Behaves like `current_input(S), get_code(S,Code)`.

Usage: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Code` **is an integer.** (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Code` **is an integer.** (`basic_props:int/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

get1_code/2: PREDICATE

`get1_code(Stream,Code)`

Reads from `Stream` the next non-layout character (see `code_class/2`) and unifies `Code` with its character code. At end of stream, unifies `Code` with the integer -1.

Usage:

- *Calls should, and exit will be compatible with:*
  `Code` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Code` **is an integer.** (`basic_props:int/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

get1_code/1: PREDICATE

`get1_code(Code)`

Behaves like `current_input(S), get1_code(S,Code)`.

Usage:

- *Calls should, and exit will be compatible with:*
  `Code` **is an integer.** (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Code` **is an integer.** (`basic_props:int/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

peek_code/2: PREDICATE

`peek_code(Stream,Code)`

Unifies `Code` with the character code of the next character of `Stream`, leaving the stream position unaltered. At end of stream, unifies `Code` with the integer -1.

Usage: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Code` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Code` **is an integer.** (`basic_props:int/1`)

peek_code/1: PREDICATE

`peek_code(Code)`

Behaves like `current_input(S), peek_code(S,Code)`.

Usage: ⟨ • ISO • ⟩

- *Calls should, and exit will be compatible with:*

  `Code` **is an integer.** (`basic_props:int/1`)
- *The following properties hold upon exit:*

  `Code` **is an integer.** (`basic_props:int/1`)

skip_code/2: PREDICATE

`skip_code(Stream,Code)`

Skips just past the next character code `Code` from `Stream`.

Usage:

- *The following properties should hold at call time:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

  `Code` **is an integer.** (`basic_props:int/1`)

skip_code/1: PREDICATE

`skip_code(Code)`

Behaves like `current_input(S), skip_code(S,Code)`.

Usage:

- *The following properties should hold at call time:*

  `Code` **is an integer.** (`basic_props:int/1`)

skip_line/1: PREDICATE

`skip_line(Stream)`

Skips from `Stream` the remaining input characters on the current line. If the end of the stream is reached, the stream will stay at its end. Portable among different operating systems.

Usage:

- *The following properties should hold at call time:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

skip_line/0: PREDICATE

`skip_line`

Behaves like `current_input(S), skip_line(S)`.

put_code/2: PREDICATE

`put_code(Stream,Code)`

Outputs to `Stream` the character corresponding to character code `Code`.

Usage: ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

  `Code` **is an integer.** (`basic_props:int/1`)

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  All calls of the form `put_code(Stream,Code)` are deterministic. (`native_props:is_det/1`)

put_code/1: PREDICATE

`put_code(Code)`

Behaves like `current_output(S), put_code(S,Code)`.

Usage: ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `Code` **is an integer.** (`basic_props:int/1`)

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  All calls of the form `put_code(Code)` are deterministic. (`native_props:is_det/1`)

nl/1: PREDICATE

`nl(Stream)`

Outputs a newline character to `Stream`. Equivalent to `put_code(Stream, 0'\n)`.

Usage: ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  All calls of the form `nl(Stream)` are deterministic. (`native_props:is_det/1`)

  All the calls of the form `nl(Stream)` do not fail. (`native_props:not_fails/1`)

nl/0: PREDICATE

`nl`

Behaves like `current_output(S), nl(S)`.

Usage: ⟨• ISO •⟩

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

  All calls of the form `nl` are deterministic. (`native_props:is_det/1`)

  All the calls of the form `nl` do not fail. (`native_props:not_fails/1`)

  Goal `nl` produces `1` solutions. (`native_props:relations/2`)

tab/2: PREDICATE

`tab(Stream,Num)`

Outputs `Num` spaces to `Stream`.

Usage:

- *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
  `Num` is an integer. (`basic_props:int/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  All calls of the form `tab(Stream,Num)` are deterministic. (`native_props:is_det/1`)

tab/1: PREDICATE

`tab(Num)`

Behaves like `current_output(S), tab(S,Num)`.

Usage:

- *The following properties should hold at call time:*
  `Num` is an integer. (`basic_props:int/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)
  All calls of the form `tab(Num)` are deterministic. (`native_props:is_det/1`)

code_class/2: PREDICATE

`code_class(Code,Class)`

Unifies `Class` with an integer corresponding to the lexical class of the character whose code is `Code`, with the following correspondence:

```
0 - layout (includes space, newline, tab)
1 - small letter
2 - capital letter (including '_')
3 - digit
4 - graphic (includes #$&*+-./:<=>?@^\`~ )
5 - punctuation (includes !;"'%(),[]{|} )
6 - extended identifier continuation
7 - invalid
```

Note that in ISO-Prolog the back quote ` is a punctuation character, whereas in Ciao it is a graphic character. Thus, if compatibility with ISO-Prolog is desired, the programmer should not use this character in unquoted names.

Usage:

- *Calls should, and exit will be compatible with:*
  `Class` is an integer. (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Code` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Class` is an integer. (`basic_props:int/1`)

string_bytes/2: PREDICATE

`string_bytes(String,Bytes)`

`String` is the list of the character codes encoded by the list of bytes `Bytes` (equivalent to `String=Bytes` if any of the lists is a list of ASCII codes).

getct/2: PREDICATE

`getct(Code,Type)`

Reads from the current input stream the next character, unifying `Code` with its character code, and `Type` with its lexical class. At end of stream, unifies both `Code` and `Type` with the integer -1. Equivalent to

```
get(Code), (Code = -1 -> Type = -1 ; code_class(Code,Type))
```

Usage:

– *Calls should, and exit will be compatible with:*

`Code` **is an integer.** (`basic_props:int/1`)

`Type` **is an integer.** (`basic_props:int/1`)

– *The following properties hold upon exit:*

`Code` **is an integer.** (`basic_props:int/1`)

`Type` **is an integer.** (`basic_props:int/1`)

getct1/2: PREDICATE

`getct1(Code,Type)`

Reads from the current input stream the next non-layout character, unifying `Code` with its character code, and `Type` with its lexical class (which will be nonzero). At end of stream, unifies both `Code` and `Type` with the integer -1. Equivalent to

```
get1(Code), (Code = -1 -> Type = -1 ; code_class(Code,Type))
```

Usage:

– *Calls should, and exit will be compatible with:*

`Code` **is an integer.** (`basic_props:int/1`)

`Type` **is an integer.** (`basic_props:int/1`)

– *The following properties hold upon exit:*

`Code` **is an integer.** (`basic_props:int/1`)

`Type` **is an integer.** (`basic_props:int/1`)

get_byte/2: PREDICATE

`get_byte(Stream,Byte)`

Reads from `Stream` the next byte and unifies it with `Byte`. At end of stream, unifies `Byte` with the integer -1.

Usage: ⟨• ISO •⟩

– *Calls should, and exit will be compatible with:*

`Byte` **is an integer.** (`basic_props:int/1`)

- *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Byte` is an integer. (`basic_props:int/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

get_byte/1: PREDICATE

`get_byte(Byte)`

Behaves like `current_input(S), get_byte(S,Byte)`.

Usage: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Byte` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Byte` is an integer. (`basic_props:int/1`)
- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

peek_byte/2: PREDICATE

`peek_byte(Stream,Byte)`

Unifies `Byte` with the next byte of `Stream`, leaving the stream position unaltered. At end of stream, unifies `Byte` with the integer -1.

Usage: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Byte` is an integer. (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties hold upon exit:*
  `Byte` is an integer. (`basic_props:int/1`)

peek_byte/1: PREDICATE

`peek_byte(Byte)`

Behaves like `current_input(S), peek_byte(S,Byte)`.

Usage: ⟨• ISO •⟩

- *Calls should, and exit will be compatible with:*
  `Byte` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Byte` is an integer. (`basic_props:int/1`)

put_byte/2: PREDICATE

`put_byte(Stream,Byte)`

Outputs to `Stream` the byte `Byte`.

Usage: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*

`Stream` **is an open stream.** (`stream_basic:stream/1`)

`Byte` **is an integer.** (`basic_props:int/1`)

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

put_byte/1: PREDICATE

`put_byte(Byte)`

Behaves like `current_output(S), put_byte(S,Byte)`.

Usage: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*

`Byte` **is an integer.** (`basic_props:int/1`)

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

All calls of the form `put_byte(Byte)` are deterministic. (`native_props:is_det/1`)

at_end_of_stream/0: PREDICATE

`at_end_of_stream`

Behaves like `current_input(S), at_end_of_stream(S)`.

at_end_of_stream/1: PREDICATE

`at_end_of_stream(Stream)`

`Stream` has a stream position end-of-stream or past-end-of-stream. This predicate is provided for ISO compatibility, but its usage is discouraged, since it cannot reliably detect that no more data is available in the stream (e.g., without blocking). Use `peek_byte/2` or `peek_code/2` instead.

Usage: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*

`Stream` **is an open stream.** (`stream_basic:stream/1`)

display/2: PREDICATE

`display(Stream,Term)`

Displays `Term` onto `Stream`. Lists are output using list notation, the other compound terms are output in functional notation. Similar to `write_term(Stream, Term, [ignore_ops(ops)])`, except that curly bracketed notation is not used with `{}/1`, and the `write_strings` flag is not honored.

Usage:

– *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold upon exit:*
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold globally:*
  `Term` is not further instantiated. (`basic_props:not_further_inst/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

display/1: PREDICATE

`display(Term)`

Behaves like `current_output(S), display(S,Term)`.

Usage:

– *The following properties should hold at call time:*
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold upon exit:*
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold globally:*
  `Term` is not further instantiated. (`basic_props:not_further_inst/2`)
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

displayq/2: PREDICATE

`displayq(Stream,Term)`

Similar to `display(Stream, Term)`, but atoms and functors that can't be read back by `read_term/3` are quoted. Thus, similar to `write_term(Stream, Term, [quoted(true), ignore_ops(ops)])`, with the same exceptions as `display/2`.

Usage:

– *The following properties should hold at call time:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold upon exit:*
  `Term` is any term. (`basic_props:term/1`)
– *The following properties hold globally:*
  `Term` is not further instantiated. (`basic_props:not_further_inst/2`)

displayq/1: PREDICATE

`displayq(Term)`

Behaves like `current_output(S), displayq(S,Term)`.

Usage:

– *The following properties should hold at call time:*
  `Term` is any term. (`basic_props:term/1`)

– *The following properties hold upon exit:*
  `Term` **is any term.** (`basic_props:term/1`)
– *The following properties hold globally:*
  `Term` **is not further instantiated.** (`basic_props:not_further_inst/2`)

at_end_of_stream/1: PREDICATE

`at_end_of_stream(Stream)`

`Stream` has a stream position end-of-stream or past-end-of-stream. This predicate is provided for ISO compatibility, but its usage is discouraged, since it cannot reliably detect that no more data is available in the stream (e.g., without blocking). Use `peek_byte/2` or `peek_code/2` instead.

Usage: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)

at_end_of_stream/0: PREDICATE

`at_end_of_stream`

Behaves like `current_input(S), at_end_of_stream(S)`.

145.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*
  `native_props`.
– *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `nativeprops`, `isomodes`.

146 Stream utilities

Author(s): The Ciao Development Team.

This module implements a collection of predicates to read/write streams (or files) from/to several sources (lists of terms, strings, predicate output, etc.).

146.1 Usage and interface

- **Library usage:**
  `:- use_module(library(stream_utils)).`
- **Exports:**
  - *Predicates:*
    `get_line/2`, `get_line/1`, `read_string_to_end/2`, `read_bytes_to_end/2`, `discard_to_end/1`, `read_bytes/3`, `copy_stream/3`, `write_string/2`, `write_string/1`, `write_bytes/2`, `write_bytes/1`, `file_to_string/2`, `string_to_file/2`, `file_to_bytes/2`, `bytes_to_file/2`, `output_to_file/2`, `open_input/2`, `close_input/1`, `open_output/2`, `close_output/1`.
  - *Regular Types:*
    `line/1`.

146.2 Documentation on exports

get_line/2: PREDICATE

Usage: `get_line(S,L)`

Reads from `Stream` a line of text and unifies `Line` with it. The end of the line can have Unix `[10]` or Windows/DOS `[13, 10]` termination, which is not included in `Line`. At EOF, the term `end_of_file` is returned.

- *The following properties should hold at call time:*
  `S` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold upon exit:*
  `stream_utils:line(L)` (`stream_utils:line/1`)

get_line/1: PREDICATE

Usage: `get_line(L)`

Behaves like `current_input(S), get_line(S,Line)`.

- *The following properties should hold upon exit:*
  `stream_utils:line(L)` (`stream_utils:line/1`)

line/1: REGTYPE

A regular type, defined as follows:

```
line(L) :-
    string(L).
line(end_of_file).
```

read_string_to_end/2: PREDICATE

Usage: `read_string_to_end(Stream,String)`

Reads in `String` all the characters from `Stream` until an EOF is found.

– *The following properties should hold at call time:*

`Stream` **is an open stream.** (`stream_basic:stream/1`)

– *The following properties should hold upon exit:*

`String` **is a string (a list of character codes).** (`basic_props:string/1`)

read_bytes_to_end/2: PREDICATE

Usage: `read_bytes_to_end(Stream,Bytes)`

Reads in `Bytes` all the bytes from `Stream` until an EOF is found.

– *The following properties should hold at call time:*

`Stream` **is an open stream.** (`stream_basic:stream/1`)

– *The following properties should hold upon exit:*

`Bytes` **is list of bytes.** (`basic_props:bytelist/1`)

discard_to_end/1: PREDICATE

Usage: `discard_to_end(Stream)`

Reads in all the bytes from `Stream` until an EOF is found.

– *The following properties should hold at call time:*

`Stream` **is an open stream.** (`stream_basic:stream/1`)

read_bytes/3: PREDICATE

Usage: `read_bytes(Stream,N,Bytes)`

Reads in `Bytes` at most `N` bytes from `Stream`, or until an EOF is found.

– *The following properties should hold at call time:*

`Stream` **is an open stream.** (`stream_basic:stream/1`)

`N` **is an integer.** (`basic_props:int/1`)

– *The following properties should hold upon exit:*

`Bytes` **is list of bytes.** (`basic_props:bytelist/1`)

copy_stream/3: PREDICATE

Usage: `copy_stream(InS,OutS,Copied)`

Copies all bytes bytes (until EOF or error) from the `InS` stream into the `OutS` stream. The number of copied bytes is returned in `Copied`

– *The following properties should hold at call time:*

`InS` **is an open stream.** (`stream_basic:stream/1`)

`OutS` **is an open stream.** (`stream_basic:stream/1`)

– *The following properties should hold upon exit:*

`Copied` **is an integer.** (`basic_props:int/1`)

write_string/2: PREDICATE

Usage: `write_string(Stream,String)`

Writes `String` **onto** `Stream`.

- *The following properties should hold at call time:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

  `String` **is a string (a list of character codes).** (`basic_props:string/1`)

write_string/1: PREDICATE

Usage: `write_string(String)`

Behaves like `current_input(S), write_string(S, String)`.

- *The following properties should hold at call time:*

  `String` **is a string (a list of character codes).** (`basic_props:string/1`)

write_bytes/2: PREDICATE

Usage: `write_bytes(Stream,Bytes)`

Writes `Bytes` **onto** `Stream`.

- *The following properties should hold at call time:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

  `Bytes` **is list of bytes.** (`basic_props:bytelist/1`)

write_bytes/1: PREDICATE

Usage: `write_bytes(Bytes)`

Behaves like `current_input(S), write_bytes(S, Bytes)`.

- *The following properties should hold at call time:*

  `Bytes` **is list of bytes.** (`basic_props:bytelist/1`)

file_to_string/2: PREDICATE

Usage: `file_to_string(FileName,String)`

Reads all the characters from the file `FileName` **and returns them in** `String`.

- *The following properties should hold at call time:*

  `FileName` **is a source name.** (`stream_basic:sourcename/1`)

- *The following properties should hold upon exit:*

  `String` **is a string (a list of character codes).** (`basic_props:string/1`)

string_to_file/2: PREDICATE

Usage: `string_to_file(String,FileName)`

Reads all the characters from the string `String` **and writes them to file** `FileName`.

- *The following properties should hold at call time:*

  `String` **is a string (a list of character codes).** (`basic_props:string/1`)

  `FileName` **is a source name.** (`stream_basic:sourcename/1`)

file_to_bytes/2: PREDICATE

Usage: `file_to_bytes(FileName,Bytes)`

Reads all the bytes from the file `FileName` and returns them in `Bytes`.

- *The following properties should hold at call time:*
  `FileName` **is a source name.** (`stream_basic:sourcename/1`)
- *The following properties should hold upon exit:*
  `Bytes` **is list of bytes.** (`basic_props:bytelist/1`)

bytes_to_file/2: PREDICATE

Usage: `bytes_to_file(Bytes,FileName)`

Reads all the bytes from the bytes `Bytes` and writes them to file `FileName`.

- *The following properties should hold at call time:*
  `Bytes` **is list of bytes.** (`basic_props:bytelist/1`)
  `FileName` **is a source name.** (`stream_basic:sourcename/1`)

output_to_file/2: PREDICATE

No further documentation available for this predicate. *Meta-predicate* **with arguments:** `output_to_file(goal,?)`.

open_input/2: PREDICATE

Usage: `open_input(FileName,InputStreams)`

- *The following properties should hold at call time:*
  `FileName` **is a source name.** (`stream_basic:sourcename/1`)
- *The following properties should hold upon exit:*
  `stream_utils:input_handler(InputStreams)` (`stream_utils:input_handler/1`)

close_input/1: PREDICATE

Usage: `close_input(InputStreams)`

- *The following properties should hold at call time:*
  `stream_utils:input_handler(InputStreams)` (`stream_utils:input_handler/1`)
- *The following properties should hold upon exit:*
  `stream_utils:input_handler(InputStreams)` (`stream_utils:input_handler/1`)

open_output/2: PREDICATE

Usage: `open_output(FileName,OutputStreams)`

- *The following properties should hold at call time:*
  `FileName` **is a source name.** (`stream_basic:sourcename/1`)
- *The following properties should hold upon exit:*
  `stream_utils:output_handler(OutputStreams)` (`stream_utils:output_handler/1`)

close_output/1: PREDICATE

Usage: `close_output(OutputStreams)`

– *The following properties should hold at call time:*

`stream_utils:output_handler(OutputStreams)` (`stream_utils:output_handler/1`)

– *The following properties should hold upon exit:*

`stream_utils:output_handler(OutputStreams)` (`stream_utils:output_handler/1`)

146.3 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `hiord`.

147 Tokenizer

Author(s): The Ciao Development Team, Jose F. Morales (curly blocks, Unicode source support).

This module defines the tokenizer for Ciao. In addition to optional flags, the main differences w.r.t. the ISO-Prolog standard are:

- ` is a graphic char, there are no `back_quoted_strings`.
- \ followed by any layout char (not only `new_line`) in a string is a `continuation_escape_sequence`.
- `\^` starts a `control_escape_char` in a string.
- `\c` skips layout in a string.
- `\e` = `ESC`, `\d` = `DEL`, `\s` = `SPACE`.
- `13'23` is 23 in base 13 (same for other bases).
- `0''` is accepted as `0'''` (if not followed by `'`).
- Quoted atoms and strings support Unicode escape `\uDDDD` (four hexadecimal digits) and `\UDDDDDDDD` (eight hexadecimal digits)
- Support for Unicode identifiers (see ⟨undefined⟩ [Unicode source code], page ⟨undefined⟩).

147.1 Unicode source code

The tokenizer for Ciao extends ISO-Prolog with support for Unicode source identifiers, based on the the Unicode Standard Annex 31 (`https://unicode.org/reports/tr31/#Case_and_Stability`), as follows:

- Identifiers can begin with `XID_Start` characters and must be followed with zero or more `XID_Continue` (see the Unicode Derived Core Properties), extended with categorty `No`. Variables are those identifiers that start with characters in the `Lu` category.
- Use `Z*` as layout characters, as well as other control characters (`Cc` categoty) with bidirectional category `WS` (whitespace), `S` (segment separator), or `B` (paragraph separator).
- Use `S*` (`Sm`, `Sc`, `Sk`, `So`) and `P*` (`Pc`, `Pd`, `Ps`, `Pe`, `Pi`, `Pf`, `Po`) as symbols.
- Identifiers that begin with `XID_Continue` are treated as *solo* tokens.

147.2 Usage and interface

- **Library usage:**
  `:- use_module(library(tokenize)).`
- **Exports:**
  - *Predicates:*
    `read_tokens/2`.
  - *Regular Types:*
    `token/1`.
  - *Multifiles:*
    `define_flag/3`.

147.3 Documentation on exports

token/1: REGTYPE

A regular type, defined as follows:

```
token(atom(A)) :-
    atm(A).
token(badatom(S)) :-
    string(S).
token(number(N)) :-
    num(N).
token(string(S)) :-
    string(S).
token(var(T,S)) :-
    term(T),
    string(S).
token('/* ...').
token(',').
token('(').
token(' (').
token(')').
token('[').
token(']').
token('|').
token('{').
token('}').
token('.').
```

read_tokens/2: PREDICATE

Usage: `read_tokens(TokenList,Dictionary)`

– *The following properties should hold upon exit:*

`TokenList` **is a list of** `tokens`. (`basic_props:list/2`)

`Dictionary` **is a dictionary.** (`dict:dictionary/1`)

147.4 Documentation on multifiles

define_flag/3: PREDICATE

Usage: `define_flag(Flag,FlagValues,Default)`

– *The following properties hold upon exit:*

`Flag` **is an atom.** (`basic_props:atm/1`)

Define the valid flag values (`basic_props:flag_values/1`)

The predicate is ***multifile*****.**

147.5 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `read`, `dict`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `define_flag`.

148 Term input

Author(s): Daniel Cabeza (modifications and documentation, adapted from SICStus 0.6 code), Manuel Carro (modifications and documentation), Jose F. Morales (modifications for curly blocks,postfix blocks, infix dot, string constants, and doccomments), Manuel Hermenegildo (minor in documentation).

This module provides facilities for reading terms in Prolog syntax. This is very convenient in many cases (and not only if you are writing a Prolog compiler) because Prolog terms are easy to write and can encode a large amount of information in a human-readable fashion. Note that, in addition, the use of operator definitions makes it possible to enhance the readibility of such terms, or, for example, data files composed of Prolog terms.

The behavior of these reading predicates can be modified in two ways:

- The operators that are active at run time when the reading predicate is called (see Chapter 151 [Defining operators], page 665).
- A number of flags (see below) which control how some special terms are read in or activate some syntax extensions.

148.1 Usage and interface

- **Library usage:**
  `:- use_module(library(read)).`
- **Exports:**
  - *Predicates:*
    `read/1`, `read/2`, `read_term/2`, `read_term/3`, `read_top_level/3`, `second_prompt/2`.
  - *Regular Types:*
    `read_option/1`.
  - *Multifiles:*
    `define_flag/3`.

148.2 Documentation on exports

read/1: PREDICATE

`read(Term)`

Like `read(Stream,Term)` with `Stream` associated to the current input stream.

Usage: ⟨ • ISO • ⟩

- *Call and exit should be compatible with:*
  `Term` is any term. (`basic_props:term/1`)
- *The following properties should hold upon exit:*
  `Term` is any term. (`basic_props:term/1`)

read/2: PREDICATE

Usage: `read(Stream,Term)` ⟨ • ISO • ⟩

The next term, delimited by a full-stop (i.e., a . followed by either a space or a control character), is read from `Stream` and is unified with `Term`. The syntax of the term must agree with current operator declarations. If the end of `Stream` has been reached, `Term` is unified with the term `end_of_file`. Further calls to `read/2` for the same stream will then cause an error, unless the stream is connected to the terminal (in which case a prompt is opened on the terminal).

– *The following properties should hold at call time:*

  `Stream` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Stream` is an open stream. (`stream_basic:stream/1`)

  `Term` is any term. (`basic_props:term/1`)

read_term/2: PREDICATE

Usage: `read_term(Term,Options)` ⟨• ISO •⟩

Like `read_term/3`, but reading from the current input

– *The following properties should hold at call time:*

  `Options` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Term` is any term. (`basic_props:term/1`)

  `Options` is a list of `read_options`. (`basic_props:list/2`)

read_term/3: PREDICATE

Usage: `read_term(Stream,Term,Options)` ⟨• ISO •⟩

Reads a `Term` from `Stream` with the ISO-Prolog `Options`. These options can control the behavior of read term (see `read_option/1`).

– *The following properties should hold at call time:*

  `Stream` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Options` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Stream` is an open stream. (`stream_basic:stream/1`)

  `Term` is any term. (`basic_props:term/1`)

  `Options` is a list of `read_options`. (`basic_props:list/2`)

read_top_level/3: PREDICATE

`read_top_level(Stream,Data,Variables)`

Predicate used to read in the Top Level.

second_prompt/2: PREDICATE

Usage: `second_prompt(Old,New)`

Changes the prompt (the *second prompt*, as opposed to the first one, used by the toplevel) used by `read/2` and friends to `New`, and returns the current one in `Old`.

– *The following properties should hold upon exit:*

  `Old` is currently instantiated to an atom. (`term_typing:atom/1`)

  `New` is currently instantiated to an atom. (`term_typing:atom/1`)

read_option/1: REGTYPE

Usage: `read_option(Option)`

`Option` is an allowed `read_term/[2,3]` option. These options are:

```
read_option(variables(_V)).
read_option(variable_names(_N)).
read_option(singletons(_S)).
read_option(lines(_StartLine,_EndLine)).
read_option(dictionary(_Dict)).
```

They can be used to return the singleton variables in the term, a list of variables, etc.

148.3 Documentation on multifiles

define_flag/3: PREDICATE

The folowing flags are defined:

```
define_flag(read_hiord,[on,off],off).
define_flag(read_curly_blocks,[on,off],off).
define_flag(read_postfix_blocks,[on,off],off).
define_flag(read_string_data_type,[on,off],off).
define_flag(read_infix_dot,[on,off],off).
```

(See Chapter 155 [Runtime system control and flags], page 683).

`read_hiord`
: If flag is `on` (it is `off` by default), a variable followed by a parenthesized list of arguments is read as a `call/N` term, except if the variable is anonymous, in which case it is read as an anonymous predicate abstraction head. For example, `P(X)` is read as `call(P,X)` and `_(X,Y)` as `''(X,Y)`.

`read_curly_blocks`
: When enabled, read terms of the form '`{` <*list of sentences*> `}`'.

`read_postfix_blocks`
: When enabled, allow '<*term*> <*term*>' as valid terms.

`read_string_data_type`
: When enabled, strings are read as '`\6\string`' terms (not as lists).

`read_infix_dot`
: When enabled, infix dot '`.`' is read as '`\6\dot`' terms (not as a list).

Usage: `define_flag(Flag,FlagValues,Default)`

– *The following properties hold upon exit:*

`Flag` is an atom. (`basic_props:atm/1`)

Define the valid flag values (`basic_props:flag_values/1`)

The predicate is *multifile*.

148.4 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`datafacts_rt`, `tokenize`, `dict`, `operators`, `lists`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`, `define_flag`, `datafacts`.

149 Term input from strings

Author(s): Francisco Bueno, Daniel Cabeza, Manuel Hermenegildo, Jose F. Morales.

Stability: [devel] This implementation is incomplete

This module implements predicates for reading (parsing) terms from strings or atom codes.

149.1 Usage and interface

- **Library usage:**
  `:- use_module(library(read_from_string)).`
- **Exports:**
  - *Predicates:*
    `read_from_string/2`,
    `read_from_string/3`, `read_from_string_opts/4`, `read_from_string_atmvars/2`,
    `read_from_string_atmvars/3`, `read_from_atom_atmvars/2`, `read_from_atom/2`.

149.2 Documentation on exports

read_from_string/2: PREDICATE

Usage: `read_from_string(String,Term)`

Read a term `Term` from `String`.

- *The following properties should hold at call time:*
  `String` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Term` **is a free variable.** (`term_typing:var/1`)

read_from_string/3: PREDICATE

Usage: `read_from_string(String,Term,Rest)`

Read a term `Term` from `String` up to `Rest` (which is the non-parsed rest of the list).

- *The following properties should hold at call time:*
  `String` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Term` **is a free variable.** (`term_typing:var/1`)

read_from_string_opts/4: PREDICATE

Usage: `read_from_string_opts(String,Term,Rest,Opts)`

`String` **is parsed into `Term` up to `Rest` (which is the non-parsed rest of the list). The options in `Opts` can be:**

`variable_names(Ns)`
: **Read variable names in `Ns`.**

– *The following properties should hold at call time:*

`String` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Term` **is a free variable.** (`term_typing:var/1`)

`Opts` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

read_from_string_atmvars/2: PREDICATE

Usage: `read_from_string_atmvars(String,Term)`

Read a term `Term` **from** `String`**. Variables are converted to atoms with the name of the variable. It there is some rest of the string left after parsing the term, it is ignored (see** `read_from_string_atmvars/3`**).**

– *The following properties should hold at call time:*

`String` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Term` **is a free variable.** (`term_typing:var/1`)

read_from_string_atmvars/3: PREDICATE

Usage: `read_from_string_atmvars(String,Term,Rest)`

Read a term `Term` **from** `String` **up to** `Rest` **(which is the non-parsed rest of the list). Unquoted uppercase identifiers (variables) are read as atoms instead of variables (thus, the read term is always ground).**

– *The following properties should hold at call time:*

`String` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Term` **is a free variable.** (`term_typing:var/1`)

read_from_atom_atmvars/2: PREDICATE

Usage: `read_from_atom_atmvars(Atom,Term)`

Like `read_from_string_atmvars/2`**, but reads the term** `Term` **from the atom codes in** `Atom`**.**

– *The following properties should hold at call time:*

`Atom` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Term` **is a free variable.** (`term_typing:var/1`)

read_from_atom/2: PREDICATE

Usage: `read_from_atom(Atom,Term)`

Read the term `Term` **from the codes in the name of** `Atom`**.**

– *The following properties should hold at call time:*

`Atom` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Term` **is a free variable.** (`term_typing:var/1`)

149.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `dict`, `read`, `operators`, `port_reify`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `basicmodes`.

150 Term output

Author(s): Richard A. O'Keefe (original version), Mats Carlsson (changes), Daniel Cabeza (changes), Manuel Hermenegildo (changes), Manuel Carro (changes).

This library provides different predicates for term output, additional to the kernel predicates `display/1`- `display/2` **and** `displayq/1`-`displayq/2`. **All the predicates defined in ISO-Prolog are included, plus other traditionally provided by Prolog Implementations. Output predicates are provided in two versions: one that uses the current output stream and another in which the stream is specified explicitly, as an additional first argument.**

150.1 Usage and interface

- **Library usage:**
  `:- use_module(library(write)).`
- **Exports:**
  - *Predicates:*
    `write_term/3`, `write_term/2`, `write/2`, `write/1`, `writeq/2`, `writeq/1`, `write_canonical/2`, `write_canonical/1`, `print/2`, `print/1`, `printq/2`, `printq/1`, `portray_clause/2`, `portray_clause/1`, `numbervars/3`, `prettyvars/1`, `printable_char/1`, `write_attribute/1`.
  - *Properties:*
    `write_option/1`.
  - *Multifiles:*
    `define_flag/3`, `portray_attribute/2`, `portray/1`.

150.2 Documentation on exports

write_term/3: PREDICATE

Usage: `write_term(Stream,Term,OptList)` ⟨• ISO •⟩

Outputs the term `Term` **to the stream** `Stream`, **with the list of write-options** `OptList`. **See** `write_option/1` **type for default options.**

- *The following properties should hold at call time:*
  `OptList` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
  `Term` **is any term.** (`basic_props:term/1`)
  `OptList` **is a list of** `write_options`. (`basic_props:list/2`)
- *The following properties should hold globally:*
  `Stream` **is not further instantiated.** (`basic_props:not_further_inst/2`)

write_term/2: PREDICATE

Usage: `write_term(Term,OptList)` ⟨• ISO •⟩

Behaves like `current_output(S), write_term(S,Term,OptList).`

– *The following properties should hold at call time:*

`OptList` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`Term` is any term. (`basic_props:term/1`)

`OptList` is a list of `write_options`. (`basic_props:list/2`)

write_option/1: PROPERTY

`Opt` is a valid write option which affects the predicate `write_term/3` and similar ones. Possible write_options are:

- **quoted**(*bool*): If *bool* is `true`, atoms and functors that can't be read back by `read_term/3` are quoted, if it is `false`, each atom and functor is written as its name. Default value is `false`.
- **ignore_ops**(*flag*): If *flag* is `true`, each compound term is output in functional notation, if it is `ops`, curly bracketed notation and list notation is enabled when outputing compound terms, if it is `false`, also operator notation is enabled when outputing compound terms. Default value is `false`.
- **numbervars**(*bool*): If *bool* is `true`, a term of the form `'$VAR'(N)` where `N` is an integer, is output as a variable name consisting of a capital letter possibly followed by an integer, a term of the form `'$VAR'(Atom)` where `Atom` is an atom, as this atom (without quotes), and a term of the form `'$VAR'(String)` where `String` is a character string, as the atom corresponding to this character string. See predicates `numbervars/3` and `prettyvars/1`. If *bool* is `false` this cases are not treated in any special way. Default value is `false`.
- **portrayed**(*bool*): If *bool* is `true`, then call multifile predicates `portray/1` and `portray_attribute/2`, to provide the user handlers for pretty printing some terms. `portray_attribute/2` is called whenever an attributed variable is to be printed, `portray/1` is called whenever a non-variable term is to be printed. If either call succeeds, then it is assumed that the term has been output, else it is printed as usual. If *bool* is `false`, these predicates are not called. Default value is `false`. This option is set by the toplevel when writing the final values of variables, and by the debugging package when writing the goals in the tracing messages. Thus you can vary the forms of these messages if you wish.
- **max_depth**(*depth*): *depth* is a positive integer or cero. If it is positive, it denotes the depth limit on printing compound terms. If it is cero, there is no limit. Default value is `0` (no limit).
- **priority**(*prio*): *prio* is an integer between 1 and 1200. If the term to be printed has higher priority than *prio*, it will be printed parenthesized. Default value is 1200 (no term parenthesized).

.

Usage: `write_option(Opt)`

`Opt` is a valid write option.

write/2: PREDICATE

Usage: `write(Stream,Term)` ⟨• ISO •⟩

Behaves like `write_term(Stream, Term, [numbervars(true)])`.

– *The following properties should hold at call time:*

`Stream` is an open stream. (`stream_basic:stream/1`)

`Term` is any term. (`basic_props:term/1`)

- *The following properties hold globally:*
  `Stream` **is not further instantiated.** (`basic_props:not_further_inst/2`)
  All calls of the form `write(Stream,Term)` **are deterministic.** (`native_props:is_det/1`)

write/1: PREDICATE

Usage: `write(Term)` ⟨• ISO •⟩

Behaves like `current_output(S), write(S,Term)`.

- *The following properties should hold at call time:*
  `Term` **is any term.** (`basic_props:term/1`)
- *The following properties hold globally:*
  All calls of the form `write(Term)` **are deterministic.** (`native_props:is_det/1`)

writeq/2: PREDICATE

Usage: `writeq(Stream,Term)` ⟨• ISO •⟩

Behaves like `write_term(Stream, Term, [quoted(true), numbervars(true)])`.

- *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
  `Term` **is any term.** (`basic_props:term/1`)
- *The following properties hold globally:*
  `Stream` **is not further instantiated.** (`basic_props:not_further_inst/2`)
  All calls of the form `writeq(Stream,Term)` **are deterministic.** (`native_props:is_det/1`)

writeq/1: PREDICATE

Usage: `writeq(Term)` ⟨• ISO •⟩

Behaves like `current_output(S), writeq(S,Term)`.

- *The following properties should hold at call time:*
  `Term` **is any term.** (`basic_props:term/1`)
- *The following properties hold globally:*
  All calls of the form `writeq(Term)` **are deterministic.** (`native_props:is_det/1`)

write_canonical/2: PREDICATE

Usage: `write_canonical(Stream,Term)` ⟨• ISO •⟩

Behaves like `write_term(Stream, Term, [quoted(true), ignore_ops(true)])`. **The output of this predicate can always be parsed by** `read_term/2` **even if the term contains special characters or if operator declarations have changed.**

- *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
  `Term` **is any term.** (`basic_props:term/1`)
- *The following properties should hold globally:*
  `Stream` **is not further instantiated.** (`basic_props:not_further_inst/2`)

write_canonical/1: PREDICATE

Usage: `write_canonical(Term)` ⟨• ISO •⟩

Behaves like `current_output(S), write_canonical(S,Term).`

– *The following properties should hold at call time:*

`Term` **is any term.** (`basic_props:term/1`)

print/2: PREDICATE

Usage: `print(Stream,Term)`

Behaves like `write_term(Stream, Term, [numbervars(true), portrayed(true)]).`

– *The following properties should hold at call time:*

`Stream` **is an open stream.** (`stream_basic:stream/1`)

`Term` **is any term.** (`basic_props:term/1`)

– *The following properties should hold globally:*

`Stream` **is not further instantiated.** (`basic_props:not_further_inst/2`)

print/1: PREDICATE

Usage: `print(Term)`

Behaves like `current_output(S), print(S,Term).`

– *The following properties should hold at call time:*

`Term` **is any term.** (`basic_props:term/1`)

printq/2: PREDICATE

Usage: `printq(Stream,Term)`

Behaves like `write_term(Stream, Term, [quoted(true), numbervars(true), portrayed(true)]).`

– *The following properties should hold at call time:*

`Stream` **is an open stream.** (`stream_basic:stream/1`)

`Term` **is any term.** (`basic_props:term/1`)

– *The following properties should hold globally:*

`Stream` **is not further instantiated.** (`basic_props:not_further_inst/2`)

printq/1: PREDICATE

Usage: `printq(Term)`

Behaves like `current_output(S), printq(S,Term).`

– *The following properties should hold at call time:*

`Term` **is any term.** (`basic_props:term/1`)

portray_clause/2: PREDICATE

Usage: `portray_clause(Stream,Clause)`

Outputs the clause `Clause` onto `Stream`, pretty printing its variables and using indentation, including a period at the end. This predicate is used by `listing/0`.

- *The following properties should hold at call time:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

  `Clause` **is any term.** (`basic_props:term/1`)

- *The following properties should hold globally:*

  `Stream` **is not further instantiated.** (`basic_props:not_further_inst/2`)

portray_clause/1: PREDICATE

Usage: `portray_clause(Clause)`

Behaves like `current_output(S), portray_clause(S,Term)`.

- *The following properties should hold at call time:*

  `Clause` **is any term.** (`basic_props:term/1`)

numbervars/3: PREDICATE

Usage: `numbervars(Term,N,M)`

Unifies each of the variables in term `Term` with a term of the form `'$VAR'(I)` where `I` is an integer from `N` onwards. `M` is unified with the last integer used plus 1. If the resulting term is output with a write option `numbervars(true)`, in the place of the variables in the original term will be printed a variable name consisting of a capital letter possibly followed by an integer. When `N` is 0 you will get the variable names A, B, ..., Z, A1, B1, etc.

- *The following properties should hold at call time:*

  `N` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Term` **is any term.** (`basic_props:term/1`)

  `N` **is an integer.** (`basic_props:int/1`)

  `M` **is any term.** (`basic_props:term/1`)

- *The following properties should hold upon exit:*

  `Term` **is any term.** (`basic_props:term/1`)

  `N` **is an integer.** (`basic_props:int/1`)

  `M` **is an integer.** (`basic_props:int/1`)

prettyvars/1: PREDICATE

Usage: `prettyvars(Term)`

Similar to `numbervars(Term,0,_)`, except that singleton variables in `Term` are unified with `'$VAR'('_')`, so that when the resulting term is output with a write option `numbervars(true)`, in the place of singleton variables `_` is written. This predicate is used by `portray_clause/2`.

- *The following properties should hold at call time:*

  `Term` **is any term.** (`basic_props:term/1`)

printable_char/1: PREDICATE

Usage: `printable_char(Char)`

`Char` is the code of a character which can be printed.

– *The following properties should hold at call time:*

`Char` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`Char` is an integer which is a character code. (`basic_props:character_code/1`)

write_attribute/1: PREDICATE

No further documentation available for this predicate.

150.3 Documentation on multifiles

define_flag/3: PREDICATE

Defines flags as follows:

```
define_flag(write_strings,[on,off],off).
```

(See Chapter 155 [Runtime system control and flags], page 683).

If flag is `on`, lists which may be written as strings are.

Usage: `define_flag(Flag,FlagValues,Default)`

– *The following properties hold upon exit:*

`Flag` is an atom. (`basic_props:atm/1`)

Define the valid flag values (`basic_props:flag_values/1`)

The predicate is *multifile*.

portray_attribute/2: PREDICATE

Usage: `portray_attribute(Attr,Var)`

A user defined predicate. When an attributed variable `Var` is about to be printed, this predicate receives the variable and its attribute `Attr`. The predicate should either print something based on `Attr` or `Var`, or do nothing and fail. In the latter case, the default printer (`write/1`) will print the attributed variable like an unbound variable, e.g. `_673`.

– *The following properties should hold at call time:*

`Attr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

`Var` is a free variable. (`term_typing:var/1`)

The predicate is *multifile*.

portray/1: PREDICATE

Usage: `portray(T)`

A user defined predicate. This should either print the `Term` and succeed, or do nothing and fail. In the latter case, the default printer (`write/1`) will print the `Term`.

– *The following properties should hold at call time:*

`T` is any term. (`basic_props:term/1`)

– *The following properties hold upon exit:*

`T` is any term. (`basic_props:term/1`)

The predicate is *multifile*.

150.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `native_props`, `operators`, `sort`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `nativeprops`, `isomodes`, `define_flag`.

151 Defining operators

Author(s): Daniel Cabeza (modifications and documentation, adapted from SICStus 0.6 code), Manuel Carro (modifications and documentation).

Operators allow writing terms in a more clear way than the standard functional notation. Standard operators in Ciao are defined by this predicate (but note that the compiler itself defines more operators at compile time):

```
standard_ops :-
    op(1200,xfx,[:-]),
    op(1200,fx,[:-,?-]),
    op(1100,xfy,[;]),
    op(1050,xfy,[->]),
    op(1000,xfy,[',']),
    op(900,fy,[\+]),
    op(700,xfx,[=,\=,==,\==,@<,@>,@=<,@>=,=..,is,=:=,=\=,<,=<,>,>=]),
    op(550,xfx,[:]),
    op(500,yfx,[+,-,/\,\/,#]),
    op(500,fy,[++,--]),
    op(400,yfx,[*,/,//,rem,mod,<<,>>]),
    op(200,fy,[+,-,\]),
    op(200,xfx,[**]),
    op(200,xfy,[^]).
```

151.1 Usage and interface

- **Library usage:**
  `:- use_module(library(operators)).`
- **Exports:**
  - *Predicates:*
    `op/3`,
    `current_op/3`, `current_prefixop/3`, `current_infixop/4`, `current_postfixop/3`, `standard_ops/0`.

151.2 Documentation on exports

op/3: PREDICATE

`op(Precedence,Type,Name)`

Declares the atom `Name` to be an operator of the stated `Type` and `Precedence` (0 =< `Precedence` =< 1200). `Name` may also be a list of atoms in which case all of them are declared to be operators. If `Precedence` is 0 then the operator properties of `Name` (if any) are cancelled. Note that, unlike in ISO-Prolog, it is allowed to define two operators with the same name, one infix and the other postfix.

Usage: ⟨• ISO •⟩

- *The following properties should hold at call time:*

  `Precedence` **is an integer.** (`basic_props:int/1`)

  `Type` **specifies the type and associativity of an operator.** (`basic_props:operator_specifier/1`)

`Name` is an atom or a list of atoms. (`basic_props:atm_or_atm_list/1`)

– *The following properties should hold globally:*

This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

current_op/3: PREDICATE

`current_op(Precedence,Type,Op)`

The atom `Op` is currently an operator of type `Type` and precedence `Precedence`. Neither `Op` nor the other arguments need be instantiated at the time of the call; i.e., this predicate can be used to generate as well as to test.

Usage: ⟨ • ISO • ⟩

– *Call and exit should be compatible with:*

`Precedence` is an integer. (`basic_props:int/1`)

`Type` specifies the type and associativity of an operator. (`basic_props:operator_specifier/1`)

`Op` is an atom. (`basic_props:atm/1`)

– *The following properties should hold upon exit:*

`Precedence` is an integer. (`basic_props:int/1`)

`Type` specifies the type and associativity of an operator. (`basic_props:operator_specifier/1`)

`Op` is an atom. (`basic_props:atm/1`)

– *The following properties should hold globally:*

This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

current_prefixop/3: PREDICATE

`current_prefixop(Op,Less,Precedence)`

Similar to `current_op/3`, but it concerns only the prefix operators. It returns only one solution. Not a predicate for general use.

Usage:

– *Call and exit should be compatible with:*

`Op` is an atom. (`basic_props:atm/1`)

`Less` is an integer. (`basic_props:int/1`)

`Precedence` is an integer. (`basic_props:int/1`)

– *The following properties should hold upon exit:*

`Op` is an atom. (`basic_props:atm/1`)

`Less` is an integer. (`basic_props:int/1`)

`Precedence` is an integer. (`basic_props:int/1`)

– *The following properties should hold globally:*

This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

current_infixop/4: PREDICATE

`current_infixop(Op,LeftLess,Prec,RightLess)`

Similar to `current_op/3`, but it concerns only infix operators. It returns only one solution. Not a predicate for general use.

Usage:

- *Call and exit should be compatible with:*
  `Op` **is an atom.** (`basic_props:atm/1`)
  `LeftLess` **is an integer.** (`basic_props:int/1`)
  `Prec` **is an integer.** (`basic_props:int/1`)
  `RightLess` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold upon exit:*
  `Op` **is an atom.** (`basic_props:atm/1`)
  `LeftLess` **is an integer.** (`basic_props:int/1`)
  `Prec` **is an integer.** (`basic_props:int/1`)
  `RightLess` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

current_postfixop/3: PREDICATE

`current_postfixop(Op,Less,Precedence)`

Similar to `current_op/3`**, but it concerns only the postfix operators. It returns only one solution. Not a predicate for general use.**

Usage:

- *Call and exit should be compatible with:*
  `Op` **is an atom.** (`basic_props:atm/1`)
  `Less` **is an integer.** (`basic_props:int/1`)
  `Precedence` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold upon exit:*
  `Op` **is an atom.** (`basic_props:atm/1`)
  `Less` **is an integer.** (`basic_props:int/1`)
  `Precedence` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

standard_ops/0: PREDICATE

No further documentation available for this predicate.

151.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `datafacts`.

152 Formatted output

Author(s): The Ciao Development Team.

The `format` family of predicates is due to Quintus Prolog. They act as a Prolog interface to the C `stdio` function `printf()`, allowing formatted output.

Output is formatted according to an output pattern which can have either a format control sequence or any other character, which will appear verbatim in the output. Control sequences act as place-holders for the actual terms that will be output. Thus

```
?- format("Hello ~q!",world).
```

will print `Hello world!`.

If there is only one item to print it may be supplied alone. If there are more they have to be given as a list. If there are none then an empty list should be supplied. There has to be as many items as control characters.

The character `~` introduces a control sequence. To print a `~` verbatim just repeat it:

```
?- format("Hello ~~world!", []).
```

will result in `Hello ~world!`.

A format may be spread over several lines. The control sequence `\c` followed by a ⟨LFD⟩ will translate to the empty string:

```
?- format("Hello \c
world!", []).
```

will result in `Hello world!`.

152.1 Usage and interface

- **Library usage:**
  `:- use_module(library(format)).`
- **Exports:**
  - *Predicates:*
    `format/2`, `format/3`, `sformat/3`, `format_to_string/3`.
  - *Regular Types:*
    `format_control/1`.

152.2 Documentation on exports

format/2: PREDICATE

Usage: `format(Format,Arguments)`

Print `Arguments` onto current output stream according to format `Format`.

- *The following properties should hold at call time:*
  `Format` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (`format:format_control/1`)

Other properties:

`format(C,A)`

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP as `format(C,A)`. (`basic_props:native/2`)

format/3: PREDICATE

Usage: `format(Stream,Format,Arguments)`

Print `Arguments` **onto** `Stream` **according to format** `Format`.

- *The following properties should hold at call time:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

  `Format` **is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with** `name/2`. (`format:format_control/1`)

Other properties:

`format(S,C,A)`

- *The following properties hold globally:*

  This predicate is understood natively by CiaoPP as `format(S,C,A)`. (`basic_props:native/2`)

sformat/3: PREDICATE

Usage: `sformat(String,Format,Arguments)`

Same as `format_to_string(Format, Arguments, String)` **(note the different argument order).**

- *The following properties should hold at call time:*

  `Format` **is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with** `name/2`. (`format:format_control/1`)

- *The following properties should hold upon exit:*

  `String` **is a string (a list of character codes).** (`basic_props:string/1`)

format_to_string/3: PREDICATE

Usage: `format_to_string(Format,Arguments,String)`

Print `Arguments` **onto current string** `String` **according to format** `Format`**. This predicate is similar to the format/2, but the result is stored in a string.**

- *The following properties should hold at call time:*

  `Format` **is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with** `name/2`. (`format:format_control/1`)

  `Arguments` **is a list.** (`basic_props:list/1`)

- *The following properties should hold upon exit:*

  `String` **is a string (a list of character codes).** (`basic_props:string/1`)

format_control/1: REGTYPE

The general format of a control sequence is `~NC`. The character `C` determines the type of the control sequence. `N` is an optional numeric argument. An alternative form of `N` is `*`. `*` implies that the next argument in `Arguments` should be used as a numeric argument in the control sequence. Example:

```
?- format("Hello~4cworld!", [0'x]).
```

and

```
?- format("Hello~*cworld!", [4,0'x]).
```

both produce

```
Helloxxxxworld!
```

The following control sequences are available.

- **`~a` The argument is an atom. The atom is printed without quoting.**
- **`~Nc` (Print character.) The argument is a number that will be interpreted as an ASCII code. `N` defaults to one and is interpreted as the number of times to print the character.**
- `~Ne`
- `~NE`
- `~Nf`
- `~Ng`
- **`~NG` (Print float). The argument is a float. The float and `N` will be passed to the C `printf()` function as**

  ```
  printf("%.Ne", Arg)
  printf("%.NE", Arg)
  printf("%.Nf", Arg)
  printf("%.Ng", Arg)
  printf("%.NG", Arg)
  ```

  If `N` is not supplied the action defaults to

  ```
  printf("%e", Arg)
  printf("%E", Arg)
  printf("%f", Arg)
  printf("%g", Arg)
  printf("%G", Arg)
  ```

- **`~Nd` (Print decimal.) The argument is an integer. `N` is interpreted as the number of digits after the decimal point. If `N` is 0 or missing, no decimal point will be printed. Example:**

  ```
  ?- format("Hello ~1d world!", [42]).
  ?- format("Hello ~d world!", [42]).
  ```

  will print as

  ```
  Hello 4.2 world!
  Hello 42 world!
  ```

  respectively.

- **`~ND` (Print decimal.) The argument is an integer. Identical to `~Nd` except that , will separate groups of three digits to the left of the decimal point. Example:**

  ```
  ?- format("Hello ~1D world!", [12345]).
  ```

  will print as

  ```
  Hello 1,234.5 world!
  ```

- `~Nr` (Print radix.) The argument is an integer. `N` is interpreted as a radix. `N` should be >= 2 and <= 36. If `N` is missing the radix defaults to 8. The letters `a-z` will denote digits larger than 9. Example:

  ```
  ?- format("Hello ~2r world!", [15]).
  ?- format("Hello ~16r world!", [15]).
  ```

  will print as

  ```
  Hello 1111 world!
  Hello f world!
  ```

  respectively.
- `~NR` (Print radix.) The argument is an integer. Identical to `~Nr` except that the letters `A-Z` will denote digits larger than 9. Example:

  ```
  ?- format("Hello ~16R world!", [15]).
  ```

  will print as

  ```
  Hello F world!
  ```
- `~Ns` (Print string.) The argument is a list of ASCII codes. Exactly `N` characters will be printed. `N` defaults to the length of the string. Example:

  ```
  ?- format("Hello ~4s ~4s!", ["new","world"]).
  ?- format("Hello ~s world!", ["new"]).
  ```

  will print as

  ```
  Hello new  worl!
  Hello new world!
  ```

  respectively.
- `~i` (Ignore argument.) The argument may be of any type. The argument will be ignored. Example:

  ```
  ?- format("Hello ~i~s world!", ["old","new"]).
  ```

  will print as

  ```
  Hello new world!
  ```
- `~k` (Print canonical.) The argument may be of any type. The argument will be passed to `write_canonical/2` (Chapter 150 [Term output], page 657). Example:

  ```
  ?- format("Hello ~k world!", [[a,b,c]]).
  ```

  will print as

  ```
  Hello .(a,.(b,.(c,[]))) world!
  ```
- `~p` (print.) The argument may be of any type. The argument will be passed to `print/2` (Chapter 150 [Term output], page 657). Example:

  suposing the user has defined the predicate

  ```
  :- multifile portray/1.
  portray([X|Y]) :- print(cons(X,Y)).
  ```

  then

  ```
  ?- format("Hello ~p world!", [[a,b,c]]).
  ```

  will print as

  ```
  Hello cons(a,cons(b,cons(c,[]))) world!
  ```
- `~q` (Print quoted.) The argument may be of any type. The argument will be passed to `writeq/2` (Chapter 150 [Term output], page 657). Example:

  ```
  ?- format("Hello ~q world!", [['A','B']]).
  ```

  will print as

```
Hello ['A','B'] world!
```

- `~w` (write.) The argument may be of any type. The argument will be passed to `write/2` (Chapter 150 [Term output], page 657). Example:

  ```
  ?- format("Hello ~w world!", [['A','B']]).
  ```

  will print as

  ```
  Hello [A,B] world!
  ```

- `~Nn` (Print newline.) Print `N` newlines. `N` defaults to 1. Example:

  ```
  ?- format("Hello ~n world!", []).
  ```

  will print as

  ```
  Hello
   world!
  ```

- `~N` (Fresh line.) Print a newline, if not already at the beginning of a line.
- `~~` (Print tilde.) Prints `~`

The following control sequences are also available for compatibility, but do not perform any useful functions.

- `~N|` (Set tab.) Set a tab stop at position `N`, where `N` defaults to the current position, and advance the current position there.
- `~N+` (Advance tab.) Set a tab stop at `N` positions past the current position, where `N` defaults to 8, and advance the current position there.
- `~Nt` (Set fill character.) Set the fill character to be used in the next position movement to `N`, where `N` defaults to ⟨SPC⟩.

Usage: `format_control(C)`

`C` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`.

– *The following properties should hold globally:*

  Documentation is still incomplete: `format_control(C)` may not conform the functionality documented. (`doc_props:doc_incomplete/1`)

152.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

  `datafacts_rt`, `streams`, `write`, `system`, `doc_props`.

– *Packages:*

  `prelude`, `initial`, `condcomp`, `dcg`, `assertions`, `assertions/assertions_basic`, `isomodes`, `datafacts`.

153 Reading/writting list of terms

Author(s): The Ciao Development Team.

This module implements predicates to read/write list of terms (to streams and files).

153.1 Usage and interface

- **Library usage:**
  `:- use_module(library(terms_io)).`
- **Exports:**
  - *Predicates:*
    `file_to_terms/2`, `file_to_terms/3`, `terms_to_file/2`, `term_write/1`.

153.2 Documentation on exports

file_to_terms/2: PREDICATE

Usage: `file_to_terms(File,Terms)`

Unifies `Terms` with the list of all terms in `File`.

- *The following properties should hold at call time:*
  `File` **is a source name.** (`stream_basic:sourcename/1`)
  `Terms` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Terms` **is a list.** (`basic_props:list/1`)

file_to_terms/3: PREDICATE

Usage: `file_to_terms(File,Terms,Terms0)`

Read all terms in into the difference list `Terms-Terms0`.

- *The following properties should hold at call time:*
  `File` **is a source name.** (`stream_basic:sourcename/1`)

terms_to_file/2: PREDICATE

Usage: `terms_to_file(File,Terms)`

Writes the terms in list `Terms` (including the ending '.') onto file `File`.

- *The following properties should hold at call time:*
  `File` **is a source name.** (`stream_basic:sourcename/1`)
  `Terms` **is a list.** (`basic_props:list/1`)

term_write/1: PREDICATE

`term_write(Term)`

Output `Term` in a way that a `read/1` will be able to read it back, even if operators change.

153.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `read`, `stream_utils`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

154 Message printing primitives

Author(s): Daniel Cabeza, Edison Mera (improvements), Jose F. Morales, Manuel V. Hermenegildo.

This module provides predicates for printing in a unified way informational messages. It is designed to be small and do not have (strict) dependencies with other larger printing and formatting libraries in the system.

154.1 Usage and interface

- **Library usage:**
  `:- use_module(engine(messages_basic)).`
- **Exports:**
  - *Predicates:*
    `message/2`, `message_lns/4`, `messages/1`, `message_type_visible/1`, `lformat/1`, `display_list/1`, `add_lines/4`.
  - *Regular Types:*
    `message_info/1`, `message_type/1`.

154.2 Documentation on exports

message/2: PREDICATE

Usage: `message(Type,Message)`

Output to standard error `Message`, which is of type `Type`. The `quiet` *prolog flag* (see Chapter 155 [Runtime system control and flags], page 683) controls which messages are actually output, depending on its type. Also, for `error`, `warning`, and `note` messages, a prefix is output which denotes the severity of the message.

- *The following properties should hold at call time:*

  `Type` is one of the accepted message types. (`messages_basic:message_type/1`)

  `Message` is an item or a list of items from the following:

  `$$(String)`
  : `String` is a string, which is output with `write_string/1`.

  `''(Term)`
  : `Term` is output quoted. If the module `write` is loaded, the term is output with `writeq/1`, else with `displayq/1`.

  `~~(Term)`
  : `Term` is output unquoted. If the module `write` is loaded, the term is output with `write/1`, else with `display/1`.

  `''({Term})`
  : `Term` is output quoted. If the module `write` is loaded, the term is output with `printq/1`, else with `displayq/1`.

  `{Term}`
  : `Term` is output unquoted. If the module `write` is loaded, the term is output with `print/1`, else with `display/1`.

  `[](Term)`
  : `Term` is recursively output as a message, can be an item or a list of items from this list.

`Term` Any other term is output with `display/1`.

(`messages_basic:lformat_text/1`)

message_lns/4: PREDICATE

Usage: `message_lns(Type,L0,L1,Message)`

Output to standard error `Message`, which is of type `Type`, and occurs between lines `L0` and `L1`. This is the same as `message/2`, but printing the lines where the message occurs in a unified way (this is useful because automatic tools such as the emacs mode know how to parse them).

– *The following properties should hold at call time:*

`Type` is one of the accepted message types. (`messages_basic:message_type/1`)

`L0` is a non-negative integer. (`basic_props:nnegint/1`)

`L1` is a non-negative integer. (`basic_props:nnegint/1`)

`Message` is an item or a list of items from the following:

`$$(String)`
String is a string, which is output with `write_string/1`.

`''(Term)` `Term` is output quoted. If the module `write` is loaded, the term is output with `writeq/1`, else with `displayq/1`.

`~~(Term)` `Term` is output unquoted. If the module `write` is loaded, the term is output with `write/1`, else with `display/1`.

`''({Term})`
`Term` is output quoted. If the module `write` is loaded, the term is output with `printq/1`, else with `displayq/1`.

`{Term}` `Term` is output unquoted. If the module `write` is loaded, the term is output with `print/1`, else with `display/1`.

`[](Term)` `Term` is recursively output as a message, can be an item or a list of items from this list.

`Term` Any other term is output with `display/1`.

(`messages_basic:lformat_text/1`)

messages/1: PREDICATE

Usage: `messages(Messages)`

Print each element in `Messages` using `message/2` or `message_lns/4` predicates.

– *The following properties should hold at call time:*

`Messages` is a list of `message_infos`. (`basic_props:list/2`)

message_type_visible/1: PREDICATE

Usage: `message_type_visible(Type)`

Succeeds if message type `Type` is visible according to the current value of the `quiet` *prolog flag*

– *The following properties should hold at call time:*

`Type` is one of the accepted message types. (`messages_basic:message_type/1`)

lformat/1: PREDICATE

Usage: `lformat(T)`

Output a formatted text `T`. See `lformat_text/1` for a description of the text formatting.

– *The following properties should hold at call time:*

  `T` is an item or a list of items from the following:

  `$$(String)`
  : **`String` is a string, which is output with `write_string/1`.**

  `''(Term)`
  : **`Term` is output quoted. If the module `write` is loaded, the term is output with `writeq/1`, else with `displayq/1`.**

  `~~(Term)`
  : **`Term` is output unquoted. If the module `write` is loaded, the term is output with `write/1`, else with `display/1`.**

  `''({Term})`
  : **`Term` is output quoted. If the module `write` is loaded, the term is output with `printq/1`, else with `displayq/1`.**

  `{Term}`
  : **`Term` is output unquoted. If the module `write` is loaded, the term is output with `print/1`, else with `display/1`.**

  `[](Term)`
  : **`Term` is recursively output as a message, can be an item or a list of items from this list.**

  `Term`
  : **Any other term is output with `display/1`.**

  (`messages_basic:lformat_text/1`)

display_list/1: PREDICATE

`display_list(List)`

Outputs `List`. If `List` is a list, do `display/1` on each of its elements, else do `display/1` on `List`.

message_info/1: REGTYPE

Usage:

The type of the elements to be printed using the `messages/1` predicate. Defined as

```
message_info(message_lns(Source,Ln0,Ln1,Type,Text)) :-
    atm(Source),
    nnegint(Ln0),
    nnegint(Ln1),
    message_type(Type),
    lformat_text(Text).
message_info(message(Type,Text)) :-
    atm(Type),
    lformat_text(Text).
message_info(error(Text)) :-
    lformat_text(Text).
message_info(warning(Text)) :-
    lformat_text(Text).
message_info(note(Text)) :-
    lformat_text(Text).
```

```
    message_info(message(Text)) :-
        lformat_text(Text).
    message_info(debug(Text)) :-
        lformat_text(Text).
.
```

message_type/1: REGTYPE

`message_type(M)`

```
message_type(error).
message_type(error0).
message_type(warning).
message_type(note).
message_type(user).
message_type(inform).
message_type(debug).
message_type(passed).
message_type(failed).
message_type(aborted).
message_type(testing).
```

Usage: `message_type(M)`

`M` is one of the accepted message types.

add_lines/4: PREDICATE

No further documentation available for this predicate.

154.3 Documentation on internals

lformat_text/1: REGTYPE

Usage: `lformat_text(Message)`

`Message` is an item or a list of items from the following:

`$$(String)`
: `String` is a string, which is output with `write_string/1`.

`''(Term)`
: `Term` is output quoted. If the module `write` is loaded, the term is output with `writeq/1`, else with `displayq/1`.

`~~(Term)`
: `Term` is output unquoted. If the module `write` is loaded, the term is output with `write/1`, else with `display/1`.

`''({Term})`
: `Term` is output quoted. If the module `write` is loaded, the term is output with `printq/1`, else with `displayq/1`.

`{Term}`
: `Term` is output unquoted. If the module `write` is loaded, the term is output with `print/1`, else with `display/1`.

`[](Term)`
: `Term` is recursively output as a message, can be an item or a list of items from this list.

`Term`
: Any other term is output with `display/1`.

154.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `native_props`, `stream_utils`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nativeprops`, `nortchecks`.

155 Runtime system control and flags

Author(s): Manuel Carro, Daniel Cabeza, Jose F. Morales, Manuel Hermenegildo (improved documentation), Mats Carlsson (original author).

This module implements some predicates which provide access to and control of many parameters of the Ciao runtime, including internal statistics, loaded modules, special properties of the predicates, garbage collection, etc.

Some of these facilities are accessed or controlled via specific predicates (see later) while in other cases such access and control is preformed through the mechanism of flags. Such flags contain values for some global parameters of the system and also control the behavior of some system or library predicates. Each flag has a name and an associated predefined value, and except some system flags which are fixed, in general their associated value is changeable.

Flags can be given values at compile time, via *declarations*, and then the changes affect only the module being processed. Predicates are also available to consult and change flag values at run time, in which case their effect is global from that moment on in the execution (see `prolog_flag/3` and related predicates below). Note that corresponding flag-processing declarations and predicates have the same predicate name; only the run-time version is documented.

Some flags are pre-defined in the system. In addition, new flags can be also be defined. The mechanism to do this is described in `define_flag/3`. Through this mechanism, many Ciao modules and language extensions define additional flags. Flags that are predefined in the system (engine) are documented here and the rest are generally documented in the modules/libraries that define them.

The predefined flags in the system are the following:

`version`
: The Ciao version, as a term `ciao(Version,Patch,CommitInfo)`. `Version` and `Patch` are atoms. `CommitInfo` is a structure describing the commit information (branch, id, date, description). Unchangeable.

`dialect`
: Value set to `ciao`. Used for compatibility with other systems when in Prolog mode. Unchangeable.

`argv`
: Its value is a list of atoms representing the program arguments supplied when the current executable was invoked. This is the value to which the argument of the `main/1` predicate is instantiated at executable startup (which is the usual way to access the list of arguments). Unchangeable.

`bounded`
: It is `false`, to denote that the range of integers can be considered infinite (but see `int/1`). Unchangeable. ⟨• ISO •⟩

`fileerrors`
: If `on`, predicates handling files produce errors (throw exceptions) when a file is non-existent or an operation is not allowed. If `off`, a failure will occur instead for those conditions. Initially `on`.

`gc`
: Controls whether garbage collection is performed. May be `on` (default) or `off`.

`gc_margin`
: `Margin` is an integer so that, if less than `Margin` kilobytes are reclaimed in a garbage collection, then the size of the garbage-collected area should be increased. Also, no garbage collection is attempted unless the garbage-collected area has at least `Margin` kilobytes. Initially set to `500`.

`gc_trace`
: Governs garbage collection trace messages. An element of `[on,off,terse,verbose]`. Initially `off`.

`integer_rounding_function`
: It is `toward_zero`, so that `-1 =:= -3//2` succeeds. Unchangeable. ⟨• ISO •⟩

`max_arity`
: It is 255, so that no compound term (or predicate) can have more than this number of arguments. Unchangeable. ⟨ • ISO • ⟩

`quiet`
: Controls which messages issued using the `messages_basic` library are actually written. As the system uses that library to emit its messages, this flag controls the *verbosity* of the system. Possible states of the flag are:

  `on` No messages are reported.

  `error` Only error messages are reported.

  `warning` Only error and warning messages are reported.

  `off` All messages are reported, except debug messages. This is the default state.

  `debug` All messages, including debug messages, are reported. This is only intended for the system implementors.

`unknown`
: Controls the action to be taken when an undefined predicate is called. The possible states of the flag are:

  `error` An error is thrown with the error term `existence_error(procedure, F/A)`.

  `fail` The call simply fails.

  `warning` A warning is written and the call fails.

  The state is initially `error`. ⟨ • ISO • ⟩

155.1 Usage and interface

- **Library usage:**

  `:- use_module(engine(runtime_control)).`
- **Exports:**
  - *Predicates:*

    `statistics/0`, `statistics/2`, `time/1`, `current_atom/1`, `new_atom/1`, `current_module/1`, `module_split/3`, `predicate_property/2`, `predicate_property/3`, `garbage_collect/0`, `set_prolog_flag/2`, `current_prolog_flag/2`, `prolog_flag/3`, `push_prolog_flag/2`, `pop_prolog_flag/1`, `prompt/2`, `fileerrors/0`, `nofileerrors/0`, `gc/0`, `nogc/0`.
  - *Regular Types:*

    `time_option/1`, `tick_option/1`, `clockfreq_option/1`, `memory_option/1`, `gc_option/1`, `symbol_option/1`, `time_result/1`, `tick_result/1`, `clockfreq_result/1`, `memory_result/1`, `gc_result/1`, `symbol_result/1`.
  - *Multifiles:*

    `define_flag/3`.

155.2 Documentation on exports

statistics/0: PREDICATE

Usage:

Prints statistics about the system.

statistics/2: PREDICATE

Usage 1: `statistics(Tick_option,Tick_result)`

Gather information about clock ticks (either run, user, system or wall tick) since the last consultation or since the start of the program. A tick is the smallest amount of time that the clock can measure.

- *The following properties should hold at call time:*

  `Tick_option` **is an option to get information about execution ticks.**

  ```
  tick_option(runtick).
  tick_option(usertick).
  tick_option(systemtick).
  tick_option(walltick).
  ```

  (`runtime_control:tick_option/1`)

- *The following properties should hold upon exit:*

  `Tick_option` **is an option to get information about execution ticks.**

  ```
  tick_option(runtick).
  tick_option(usertick).
  tick_option(systemtick).
  tick_option(walltick).
  ```

  (`runtime_control:tick_option/1`)

  `Tick_result` **is a two-element list of numbers. The first number is the number of ticks since the start of the execution; the second number is the number of ticks since the previous consultation to tick.** (`runtime_control:tick_result/1`)

Usage 2: `statistics(Clockfreq_option,Clockfreq_result)`

Gather information about frequency of the clocks used to measure the ticks (either run-user, system or wall clock). Results are returned in Hertz. This value also can be defined as the amount of ticks that a clock can measure in one second.

- *The following properties should hold at call time:*

  `Clockfreq_option` **is and option to get information about the frequency of clocks used to get the ticks.**

  ```
  clockfreq_option(runclockfreq).
  clockfreq_option(userclockfreq).
  clockfreq_option(systemclockfreq).
  clockfreq_option(wallclockfreq).
  ```

  (`runtime_control:clockfreq_option/1`)

- *The following properties should hold upon exit:*

  `Clockfreq_option` **is and option to get information about the frequency of clocks used to get the ticks.**

  ```
  clockfreq_option(runclockfreq).
  clockfreq_option(userclockfreq).
  clockfreq_option(systemclockfreq).
  clockfreq_option(wallclockfreq).
  ```

  (`runtime_control:clockfreq_option/1`)

  `Clockfreq_result` **is a number which gives the frequency in Hertz used by the clock get the ticks.** (`runtime_control:clockfreq_result/1`)

Usage 3: `statistics(Time_option,Time_result)`

Gather information about time (either process time or wall time) since the last consultation or since start of program. Results are returned in milliseconds. Note that internally, time is calculated as:

```
Time_result = (Tick_result / Clockfreq_result) * 1000
```

- *The following properties should hold at call time:*

  `Time_option` **is an option to get information about execution time.**

  ```
  time_option(runtime).
  time_option(usertime).
  time_option(systemtime).
  time_option(walltime).
  ```

  (`runtime_control:time_option/1`)
- *The following properties should hold upon exit:*

  `Time_option` **is an option to get information about execution time.**

  ```
  time_option(runtime).
  time_option(usertime).
  time_option(systemtime).
  time_option(walltime).
  ```

  (`runtime_control:time_option/1`)

  `Time_result` **is a two-element list of numbers. The first number is the time since the start of the execution; the second number is the time since the previous time consultation to time.** (`runtime_control:time_result/1`)

Usage 4: `statistics(Memory_option,Memory_result)`

Gather information about memory consumption.

- *The following properties should hold at call time:*

  `Memory_option` **is an option to get information about memory usage.**

  ```
  memory_option(memory).
  memory_option(symbols).
  memory_option(program).
  memory_option(global_stack).
  memory_option(local_stack).
  memory_option(trail).
  memory_option(choice).
  ```

  (`runtime_control:memory_option/1`)
- *The following properties should hold upon exit:*

  `Memory_option` **is an option to get information about memory usage.**

  ```
  memory_option(memory).
  memory_option(symbols).
  memory_option(program).
  memory_option(global_stack).
  memory_option(local_stack).
  memory_option(trail).
  memory_option(choice).
  ```

  (`runtime_control:memory_option/1`)

  `Memory_result` **is a two-element list of integers. The first element is the space taken up by the option selected, measured in bytes; the second integer is zero for program space (which grows as necessary), and the amount of free space otherwise.** (`runtime_control:memory_result/1`)

Usage 5: `statistics(GC_option,GC_result)`

Gather information about garbage collection.

– *The following properties should hold at call time:*

`GC_option` **is an option to get information about garbage collection.**

```
gc_option(garbage_collection).
gc_option(stack_shifts).
```

(`runtime_control:gc_option/1`)

– *The following properties should hold upon exit:*

`GC_option` **is an option to get information about garbage collection.**

```
gc_option(garbage_collection).
gc_option(stack_shifts).
```

(`runtime_control:gc_option/1`)

`GC_result` **is a tree-element list of integers, related to garbage collection and memory management. When `stack_shifts` is selected, the first one is the number of shifts (reallocations) of the local stack; the second is the number of shifts of the trail; and the third is the time spent in these shifts. When `garbage_collection` is selected, the numbers are, respectively, the number of garbage collections performed, the number of bytes freed, and the time spent in garbage collection.** (`runtime_control:gc_result/1`)

Usage 6: `statistics(Symbol_option,Symbol_result)`

Gather information about number of symbols and predicates.

– *The following properties should hold at call time:*

`Symbol_option` **is an option to get information about the number of symbols in the program.**

```
symbol_option(symbols).
```

(`runtime_control:symbol_option/1`)

– *The following properties should hold upon exit:*

`Symbol_option` **is an option to get information about the number of symbols in the program.**

```
symbol_option(symbols).
```

(`runtime_control:symbol_option/1`)

`Symbol_result` **is a two-element list of integers. The first one is the number of atom, functor, and predicate names in the symbol table. The second is the number of predicates known to be defined (although maybe without clauses).** (`runtime_control:symbol_result/1`)

Usage 7: `statistics(Option,Arg2)`

If `Option` is unbound, it is bound by backtracking to the values of all the other cases.

– *Call and exit should be compatible with:*

`Arg2` **is any term.** (`basic_props:term/1`)

– *The following properties should hold at call time:*

`Option` **is a free variable.** (`term_typing:var/1`)

– *The following properties should hold upon exit:*

`Arg2` **is any term.** (`basic_props:term/1`)

time_option/1: REGTYPE

Usage: `time_option(M)`

`M` **is an option to get information about execution time.**

```
time_option(runtime).
time_option(usertime).
time_option(systemtime).
time_option(walltime).
```

tick_option/1: REGTYPE

Usage: `tick_option(M)`

`M` **is an option to get information about execution ticks.**

```
tick_option(runtick).
tick_option(usertick).
tick_option(systemtick).
tick_option(walltick).
```

clockfreq_option/1: REGTYPE

Usage: `clockfreq_option(M)`

`M` **is and option to get information about the frequency of clocks used to get the ticks.**

```
clockfreq_option(runclockfreq).
clockfreq_option(userclockfreq).
clockfreq_option(systemclockfreq).
clockfreq_option(wallclockfreq).
```

memory_option/1: REGTYPE

Usage: `memory_option(M)`

`M` **is an option to get information about memory usage.**

```
memory_option(memory).
memory_option(symbols).
memory_option(program).
memory_option(global_stack).
memory_option(local_stack).
memory_option(trail).
memory_option(choice).
```

gc_option/1: REGTYPE

Usage: `gc_option(M)`

`M` **is an option to get information about garbage collection.**

```
gc_option(garbage_collection).
gc_option(stack_shifts).
```

symbol_option/1: REGTYPE

Usage: `symbol_option(M)`

`M` **is an option to get information about the number of symbols in the program.**

```
symbol_option(symbols).
```

time_result/1: REGTYPE

Usage: `time_result(Result)`

`Result` is a two-element list of numbers. The first number is the time since the start of the execution; the second number is the time since the previous time consultation to time.

tick_result/1: REGTYPE

Usage: `tick_result(Result)`

`Result` is a two-element list of numbers. The first number is the number of ticks since the start of the execution; the second number is the number of ticks since the previous consultation to tick.

clockfreq_result/1: REGTYPE

Usage: `clockfreq_result(Result)`

`Result` is a number which gives the frequency in Hertz used by the clock get the ticks.

memory_result/1: REGTYPE

Usage: `memory_result(Result)`

`Result` is a two-element list of integers. The first element is the space taken up by the option selected, measured in bytes; the second integer is zero for program space (which grows as necessary), and the amount of free space otherwise.

gc_result/1: REGTYPE

Usage: `gc_result(Result)`

`Result` is a tree-element list of integers, related to garbage collection and memory management. When `stack_shifts` is selected, the first one is the number of shifts (reallocations) of the local stack; the second is the number of shifts of the trail; and the third is the time spent in these shifts. When `garbage_collection` is selected, the numbers are, respectively, the number of garbage collections performed, the number of bytes freed, and the time spent in garbage collection.

symbol_result/1: REGTYPE

Usage: `symbol_result(Result)`

`Result` is a two-element list of integers. The first one is the number of atom, functor, and predicate names in the symbol table. The second is the number of predicates known to be defined (although maybe without clauses).

time/1: PREDICATE

Usage: `time(G)`

Execute `G` and print runtime statistics to the user, for each solution.

Meta-predicate with arguments: `time(goal)`.

current_atom/1: PREDICATE

Usage: `current_atom(Atom)`

Enumerates on backtracking all the existing atoms in the system.

- *The following properties should hold at call time:*

  `Atom` **is a free variable.** (`term_typing:var/1`)

- *The following properties hold upon exit:*

  `Atom` **is an atom.** (`basic_props:atm/1`)

new_atom/1: PREDICATE

Usage: `new_atom(Atom)`

Returns, on success, in `Atom` a new atom, not existing before in the system. The entry argument must be a variable. The idea behind this atom generation is to provide a fast source of identifiers for new objects, concurrent predicates, etc. on the fly.

- *The following properties should hold at call time:*

  `Atom` **is a free variable.** (`term_typing:var/1`)

- *The following properties hold upon exit:*

  `Atom` **is an atom.** (`basic_props:atm/1`)

current_module/1: PREDICATE

This predicate will successively unify its argument with all module names currently loaded. Module names will be simple atoms.

When called using a free variable as argument, it will retrieve on backtracking all modules currently loaded. This is useful when called from the Ciao `toplevel`.

When called using a module name as argument it will check whether the given module is loaded or not. This is useful when called from user programs.

Usage 1: `current_module(Module)`

Retrieves (on backtracking) in `Module` all the loaded modules (either statically or dynamically).

- *The following properties should hold at call time:*

  `Module` **is a free variable.** (`term_typing:var/1`)

- *The following properties should hold upon exit:*

  `Module` **is an internal module identifier** (`basic_props:internal_module_id/1`)

- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

Usage 2: `current_module(Module)`

Checks whether module `Module` is loaded or not (either statically or dynamically).

- *The following properties should hold at call time:*

  `Module` **is a free variable.** (`term_typing:var/1`)

- *The following properties should hold upon exit:*

  `Module` **is an internal module identifier** (`basic_props:internal_module_id/1`)

- *The following properties should hold globally:*

  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

module_split/3: PREDICATE

No further documentation available for this predicate.

predicate_property/2: PREDICATE

Usage: `predicate_property(Head,Property)`

The predicate `Head`, visible from the current module, (a goal) has the property `Property`.

– *The following properties should hold upon exit:*

`Head` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

`Property` is an atom. (`basic_props:atm/1`)

predicate_property/3: PREDICATE

No further documentation available for this predicate.

garbage_collect/0: PREDICATE

Usage:

Forces garbage collection when called.

set_prolog_flag/2: PREDICATE

`set_prolog_flag(FlagName,Value)`

Set existing flag `FlagName` to `Value`. Can also be a declaration (which is then compile-time and module-local).

Usage: ⟨ • ISO • ⟩

– *The following properties should hold at call time:*

`FlagName` is an atom. (`basic_props:atm/1`)

`Value` is any term. (`basic_props:term/1`)

– *The following properties should hold upon exit:*

`FlagName` is an atom. (`basic_props:atm/1`)

`Value` is any term. (`basic_props:term/1`)

current_prolog_flag/2: PREDICATE

`current_prolog_flag(FlagName,Value)`

`FlagName` is an existing flag and `Value` is the value currently associated with it.

Usage:

– *The following properties should hold upon exit:*

`FlagName` is an atom. (`basic_props:atm/1`)

`Value` is any term. (`basic_props:term/1`)

prolog_flag/3: PREDICATE

`prolog_flag(FlagName,OldValue,NewValue)`

`FlagName` is an existing flag. Unify `OldValue` with the value associated with it, and set it to new value `NewValue`. Can also be a declaration (which is then compile-time and module-local).

Usage 1: `prolog_flag(FlagName,OldValue,NewValue)`

- *The following properties should hold at call time:*
  `NewValue` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
- *The following properties should hold upon exit:*
  `FlagName` is an atom. (`basic_props:atm/1`)
  `OldValue` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `NewValue` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Usage 2: `prolog_flag(FlagName,OldValue,NewValue)`

Same as `current_prolog_flag(FlagName, OldValue)`. `OldValue` and `NewValue` must be strictly identical variables.

- *The following properties should hold at call time:*
  `OldValue` is a free variable. (`term_typing:var/1`)
  `NewValue` is a free variable. (`term_typing:var/1`)
  `FlagName` is an atom. (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `OldValue` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `NewValue` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

push_prolog_flag/2: PREDICATE

`push_prolog_flag(Flag,NewValue)`

Same as `set_prolog_flag/2`, but storing the current value of `Flag` to restore it with `pop_prolog_flag/1`. Can also be a declaration (which is then compile-time and module-local).

Usage:

- *The following properties should hold at call time:*
  `Flag` is an atom. (`basic_props:atm/1`)
  `NewValue` is any term. (`basic_props:term/1`)
- *The following properties should hold upon exit:*
  `Flag` is an atom. (`basic_props:atm/1`)
  `NewValue` is any term. (`basic_props:term/1`)

pop_prolog_flag/1: PREDICATE

`pop_prolog_flag(Flag)`

Restore the value of `Flag` previous to the last non-canceled `push_prolog_flag/2` on it. Can also be a declaration (which is then compile-time and module-local).

Usage:

- *The following properties should hold at call time:*
  `Flag` is an atom. (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Flag` is an atom. (`basic_props:atm/1`)

prompt/2: PREDICATE

`prompt(Old,New)`

Allows reading or changing the prompt used for reading in terms.

Usage 1: `prompt(Old,New)`

Unify `Old` with the current prompt for reading, change it to `New`. To read the current prompt just call `prompt(X,X)`.

- *The following properties should hold at call time:*

  `New` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*

  `Old` **is an atom.** (`basic_props:atm/1`)

Usage 2: `prompt(Old,New)`

Unify `Old` with the current prompt for reading without changing it. On calls, `Old` and `New` must be strictly identical variables.

- *The following properties should hold at call time:*

  `Old` **is a free variable.** (`term_typing:var/1`)

  `New` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*

  `Old` **is an atom.** (`basic_props:atm/1`)

  `New` **is an atom.** (`basic_props:atm/1`)

fileerrors/0: PREDICATE

Usage:

Enable reporting of file errors. Equivalent to `set_prolog_flag(fileerrors, on)`

- *The following properties should hold globally:*

  `fileerrors` **is equivalent to** `set_prolog_flag(fileerrors,on)`. (`basic_props:equiv/2`)

nofileerrors/0: PREDICATE

Usage:

Disable reporting of file errors. Equivalent to `set_prolog_flag(fileerrors, off)`

- *The following properties should hold globally:*

  `nofileerrors` **is equivalent to** `set_prolog_flag(fileerrors,off)`. (`basic_props:equiv/2`)

gc/0: PREDICATE

Usage:

Enable garbage collection. Equivalent to `set_prolog_flag(gc, on)`

- *The following properties should hold globally:*

  `gc` **is equivalent to** `set_prolog_flag(gc,on)`. (`basic_props:equiv/2`)

nogc/0: PREDICATE

Usage:

Disable garbage collection. Equivalent to `set_prolog_flag(gc, off)`

– *The following properties should hold globally:*

`nogc` **is equivalent to** `set_prolog_flag(gc,off)`. (`basic_props:equiv/2`)

155.3 Documentation on multifiles

define_flag/3: PREDICATE

`define_flag(Flag,Values,Default)`

New flags can be defined by writing facts of this predicate. `Flag` **is the name of the new flag,** `Values` **defines the posible values for the flag (see below) and** `Default` **defines the predefined value associated with the flag (which should be compatible with** `Values`**).**

Usage: `define_flag(Flag,FlagValues,Default)`

– *The following properties hold upon exit:*

`Flag` **is an atom.** (`basic_props:atm/1`)

Define the valid flag values (`basic_props:flag_values/1`)

The predicate is *multifile***.**

155.4 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`datafacts_rt`.

– *Packages:*

`noprelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `nortchecks`, `define_flag`, `datafacts`.

156 Runtime system information

Author(s): Daniel Cabeza, Manuel Carro, Jose F. Morales.

This module provides internal information about the current running runtime system (engine and enviroment). That information includes the architecture, platform, operating system, location of libraries, and C header files. That information is mainly used in parts of the Ciao dynamic compilation (location of source, generation of gluecode for the foreign interface, etc.).

156.1 Usage and interface

- **Library usage:**
  `:- use_module(engine(system_info)).`
- **Exports:**
  - *Predicates:*
    `get_arch/1`, `get_os/1`, `get_platform/1`, `eng_debug_level/1`, `eng_is_sharedlib/0`, `get_ciao_ext/1`, `get_exec_ext/1`, `get_so_ext/1`, `get_a_ext/1`, `ciao_c_headers_dir/1`.

156.2 Documentation on exports

get_arch/1: PREDICATE

This predicate will describe the computer architecture wich is currently executing the predicate.

Computer architectures are identified by a simple atom. This atom is implementation-defined, and may suffer any change from one Ciao version to another.

For example, Ciao running on an 32-bit Intel-based machine will retrieve:

```
?- get_arch(I).

I = i686 ? ;

no
?-
```

Usage: `get_arch(ArchDescriptor)`

Unifies `ArchDescriptor` with a simple atom which describes the computer architecture currently executing the predicate.

- *The following properties hold upon exit:*
  `ArchDescriptor` is an atom. (`basic_props:atm/1`)

get_os/1: PREDICATE

This predicate will describe the operating system which is running on the machine currently executing the Prolog program.

Operating systems are identified by a simple atom. This atom is implementation-defined, and may suffer changes from one Ciao version to another.

For example, Ciao running on Linux will retrieve:

```
?- get_os(I).

I = 'LINUX' ? ;

no
?-
```

Usage: `get_os(OsDescriptor)`

Unifies `OsDescriptor` with a simple atom which describes the running operating system when predicate was called.

– *The following properties hold upon exit:*

`OsDescriptor` **is an atom.** (`basic_props:atm/1`)

get_platform/1: PREDICATE

Usage: `get_platform(Platform)`

`Platform` **is the atom describing the current operating system and computer architecture.**

– *The following properties hold upon exit:*

`Platform` **is an atom.** (`basic_props:atm/1`)

eng_debug_level/1: PREDICATE

Usage: `eng_debug_level(Debug)`

Unifies `Debug` with the value of `core:debug_level` configuration flag used to build this engine.

– *The following properties hold upon exit:*

`Debug` **is an atom.** (`basic_props:atm/1`)

eng_is_sharedlib/0: PREDICATE

Usage:

This engine is linked as a shared library (instead of as an executable)

get_ciao_ext/1: PREDICATE

Usage: `get_ciao_ext(Ext)`

`Ext` **is the default extension for the executable Ciao programs.**

– *The following properties hold upon exit:*

`Ext` **is an atom.** (`basic_props:atm/1`)

get_exec_ext/1: PREDICATE

Usage: `get_exec_ext(Ext)`

`Ext` **is the extension for executables.**

– *The following properties hold upon exit:*

`Ext` **is an atom.** (`basic_props:atm/1`)

get_so_ext/1: PREDICATE

Usage: `get_so_ext(Ext)`

`Ext` **is the default extension for the shared libraries. For example,** `.dll` **in Windows and** `.so` **in most Unix systems.**

- *Calls should, and exit will be compatible with:*

  `Ext` **is an atom.** (`basic_props:atm/1`)

get_a_ext/1: PREDICATE

Usage: `get_a_ext(Ext)`

`Ext` **is the default extension for the static libraries.**

- *Calls should, and exit will be compatible with:*

  `Ext` **is an atom.** (`basic_props:atm/1`)

ciao_c_headers_dir/1: PREDICATE

Usage: `ciao_c_headers_dir(Path)`

`Path` **is the path to the root of the installed Ciao header C files (.h), typically used for interfacing Ciao and C.**

- *The following properties hold upon exit:*

  `Path` **is an atom.** (`basic_props:atm/1`)

156.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`.

157 Assertion processing library

Author(s): Manuel Hermenegildo.

This module defines some predicates which are useful for writing programs which process assertions (as defined in the `assertions` library). The exported predicates allow reading assertions in all acceptable syntactic forms and converting them to a normalized format.

If you want to have access to some of the declarations read by the predicates exported by this file it is also necessary to include the library `compiler/c_itf`.

157.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assertions/assrt_lib)).`
- **Exports:**
  - *Predicates:*
    `get_code_and_related_assertions/5`, `get_code_and_related_assertions_opts/6`, `cleanup_code_and_related_assertions/0`, `check_code_and_assrt_syntax/1`, `use_pkg/2`, `clause_read/7`, `assertion_read/9`, `assertion_body/7`, `generate_asr_file/2`, `read_asr_file/2`, `rel_data/1`, `print_assertions/1`, `print_unformatted_assertions/1`, `comps_to_goal/3`, `comps_to_goal/4`, `normalize_assertion/9`, `norm_goal_prop/3`, `denorm_goal_prop/3`, `prop_apply/3`, `prop_unapply/3`, `prop_argvar/2`.

157.2 Documentation on exports

get_code_and_related_assertions/5: PREDICATE

Usage: `get_code_and_related_assertions(I,M,Base,Suffix,Dir)`

This is the main entry point to the assertion reader/normalizer. Reads all declarations and code in `I` and leaves it asserted in the database, in the format defined in `'compiler/c_itf'`. Clauses are stored in `clause_read/7`. Used packages are stored in `use_pkg/2`.

Also, it reads and normalizes *all* assertions in this file and all related files, leaving them asserted in `assertion_read/9` facts. If up to date `.asr` files exist for this or any of the related files, the assertion information is read directly from such `.asr` files. Otherwise, the `.pl` file is read and an up to date `.asr` file is generated containing all assertions in the `.pl` file, normalized and stored as `assertion_read/9` facts.

`M` is the name of the module defined by the file. `Base` is the absolute name of the file `I` but with no suffix. `Suffix` is the file suffix (e.g., '`.pl`'). `Dir` is the directory part of the filename (with no / at the end).

Since this predicate is intended for gathering file information for purposes which can be other than compilation to executable code (e.g., generating documentation or in the preprocessor) this predicate catches errors and proceeds in cases where file processing (e.g., during actual compilation) might normally abort.

- *Call and exit should be compatible with:*
  `I` is an atom describing the name of a file. (`c_itf_props:filename/1`)
  `M` is a module descriptor. (`c_itf_props:moddesc/1`)
  `Base` is an atom. (`basic_props:atm/1`)

`Suffix` **is an atom.** (basic_props:atm/1)
`Dir` **is an atom.** (basic_props:atm/1)

– *The following properties should hold at call time:*
`I` **is currently ground (it contains no variables).** (term_typing:ground/1)

– *The following properties should hold upon exit:*
`I` **is currently ground (it contains no variables).** (term_typing:ground/1)
`M` **is currently ground (it contains no variables).** (term_typing:ground/1)
`Base` **is currently ground (it contains no variables).** (term_typing:ground/1)
`Suffix` **is currently ground (it contains no variables).** (term_typing:ground/1)
`Dir` **is currently ground (it contains no variables).** (term_typing:ground/1)

get_code_and_related_assertions_opts/6: PREDICATE

Usage: `get_code_and_related_assertions_opts(I,Opts,M,Base,Suffix,Dir)`

Version which accepts some options in `Opts`**. In particular,** `'-v'` **produces verbose output for debugging. Also passes on the options in** `Opts` **to pass two of the assertion normalizer.**

– *Call and exit should be compatible with:*
`I` **is an atom describing the name of a file.** (c_itf_props:filename/1)
`Opts` **is a list of** `atm`**s.** (basic_props:list/2)
`M` **is a module descriptor.** (c_itf_props:moddesc/1)
`Base` **is an atom.** (basic_props:atm/1)
`Suffix` **is an atom.** (basic_props:atm/1)
`Dir` **is an atom.** (basic_props:atm/1)

– *The following properties should hold at call time:*
`I` **is currently ground (it contains no variables).** (term_typing:ground/1)
`Opts` **is currently ground (it contains no variables).** (term_typing:ground/1)

– *The following properties should hold upon exit:*
`I` **is currently ground (it contains no variables).** (term_typing:ground/1)
`Opts` **is currently ground (it contains no variables).** (term_typing:ground/1)
`M` **is currently ground (it contains no variables).** (term_typing:ground/1)
`Base` **is currently ground (it contains no variables).** (term_typing:ground/1)
`Suffix` **is currently ground (it contains no variables).** (term_typing:ground/1)
`Dir` **is currently ground (it contains no variables).** (term_typing:ground/1)

cleanup_code_and_related_assertions/0: PREDICATE

Usage:

Cleans up data asserted by assertion/code reader/normalizer.

check_code_and_assrt_syntax/1: PREDICATE

Usage: `check_code_and_assrt_syntax(I)`

This predicate is useful for checking the syntax of the code and assertions in a file, as well as imports and exports. Full (semantic) assertion checking must be done with the preprocessor.

- *Call and exit should be compatible with:*
  `I` **is an atom describing the name of a file.** (`c_itf_props:filename/1`)
- *The following properties should hold at call time:*
  `I` **is currently ground (it contains no variables).** (`term_typing:ground/1`)
- *The following properties should hold upon exit:*
  `I` **is currently ground (it contains no variables).** (`term_typing:ground/1`)

use_pkg/2: PREDICATE

Usage: `use_pkg(Base,Pkg)`

After calling `get_assertions_and_code/5` **this predicate contains the packages used in the file. The format is the same as that of** `package/2` **in** `c_itf`

- *The following properties should hold upon exit:*
  `Base` **is an atom.** (`basic_props:atm/1`)
  `Pkg` **is an atom describing the name of a file.** (`c_itf_props:filename/1`)

The predicate is of type *data***.**

clause_read/7: PREDICATE

Usage: `clause_read(Base,Head,Body,VarNames,Source,Line0,Line1)`

After calling `get_assertions_and_code/5` **this predicate contains the clauses in the file. The format is the same as that of** `clause_of/7` **in** `c_itf`

- *The following properties should hold upon exit:*
  `Base` **is an atom.** (`basic_props:atm/1`)
  `Head` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)
  `Source` **is an atom describing the name of a file.** (`c_itf_props:filename/1`)
  `Line0` **is an integer.** (`basic_props:int/1`)
  `Line1` **is an integer.** (`basic_props:int/1`)

The predicate is of type *data***.**

assertion_read/9: PREDICATE

Usage: `assertion_read(Goal,M,Status,Type,Body,Dict,Source,LB,LE)`

Each fact represents that an assertion for `Goal` **has been read in module** `M`**, which has status** `Status` **and is of type** `Type`**.** `Body` **is the actual body of the assertion.** `Dict` **contains the names of the variables which appear in the assertion.** `Source` **is the file in which the assertion appears (treats included files correctly).** `LB` **and** `LE` **are the first and last line numbers in this source file in which the assertion appears (if the source is not available or has not been read** `LB`**=**`LE`**=0).** `Goal` **may be normalized or not, i.e., it may contain modes or properties, but it is always a term of the same functor and arity as the predicate it represents (i.e., it is not in Functor/Arity format).** `Body` **is always normalized (but the properties or property conjunctions inside may not – see** `normalize_assertions_pass_one/1` **and** `normalize_assertions_pass_two/1`**.**

– *The following properties should hold upon exit:*
  M is a module descriptor. (c_itf_props:moddesc/1)
  Status is an acceptable status for an assertion. (assertions_props:assrt_status/1)
  Type is an admissible kind of assertion. (assertions_props:assrt_type/1)
  Body is an assertion body. (assertions_props:assrt_body/1)
  Dict is a dictionary. (dict:dictionary/1)
  LB is an integer. (basic_props:int/1)
  Source is an atom describing the name of a file. (c_itf_props:filename/1)
  LE is an integer. (basic_props:int/1)

The predicate is of type *data*.

assertion_body/7: PREDICATE

`assertion_body(Pred,Compat,Call,Succ,Comp,Comm,Asst)`

– *If the following properties hold at call time:*
  Pred is currently ground (it contains no variables). (term_typing:ground/1)
  Compat is currently ground (it contains no variables). (term_typing:ground/1)
  Call is currently ground (it contains no variables). (term_typing:ground/1)
  Succ is currently ground (it contains no variables). (term_typing:ground/1)
  Comp is currently ground (it contains no variables). (term_typing:ground/1)
  Comm is currently ground (it contains no variables). (term_typing:ground/1)
  then the following properties should hold upon exit:
  Asst is currently ground (it contains no variables). (term_typing:ground/1)

`assertion_body(Pred,Compat,Call,Succ,Comp,Comm,Asst)`

– *If the following properties hold at call time:*
  Asst is currently ground (it contains no variables). (term_typing:ground/1)
  then the following properties should hold upon exit:
  Pred is currently ground (it contains no variables). (term_typing:ground/1)
  Compat is currently ground (it contains no variables). (term_typing:ground/1)
  Call is currently ground (it contains no variables). (term_typing:ground/1)
  Succ is currently ground (it contains no variables). (term_typing:ground/1)
  Comp is currently ground (it contains no variables). (term_typing:ground/1)
  Comm is currently ground (it contains no variables). (term_typing:ground/1)

generate_asr_file/2: PREDICATE

No further documentation available for this predicate.

read_asr_file/2: PREDICATE

Usage:

Reads the data in the .asr file. Fails if version is different from current version (so that .asr will be regenerated).

rel_data/1: PREDICATE

No further documentation available for this predicate. The predicate is of type *data*.

print_assertions/1: PREDICATE

Usage: `print_assertions(M)`

Prints the assertions stored in the database as `assertion_read/9` facts, performing some pretty-printing and simplification (e.g., eliminating empty fields). If `M` is instantiated, only information on module `M` is printed. Otherwise information for all modules is printed.

- *Call and exit should be compatible with:*

  `M` **is a module descriptor.** (`c_itf_props:moddesc/1`)

print_unformatted_assertions/1: PREDICATE

Usage: `print_unformatted_assertions(M)`

Prints the assertions stored in the database as `assertion_read/9` facts, in a raw format (no attempt is made to simplify the assertions). If `M` is instantiated, only information on module `M` is printed. Otherwise information for all modules is printed.

- *Call and exit should be compatible with:*

  `M` **is a module descriptor.** (`c_itf_props:moddesc/1`)

comps_to_goal/3: PREDICATE

Usage 1:

This predicate allows to compound a list of global properties in to sucessive meta-calls

Usage 2:

This predicate allows to compound a list of global properties in to successive meta-calls, but in the third argument you can use your own selector.

comps_to_goal/4: PREDICATE

No further documentation available for this predicate. *Meta-predicate* **with arguments:** `comps_to_goal(?,pred(3),?,?)`.

normalize_assertion/9: PREDICATE

Usage: `normalize_assertion(M,Assrt,PD,AStatus,AType,NABody,S,LB,LE)`

Normalizes one assertion (see `normalize_assertions/3`).

- *Call and exit should be compatible with:*

  `M` **is a module descriptor.** (`c_itf_props:moddesc/1`)

  `Assrt` **is any term.** (`basic_props:term/1`)

  `PD` **is any term.** (`basic_props:term/1`)

  `AStatus` **is an acceptable status for an assertion.** (`assertions_props:assrt_status/1`)

  `AType` **is an admissible kind of assertion.** (`assertions_props:assrt_type/1`)

  `NABody` **is an assertion body.** (`assertions_props:assrt_body/1`)

  `S` **is an atom.** (`basic_props:atm/1`)

  `LB` **is an integer.** (`basic_props:int/1`)

  `LE` **is an integer.** (`basic_props:int/1`)

norm_goal_prop/3: PREDICATE

`norm_goal_prop(Prop,NProp,NPr)`

`Prop` is a term describing a global property in an assertion. `NProp` is its normalized version, where `NPr` is the extra argument. E.g., `norm_goal_prop(regtype,regtype(p(X)),p(X))`.

Usage 1: `norm_goal_prop(Prop,NProp,NPr)`

Normalizes a global property.

- *The following properties should hold at call time:*

  `Prop` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `NProp` is a free variable. (`term_typing:var/1`)

  `NPr` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Usage 2: `norm_goal_prop(Prop,NProp,NPr)`

Denormalizes a global property.

- *The following properties should hold at call time:*

  `Prop` is a free variable. (`term_typing:var/1`)

  `NProp` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `NPr` is a free variable. (`term_typing:var/1`)

denorm_goal_prop/3: PREDICATE

Usage: `denorm_goal_prop(NProp,Prop,NPr)`

Denormalizes a global property.

- *The following properties should hold at call time:*

  `NProp` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

  `Prop` is a free variable. (`term_typing:var/1`)

  `NPr` is a free variable. (`term_typing:var/1`)

prop_apply/3: PREDICATE

No further documentation available for this predicate.

prop_unapply/3: PREDICATE

No further documentation available for this predicate.

prop_argvar/2: PREDICATE

No further documentation available for this predicate.

157.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `datafacts_rt`, `aggregates`, `read`, `terms_io`, `fastrw`, `assrt_write`, `c_itf`, `file_buffer`, `ctrlcclean`, `errhandle`, `lists`, `system`, `translation`, `assertions_props`, `c_itf_props`, `messages`, `terms_vars`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `basicmodes`, `regtypes`, `dcg`, `hiord`, `datafacts`.

158 Dynamic loading and compilation

Author(s): The Ciao Development Team.

This module provides a simplified interface to the Ciao compiler front-end (see `toplevel` **for details).**

158.1 Usage and interface

- **Library usage:**
  `:- use_module(library(compiler)).`
- **Exports:**
  - *Predicates:*
    `make_po/1`, `make_wam/1`, `ensure_loaded/1`, `ensure_loaded/2`, `use_module/1`, `use_module/2`, `use_module/3`, `unload/1`, `set_debug_mode/1`, `set_nodebug_mode/1`, `set_debug_module/1`, `set_nodebug_module/1`, `set_debug_module_source/1`, `mode_of_module/2`, `module_of/2`.

158.2 Documentation on exports

make_po/1: PREDICATE

No further documentation available for this predicate.

make_wam/1: PREDICATE

No further documentation available for this predicate.

ensure_loaded/1: PREDICATE

No further documentation available for this predicate.

ensure_loaded/2: PREDICATE

No further documentation available for this predicate.

use_module/1: PREDICATE

No further documentation available for this predicate.

use_module/2: PREDICATE

No further documentation available for this predicate. *Meta-predicate* **with arguments:** `use_module(?,addmodule(?))`.

use_module/3: PREDICATE

No further documentation available for this predicate.

unload/1: PREDICATE

No further documentation available for this predicate.

set_debug_mode/1: PREDICATE

No further documentation available for this predicate.

set_nodebug_mode/1: PREDICATE

No further documentation available for this predicate.

set_debug_module/1: PREDICATE

No further documentation available for this predicate.

set_nodebug_module/1: PREDICATE

No further documentation available for this predicate.

set_debug_module_source/1: PREDICATE

No further documentation available for this predicate.

mode_of_module/2: PREDICATE

No further documentation available for this predicate.

module_of/2: PREDICATE

No further documentation available for this predicate.

158.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`, `system`, `c_itf`, `exemaker`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `datafacts`.

159 Customizing path aliases

Author(s): Daniel Cabeza.

This library provides means for customizing the path aliases known by an executable from the `CIAOALIASPATH` environment variable. This is used by many applications of Ciao, including `ciaoc`, `ciaosh`, and `ciao-shell`, specially during system boostrapping.

This library is not recommended for user programs. Use bundles instead.

159.1 Usage and interface

- **Library usage:**
  `:- use_module(library(libpaths)).`
- **Exports:**
  - *Predicates:*
    `get_alias_path/0`.
  - *Multifiles:*
    `file_search_path/2`, `library_directory/1`.

159.2 Documentation on exports

get_alias_path/0: PREDICATE

`get_alias_path`

Consult the environment variable `CIAOALIASPATH` and add facts to predicates `library_directory/1` and `file_search_path/2` to define new library paths and path aliases. The format of `CIAOALIASPATH` is a sequence of paths or alias assignments separated by a path list separator character (see `extract_paths/2`), an alias assignment is the name of the alias, an = and the path represented by that alias (no blanks allowed). For example, given

```
CIAOALIASPATH=/home/bardo/ciao:contrib=/usr/local/lib/ciao
```

the predicate will define `/home/bardo/ciao` as a library path and `/usr/local/lib/ciao` as the path represented by `contrib`.

159.3 Documentation on multifiles

file_search_path/2: PREDICATE

See Chapter 144 [Basic file/stream handling], page 619. The predicate is *multifile*.

The predicate is of type *dynamic*.

`file_search_path(X,Y)`

- *The following properties hold upon exit:*
  `X` **is ground.** (`basic_props:gnd/1`)
  `Y` **is ground.** (`basic_props:gnd/1`)

library_directory/1: PREDICATE

See Chapter 144 [Basic file/stream handling], page 619. The predicate is *multifile***.**

The predicate is of type *dynamic***.**

`library_directory(X)`

- *The following properties hold upon exit:*

  `X` **is ground.** (`basic_props:gnd/1`)

159.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `datafacts_rt`, `system`, `lists`.

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `datafacts`.

160 Operating system utilities

Author(s): Daniel Cabeza, Manuel Carro.

This module contains predicates for invoking services which are typically provided by the operating system. Note that the predicates which take names of files or directories as arguments in this module expect atoms, not path aliases. I.e., generally these predicates will not call `absolute_file_name/2` **on names of files or directories taken as arguments.**

160.1 Usage and interface

- **Library usage:**
  `:- use_module(library(system)).`
- **Exports:**
  - *Predicates:*
    `now/1`, `datime/1`, `datime/9`, `copy_file/2`, `copy_file/3`, `getenvstr/2`, `setenvstr/2`, `set_env/2`, `del_env/1`, `current_env/2`, `extract_paths/2`, `current_host/1`, `current_executable/1`, `umask/2`, `working_directory/2`, `cd/1`, `directory_files/2`, `mktemp/2`, `mktemp_in_tmp/2`, `file_exists/1`, `file_exists/2`, `file_property/2`, `file_properties/6`, `modif_time/2`, `modif_time0/2`, `touch/1`, `fmode/2`, `chmod/2`, `chmod/3`, `set_exec_mode/2`, `delete_directory/1`, `delete_file/1`, `rename_file/2`, `make_directory/2`, `make_directory/1`, `system_error_report/1`, `get_tmp_dir/1`, `dev_null/1`, `pause/1`, `wait/2`, `kill/2`, `get_pid/1`, `get_uid/1`, `get_gid/1`, `get_pwnam/1`, `get_grnam/1`, `get_numcores/1`, `shell/0`, `shell/1`, `shell/2`, `system/1`, `system/2`, `fd_dup/2`, `fd_close/1`, `winpath/3`, `winpath/2`, `winpath_c/3`, `cyg2win/3`, `no_swapslash/3`, `cyg2win_a/3`, `using_windows/0`, `get_home/1`, `find_executable/2`.
  - *Regular Types:*
    `datime_struct/1`, `copy_option/1`, `copy_options/1`.

160.2 Documentation on exports

now/1: PREDICATE

`now(Time)`

`Time` **is unified with the number of seconds elapsed since January, 1, 1970 (UTC).**

Usage:
- *Calls should, and exit will be compatible with:*
  `Time` **is an integer.** (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Time` **is an integer.** (`basic_props:int/1`)

datime/1: PREDICATE

`datime(Datime)`

`Datime` **is unified with a term of the form** `datime(Year,Month,Day,Hour,Minute,Second)` **which contains the current date and time.**

Usage:

- *Call and exit should be compatible with:*
  system:datime_struct(Datime) (system:datime_struct/1)
- *The following properties should hold upon exit:*
  system:datime_struct(Datime) (system:datime_struct/1)

datime_struct/1: REGTYPE

A regular type, defined as follows:

```
datime_struct(datime(Year,Month,Day,Hour,Min,Sec)) :-
    int(Year),
    int(Month),
    int(Day),
    int(Hour),
    int(Min),
    int(Sec).
```

datime/9: PREDICATE

datime(Time,Year,Month,Day,Hour,Min,Sec,WeekDay,YearDay)

Time is as in now/1. WeekDay is the number of days since Sunday, in the range 0 to 6. YearDay is the number of days since January 1, in the range 0 to 365.

Usage 1:

If Time is given, the rest of the arguments are unified with the date and time to which the Time argument refers.

- *Calls should, and exit will be compatible with:*
  Year **is an integer.** (basic_props:int/1)
  Month **is an integer.** (basic_props:int/1)
  Day **is an integer.** (basic_props:int/1)
  Hour **is an integer.** (basic_props:int/1)
  Min **is an integer.** (basic_props:int/1)
  Sec **is an integer.** (basic_props:int/1)
  WeekDay **is an integer.** (basic_props:int/1)
  YearDay **is an integer.** (basic_props:int/1)
- *The following properties should hold at call time:*
  Time **is an integer.** (basic_props:int/1)
- *The following properties hold upon exit:*
  Year **is an integer.** (basic_props:int/1)
  Month **is an integer.** (basic_props:int/1)
  Day **is an integer.** (basic_props:int/1)
  Hour **is an integer.** (basic_props:int/1)
  Min **is an integer.** (basic_props:int/1)
  Sec **is an integer.** (basic_props:int/1)
  WeekDay **is an integer.** (basic_props:int/1)
  YearDay **is an integer.** (basic_props:int/1)

Usage 2:

Bound Time, WeekDay and YearDay as determined by the input arguments.

- *Calls should, and exit will be compatible with:*
  `Time` **is an integer.** (basic_props:int/1)
  `WeekDay` **is an integer.** (basic_props:int/1)
  `YearDay` **is an integer.** (basic_props:int/1)
- *The following properties should hold at call time:*
  `Year` **is an integer.** (basic_props:int/1)
  `Month` **is an integer.** (basic_props:int/1)
  `Day` **is an integer.** (basic_props:int/1)
  `Hour` **is an integer.** (basic_props:int/1)
  `Min` **is an integer.** (basic_props:int/1)
  `Sec` **is an integer.** (basic_props:int/1)
- *The following properties hold upon exit:*
  `Time` **is an integer.** (basic_props:int/1)
  `WeekDay` **is an integer.** (basic_props:int/1)
  `YearDay` **is an integer.** (basic_props:int/1)

Usage 3:

Bound `Time` to current time and the rest of the arguments refer to current time.

- *Calls should, and exit will be compatible with:*
  `WeekDay` **is an integer.** (basic_props:int/1)
  `YearDay` **is an integer.** (basic_props:int/1)
- *The following properties should hold at call time:*
  `Time` **is a free variable.** (term_typing:var/1)
  `Year` **is a free variable.** (term_typing:var/1)
  `Month` **is a free variable.** (term_typing:var/1)
  `Day` **is a free variable.** (term_typing:var/1)
  `Hour` **is a free variable.** (term_typing:var/1)
  `Min` **is a free variable.** (term_typing:var/1)
  `Sec` **is a free variable.** (term_typing:var/1)
- *The following properties hold upon exit:*
  `Time` **is an integer.** (basic_props:int/1)
  `Year` **is an integer.** (basic_props:int/1)
  `Month` **is an integer.** (basic_props:int/1)
  `Day` **is an integer.** (basic_props:int/1)
  `Hour` **is an integer.** (basic_props:int/1)
  `Min` **is an integer.** (basic_props:int/1)
  `Sec` **is an integer.** (basic_props:int/1)
  `WeekDay` **is an integer.** (basic_props:int/1)
  `YearDay` **is an integer.** (basic_props:int/1)

copy_option/1: REGTYPE

Usage: `copy_option(Opt)`

`Opt` **is a file copy option.**

```
copy_option(overwrite).
copy_option(timestamp).
copy_option(symlink).
copy_option(append).
```

copy_options/1: REGTYPE

Usage: `copy_options(Opts)`

`Opts` is a list of file copy options.

copy_file/2: PREDICATE

`copy_file(Source,Destination)`

Copies the file `Source` to `Destination`.

Usage:

- *The following properties should hold at call time:*
  `Source` is an atom. (`basic_props:atm/1`)
  `Destination` is an atom. (`basic_props:atm/1`)

copy_file/3: PREDICATE

Usage:

- *The following properties should hold at call time:*
  `Arg1` is an atom. (`basic_props:atm/1`)
  `Arg2` is an atom. (`basic_props:atm/1`)
  `Arg3` is a list of file copy options. (`system:copy_options/1`)

getenvstr/2: PREDICATE

`getenvstr(Name,Value)`

The environment variable `Name` has `Value`. Fails if variable `Name` is not defined.

Usage:

- *Call and exit should be compatible with:*
  `Value` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Name` is an atom. (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Value` is a string (a list of character codes). (`basic_props:string/1`)

setenvstr/2: PREDICATE

`setenvstr(Name,Value)`

The environment variable `Name` is assigned `Value`.

Usage:

- *The following properties should hold at call time:*
  `Name` is an atom. (`basic_props:atm/1`)
  `Value` is a string (a list of character codes). (`basic_props:string/1`)

set_env/2: PREDICATE

`set_env(Name,Value)`

The environment variable `Name` is assigned `Value`.

Usage:

– *The following properties should hold at call time:*

`Name` **is an atom.** (`basic_props:atm/1`)

`Value` **is an atom.** (`basic_props:atm/1`)

del_env/1: PREDICATE

`del_env(Name)`

The environment variable `Name` is removed.

Usage:

– *The following properties should hold at call time:*

`Name` **is an atom.** (`basic_props:atm/1`)

current_env/2: PREDICATE

`current_env(Name,Value)`

If `Name` is an atom, then unifies the environment variable `Name` with its value. Note that this predicate can be used to enumerate all the environment variables using backtracking.

Usage:

– *Call and exit should be compatible with:*

`Name` **is an atom.** (`basic_props:atm/1`)

`Value` **is an atom.** (`basic_props:atm/1`)

– *The following properties should hold upon exit:*

`Name` **is an atom.** (`basic_props:atm/1`)

`Value` **is an atom.** (`basic_props:atm/1`)

extract_paths/2: PREDICATE

`extract_paths(PathList,Paths)`

Split `PathList` atom into the list of paths `Paths`. Paths in `String` are separated by the path list separator character (colons in POSIX-like systems, semicolons in Windows). Empty paths are removed from `Paths`. `Paths` is empty if `PathList` is the empty atom.

Usage:

– *Calls should, and exit will be compatible with:*

`Paths` **is a list of `atm`s.** (`basic_props:list/2`)

– *The following properties should hold at call time:*

`PathList` **is an atom.** (`basic_props:atm/1`)

– *The following properties hold upon exit:*

`Paths` **is a list of `atm`s.** (`basic_props:list/2`)

current_host/1: PREDICATE

`current_host(Hostname)`

`Hostname` is unified with the fully qualified name of the host.

Usage:

- *Calls should, and exit will be compatible with:*
  `Hostname` is an atom. (`basic_props:atm/1`)
- *The following properties hold upon exit:*
  `Hostname` is an atom. (`basic_props:atm/1`)

current_executable/1: PREDICATE

`current_executable(Path)`

Unifies `Path` with the path to the current Ciao executable (which may be a standalone binary or bytecode executable)

Usage:

- *Calls should, and exit will be compatible with:*
  `Path` is an atom. (`basic_props:atm/1`)
- *The following properties hold upon exit:*
  `Path` is an atom. (`basic_props:atm/1`)

umask/2: PREDICATE

Usage 1: `umask(OldMask,NewMask)`

The process file creation mask was `OldMask`, and it is changed to `NewMask`.

- *The following properties should hold at call time:*
  `NewMask` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `OldMask` is an integer. (`basic_props:int/1`)

Usage 2: `umask(OldMask,NewMask)`

Gets the process file creation mask without changing it.

- *The following properties should hold at call time:*
  `OldMask` is a free variable. (`term_typing:var/1`)
  `NewMask` is a free variable. (`term_typing:var/1`)
  The terms `OldMask` and `NewMask` are strictly identical. (`term_compare:== /2`)
- *The following properties hold upon exit:*
  `OldMask` is an integer. (`basic_props:int/1`)
  `NewMask` is an integer. (`basic_props:int/1`)

working_directory/2: PREDICATE

`working_directory(OldDir,NewDir)`

Unifies current working directory with `OldDir`, and then changes the working directory to `NewDir`. Calling `working_directory(Dir,Dir)` simply unifies `Dir` with the current working directory without changing anything else.

Usage 1:

Changes current working directory.

- *Calls should, and exit will be compatible with:*
  `OldDir` **is an atom.** (basic_props:atm/1)
- *The following properties should hold at call time:*
  `NewDir` **is an atom.** (basic_props:atm/1)
- *The following properties hold upon exit:*
  `OldDir` **is an atom.** (basic_props:atm/1)

Usage 2: `working_directory(OldDir,NewDir)`

Gets current working directory.

- *The following properties should hold at call time:*
  `OldDir` **is a free variable.** (term_typing:var/1)
  `NewDir` **is a free variable.** (term_typing:var/1)
  The terms `OldDir` **and** `NewDir` **are strictly identical.** (term_compare:== /2)
- *The following properties hold upon exit:*
  `OldDir` **is an atom.** (basic_props:atm/1)
  `NewDir` **is an atom.** (basic_props:atm/1)

cd/1: PREDICATE

`cd(Path)`

Changes working directory to `Path`**.**

Usage:

- *The following properties should hold at call time:*
  `Path` **is an atom.** (basic_props:atm/1)

directory_files/2: PREDICATE

`directory_files(Directory,FileList)`

`FileList` **is the unordered list of entries (files, directories, etc.) in** `Directory`**.**

Usage:

- *Calls should, and exit will be compatible with:*
  `FileList` **is a list of** `atm`**s.** (basic_props:list/2)
- *The following properties should hold at call time:*
  `Directory` **is an atom.** (basic_props:atm/1)
- *The following properties hold upon exit:*
  `FileList` **is a list of** `atm`**s.** (basic_props:list/2)

mktemp/2: PREDICATE

`mktemp(Template,Filename)`

Returns a unique `Filename` **based on** `Template`**:** `Template` **must be a valid file name with six trailing X, which are substituted to create a new file name.** `Filename` **is created in read/write mode but closed immediately after creation.**

Usage:

- *Calls should, and exit will be compatible with:*
  `Filename` **is an atom.** (basic_props:atm/1)

- *The following properties should hold at call time:*
  `Template` is an atom. (basic_props:atm/1)
- *The following properties hold upon exit:*
  `Filename` is an atom. (basic_props:atm/1)

mktemp_in_tmp/2: PREDICATE

No further documentation available for this predicate.

file_exists/1: PREDICATE

`file_exists(File)`

Succeeds if `File` (a file or directory) exists (and is accessible).

Usage:

- *The following properties should hold at call time:*
  `File` is an atom. (basic_props:atm/1)

file_exists/2: PREDICATE

`file_exists(File,Mode)`

`File` (a file or directory) exists and it is accessible with `Mode`, as in the Unix call `access(2)`. Typically, `Mode` is 4 for read permission, 2 for write permission and 1 for execute permission.

Usage:

- *The following properties should hold at call time:*
  `File` is an atom. (basic_props:atm/1)
  `Mode` is an integer. (basic_props:int/1)
- *The following properties hold upon exit:*
  `File` is an atom. (basic_props:atm/1)
  `Mode` is an integer. (basic_props:int/1)

file_property/2: PREDICATE

`file_property(File,Property)`

`File` has the property `Property`. The possible properties are:

type(`Type`)
: `Type` is one of `regular`, `directory`, `fifo`, `socket` or `unknown`.

linkto(`Linkto`)
: If `File` is a symbolic link, `Linkto` is the file pointed to by the link (and the other properties come from that file, not from the link itself).

mod_time(`ModTime`)
: `ModTime` is the time of last modification (seconds since January, 1, 1970).

mode(`Protection`)
: `Protection` is the protection mode.

size(`Size`)
: `Size` is the size.

If `Property` is uninstantiated, the predicate will enumerate the properties on backtracking.

Usage:

- *Call and exit should be compatible with:*
  `Property` **is a compound term.** (basic_props:struct/1)
- *The following properties should hold at call time:*
  `File` **is an atom.** (basic_props:atm/1)
- *The following properties should hold upon exit:*
  `Property` **is a compound term.** (basic_props:struct/1)

file_properties/6: PREDICATE

`file_properties(Path,Type,Linkto,Time,Protection,Size)`

The file `Path` has the following properties:

- **File type `Type` (one of `regular`, `directory`, `fifo`, `socket` or `unknown`).**
- **If `Path` is a symbolic link, `Linkto` is the file pointed to. All other properties come from the file pointed, not the link. `Linkto` is '' if `Path` is not a symbolic link.**
- **Time of last modification `Time` (seconds since January, 1, 1970).**
- **Protection mode `Protection`.**
- **Size in bytes `Size`.**

Usage:

- *Calls should, and exit will be compatible with:*
  `Type` **is an atom.** (basic_props:atm/1)
  `Linkto` **is an atom.** (basic_props:atm/1)
  `Time` **is an integer.** (basic_props:int/1)
  `Protection` **is an integer.** (basic_props:int/1)
  `Size` **is an integer.** (basic_props:int/1)
- *The following properties should hold at call time:*
  `Path` **is an atom.** (basic_props:atm/1)
- *The following properties hold upon exit:*
  `Type` **is an atom.** (basic_props:atm/1)
  `Linkto` **is an atom.** (basic_props:atm/1)
  `Time` **is an integer.** (basic_props:int/1)
  `Protection` **is an integer.** (basic_props:int/1)
  `Size` **is an integer.** (basic_props:int/1)

modif_time/2: PREDICATE

`modif_time(File,Time)`

The file `File` was last modified at `Time`, which is in seconds since January, 1, 1970. Fails if `File` does not exist.

Usage:

- *Call and exit should be compatible with:*
  `Time` **is an integer.** (basic_props:int/1)

- *The following properties should hold at call time:*
  `File` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Time` **is an integer.** (`basic_props:int/1`)

modif_time0/2: PREDICATE

`modif_time0(File,Time)`

If `File` **exists,** `Time` **is its latest modification time, as in** `modif_time/2`**. Otherwise, if** `File` **does not exist,** `Time` **is zero.**

Usage:

- *Call and exit should be compatible with:*
  `Time` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `File` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Time` **is an integer.** (`basic_props:int/1`)

touch/1: PREDICATE

`touch(File)`

Change the modification time of `File` **to the current time of day. If the file does not exist, it is created with default permissions.**

Note: This operation cannot be fully implemented with `modif_time/2`**. In POSIX systems, that can be done as long as the user has write permissions on the file, even if the owner is different. Change of modification time to arbitrary time values is not allowed in this case.**

Usage:

- *The following properties should hold at call time:*
  `File` **is an atom.** (`basic_props:atm/1`)

fmode/2: PREDICATE

`fmode(File,Mode)`

The file `File` **has protection mode** `Mode`**.**

Usage:

- *Call and exit should be compatible with:*
  `Mode` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `File` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Mode` **is an integer.** (`basic_props:int/1`)

chmod/2: PREDICATE

`chmod(File,NewMode)`

Change the protection mode of file `File` to `NewMode`.

Usage:

- *The following properties should hold at call time:*
  `File` **is an atom.** (`basic_props:atm/1`)
  `NewMode` **is an integer.** (`basic_props:int/1`)

chmod/3: PREDICATE

`chmod(File,OldMode,NewMode)`

The file `File` has protection mode `OldMode` and it is changed to `NewMode`.

Usage 1:

- *Call and exit should be compatible with:*
  `OldMode` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `File` **is an atom.** (`basic_props:atm/1`)
  `NewMode` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold upon exit:*
  `OldMode` **is an integer.** (`basic_props:int/1`)

Usage 2: `chmod(File,OldMode,NewMode)`

If `OldMode` is identical to `NewMode` then it is equivalent to fmode(`File`,`OldMode`)

- *The following properties should hold at call time:*
  `File` **is an atom.** (`basic_props:atm/1`)
  `OldMode` **is a free variable.** (`term_typing:var/1`)
  `NewMode` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `File` **is an atom.** (`basic_props:atm/1`)
  `OldMode` **is an atom.** (`basic_props:atm/1`)
  `NewMode` **is an atom.** (`basic_props:atm/1`)

set_exec_mode/2: PREDICATE

`set_exec_mode(SourceName,ExecName)`

Copies the permissions of `SourceName` to `ExecName` adding permissions to execute.

Usage:

- *The following properties should hold at call time:*
  `SourceName` **is an atom.** (`basic_props:atm/1`)
  `ExecName` **is an atom.** (`basic_props:atm/1`)

delete_directory/1: PREDICATE

`delete_directory(File)`

Delete the directory `Directory`.

Usage:

– *The following properties should hold at call time:*
`File` **is an atom.** (basic_props:atm/1)

delete_file/1: PREDICATE

`delete_file(File)`

Delete the file `File`.

Usage:

– *The following properties should hold at call time:*
`File` **is an atom.** (basic_props:atm/1)

rename_file/2: PREDICATE

`rename_file(File1,File2)`

Change the name of `File1` to `File2`.

Usage:

– *The following properties should hold at call time:*
`File1` **is an atom.** (basic_props:atm/1)
`File2` **is an atom.** (basic_props:atm/1)

make_directory/2: PREDICATE

`make_directory(DirName,Mode)`

Creates the directory `DirName` with a given `Mode`. This is, as usual, operated against the current umask value.

Usage:

– *The following properties should hold at call time:*
`DirName` **is an atom.** (basic_props:atm/1)
`Mode` **is an integer.** (basic_props:int/1)

make_directory/1: PREDICATE

`make_directory(DirName)`

Equivalent to `make_directory(D,0o777)`.

Usage:

– *The following properties should hold at call time:*
`DirName` **is an atom.** (basic_props:atm/1)

system_error_report/1: PREDICATE

Usage: `system_error_report(Report)`

Report is the error message from the last system call, like `strerror` in POSIX.

– *The following properties should hold at call time:*
`Report` **is a free variable.** (term_typing:var/1)

– *The following properties should hold upon exit:*
`Report` **is an atom.** (basic_props:atm/1)

get_tmp_dir/1: PREDICATE

Usage: `get_tmp_dir(TmpDir)`

`TmpDir` is the (normalized) temporary directory for scratch space. On POSIX systems it is computed from normalizing the path specified in the `TMPDIR` environment variable. On Windows it is determined by the `TMP` or `TEMP` environment variable. If none is defined, this predicate tries to guess it from some usual locations (`/tmp`, `/var/tmp`, `/usr/tmp` on POSIX, `c:\temp`, `c:\tmp`, `\temp`, `\tmp` on Windows) or use `.` as last resort.

- *The following properties should hold at call time:*
  `TmpDir` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `TmpDir` is an atom. (`basic_props:atm/1`)

dev_null/1: PREDICATE

Usage: `dev_null(Path)`

File path for the null device (`/dev/null` file in POSIX, `nul` in Windows)

- *The following properties should hold upon exit:*
  `Path` is an atom. (`basic_props:atm/1`)

pause/1: PREDICATE

`pause(Seconds)`

Make this thread sleep for some `Seconds`.

Usage:

- *The following properties should hold at call time:*
  `Seconds` is an integer. (`basic_props:int/1`)

wait/2: PREDICATE

Usage: `wait(Pid,ReturnCode)`

`wait/2` waits for the process numbered `Pid`. Fails if the process does not terminate normally or in case of error (see C `waitpid()` for details). `RetCode` is the process return code.

- *The following properties should hold at call time:*
  `Pid` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `ReturnCode` is a free variable. (`term_typing:var/1`)
  `Pid` is an integer. (`basic_props:int/1`)
  `ReturnCode` is a free variable. (`term_typing:var/1`)
- *The following properties hold upon exit:*
  `ReturnCode` is an integer. (`basic_props:int/1`)

kill/2: PREDICATE

Usage: `kill(Pid,Signal)`

`kill/2` sends the signal `Signal` to the process or process group specified by `Pid`. See Unix man page for a detailed description of signals.

- *The following properties should hold at call time:*
  `Pid` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Signal` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Pid` is an integer. (`basic_props:int/1`)
  `Signal` is an integer. (`basic_props:int/1`)

get_pid/1: PREDICATE

`get_pid(Pid)`

Unifies `Pid` with the process identificator of the current process or thread.

Usage:

- *Calls should, and exit will be compatible with:*
  `Pid` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Pid` is an integer. (`basic_props:int/1`)

get_uid/1: PREDICATE

`get_uid(Uid)`

Unifies `Uid` with the user id of the current process.

Usage:

- *Calls should, and exit will be compatible with:*
  `Uid` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Uid` is an integer. (`basic_props:int/1`)

get_gid/1: PREDICATE

`get_gid(Uid)`

Unifies `Gid` with the group id of the current process.

Usage:

- *Calls should, and exit will be compatible with:*
  `Uid` is an integer. (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `Uid` is an integer. (`basic_props:int/1`)

get_pwnam/1: PREDICATE

`get_pwnam(User)`

Unifies `User` with the user of the current process, as specified in the /etc/passwd file.

Usage:

- *Calls should, and exit will be compatible with:*
  `User` is an atom. (`basic_props:atm/1`)
- *The following properties hold upon exit:*
  `User` is an atom. (`basic_props:atm/1`)

get_grnam/1: PREDICATE

`get_grnam(Group)`

Unifies `Group` with the group of the current process, as specified in the /etc/group file.

Usage:

- *Calls should, and exit will be compatible with:*
  `Group` **is an atom.** (`basic_props:atm/1`)
- *The following properties hold upon exit:*
  `Group` **is an atom.** (`basic_props:atm/1`)

get_numcores/1: PREDICATE

`get_numcores(N)`

Unifies `N` with the number of CPU cores.

Usage:

- *Calls should, and exit will be compatible with:*
  `N` **is an integer.** (`basic_props:int/1`)
- *The following properties hold upon exit:*
  `N` **is an integer.** (`basic_props:int/1`)

shell/0: PREDICATE

Usage:

Executes the OS-specific system shell. When the shell process terminates, control is returned to Prolog. See `shell/2` for details.

shell/1: PREDICATE

`shell(Command)`

`Command` **is executed in the OS-specific system shell. It succeeds if the exit code is zero and fails otherwise. See `shell/2` for details.**

Usage:

- *The following properties should hold at call time:*
  `Command` **is an atom.** (`basic_props:atm/1`)

shell/2: PREDICATE

`shell(Command,RetCode)`

Executes `Command` using the OS-specific system shell and stores the exit code in `RetCode`.

On POSIX-like systems the system shell is specified by the `SHELL` environment variable (`$SHELL -c "command"` for passing user commands). On Windows (native builds, MinGW) it is specified by the `COMSPEC` environment variable (`%COMSPEC% /s /c "command"` for passing user commands).

Note that the use of `shell/2` predicates is discouraged for portability and security reasons. Please consider `process` for a more robust way to launch external processes.

Usage:

- *Calls should, and exit will be compatible with:*
  `RetCode` **is an integer.** (basic_props:int/1)
- *The following properties should hold at call time:*
  `Command` **is an atom.** (basic_props:atm/1)
- *The following properties hold upon exit:*
  `RetCode` **is an integer.** (basic_props:int/1)

system/1: PREDICATE

`system(Command)`

Like `shell/1` but ignores exit code.

Usage:

- *The following properties should hold at call time:*
  `Command` **is an atom.** (basic_props:atm/1)

system/2: PREDICATE

`system(Command,RetCode)`

Synonym for `shell/2`.

Usage:

- *Calls should, and exit will be compatible with:*
  `RetCode` **is an integer.** (basic_props:int/1)
- *The following properties should hold at call time:*
  `Command` **is an atom.** (basic_props:atm/1)
- *The following properties hold upon exit:*
  `RetCode` **is an integer.** (basic_props:int/1)

fd_dup/2: PREDICATE

`fd_dup(FD,NewFD)`

Duplicate the file descriptor `FD` into `NewFD`

Usage 1:

Duplicate a file description into a new free file descriptor (C `dup()`)

- *The following properties should hold at call time:*
  `FD` **is an integer.** (basic_props:int/1)
  `NewFD` **is a free variable.** (term_typing:var/1)
- *The following properties should hold upon exit:*
  `NewFD` **is an integer.** (basic_props:int/1)

Usage 2:

Duplicate a file description into a specified file descriptor (C `dup2()`)

- *The following properties should hold at call time:*
  `FD` **is an integer.** (basic_props:int/1)
  `NewFD` **is an integer.** (basic_props:int/1)

fd_close/1: PREDICATE

Usage:

Close the file descriptor (using C close())

- *The following properties should hold at call time:*
  `Arg1` **is an integer.** (`basic_props:int/1`)

winpath/3: PREDICATE

`winpath(Option,Posix,WinPath)`

`Option` **specifies if you want to get a relative or a full path.** `Posix` **represent a path as usual in unix, and** `WinPath` **is the Windows-Style representation of** `Posix`**.**

Usage 1:

- *The following properties should hold at call time:*
  `Option` **is a free variable.** (`term_typing:var/1`)
  `Posix` **is a free variable.** (`term_typing:var/1`)
  `WinPath` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `system:winpath_option(Option)` (`system:winpath_option/1`)
  `Posix` **is an atom.** (`basic_props:atm/1`)

Usage 2:

- *The following properties should hold at call time:*
  `Option` **is a free variable.** (`term_typing:var/1`)
  `Posix` **is an atom.** (`basic_props:atm/1`)
  `WinPath` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `system:winpath_option(Option)` (`system:winpath_option/1`)
  `WinPath` **is an atom.** (`basic_props:atm/1`)

winpath/2: PREDICATE

Usage 1: `winpath(A,B)`

- *The following properties should hold at call time:*
  `A` **is an atom.** (`basic_props:atm/1`)
  `B` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `A` **is an atom.** (`basic_props:atm/1`)
  `B` **is an atom.** (`basic_props:atm/1`)

Usage 2: `winpath(A,B)`

- *The following properties should hold at call time:*
  `A` **is a free variable.** (`term_typing:var/1`)
  `B` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `A` **is an atom.** (`basic_props:atm/1`)
  `B` **is an atom.** (`basic_props:atm/1`)

Usage 3: `winpath(A,B)`

- *The following properties should hold at call time:*
  `A` is an atom. (`basic_props:atm/1`)
  `B` is an atom. (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `A` is an atom. (`basic_props:atm/1`)
  `B` is an atom. (`basic_props:atm/1`)

winpath_c/3: PREDICATE

Same as winpath/3, but for strings.

cyg2win/3: PREDICATE

Usage: `cyg2win(CygWinPath,WindowsPath,SwapSlash)`

Converts a posix path to a Windows-style path. If `SwapSlash` is `swap`, slashes are converted in to backslash. If it is `noswap`, they are preserved.

- *The following properties should hold at call time:*
  `CygWinPath` is a string (a list of character codes). (`basic_props:string/1`)
  `WindowsPath` is a free variable. (`term_typing:var/1`)
  `SwapSlash` is currently instantiated to an atom. (`term_typing:atom/1`)
- *The following properties should hold upon exit:*
  `CygWinPath` is a string (a list of character codes). (`basic_props:string/1`)
  `WindowsPath` is a string (a list of character codes). (`basic_props:string/1`)
  `SwapSlash` is currently instantiated to an atom. (`term_typing:atom/1`)

no_swapslash/3: PREDICATE

No further documentation available for this predicate.

cyg2win_a/3: PREDICATE

Same as cyg2win/3, but for atoms.

using_windows/0: PREDICATE

Usage:

Using the Windows native API (not POSIX)

get_home/1: PREDICATE

Usage: `get_home(H)`

`H` is the home directory (`HOME` environment variable in POSIX systems and APPDATA in Windows)

- *The following properties should hold at call time:*
  `H` is a free variable. (`term_typing:var/1`)

find_executable/2: PREDICATE

Usage: `find_executable(Name,Path)`

`Path` is the absolute path of the command `Name`, reachable from the `PATH` (environment variable) directories if `Name` is not an absolute path. The suffix for executable files is optionally added to `Path` depending on the current operating system.

– *The following properties should hold at call time:*

`Name` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

`Path` **is a free variable.** (`term_typing:var/1`)

160.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`lists`, `pathnames`.

– *Packages:*

`noprelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `isomodes`, `regtypes`.

161 File path names

Author(s): Jose F. Morales.

A pathname is an symbolic identifier that locates a file in a filesystem (e.g., `foo/bar/baz.txt`). This module provides predicates to manipulate pathnames, encoded as atoms. No file system access is required for pathname manipulation.

161.1 Usage and interface

- **Library usage:**
  `:- use_module(library(pathnames)).`
- **Exports:**
  - *Predicates:*
    `path_is_absolute/1`, `path_is_relative/1`, `path_is_basename/1`, `path_is_root/1`, `path_concat/3`, `path_split/3`, `path_norm/2`, `path_splitext/3`, `path_basename/2`, `path_dirname/2`, `path_relocate/4`, `path_get_relative/3`, `path_split_list/2`, `path_concat_list/2`.
  - *Regular Types:*
    `pathname/1`.

161.2 Documentation on exports

pathname/1: REGTYPE

Usage: `pathname(X)`

`X` **is a pathname (encoded as an atom)**

path_is_absolute/1: PREDICATE

Usage: `path_is_absolute(Path)`

`Path` **is an absolute pathname**

- *Call and exit should be compatible with:*
  `Path` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)
- *The following properties should hold at call time:*
  `Path` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

path_is_relative/1: PREDICATE

Usage: `path_is_relative(Path)`

`Path` **is a relative pathname**

- *Call and exit should be compatible with:*
  `Path` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)
- *The following properties should hold at call time:*
  `Path` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

path_is_basename/1: PREDICATE

Usage: `path_is_basename(Path)`

`Path` is a basename (empty directory part) (equivalent to `path_split(Path, '', _)`

- *Call and exit should be compatible with:*
  `Path` is a pathname (encoded as an atom) (`pathnames:pathname/1`)
- *The following properties should hold at call time:*
  `Path` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

path_is_root/1: PREDICATE

Usage: `path_is_root(Path)`

`Path` is a root directory (normalized into `/` or `//`) (equivalent to `+ Path = ''`, `path_split(Path, Path, '')`

- *Call and exit should be compatible with:*
  `Path` is a pathname (encoded as an atom) (`pathnames:pathname/1`)
- *The following properties should hold at call time:*
  `Path` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

path_concat/3: PREDICATE

Usage: `path_concat(PathA,PathB,Path)`

Concatenate pathnames `PathA` and `PathB` in a new path `Path`, adding a `/` separator if needed. If `PathB` is `''`, then `Path` is a `/` ended path. If `PathB` is an absolute pathname, `Path` is `PathB`. No pathname normalization is performed in any case.

- *Call and exit should be compatible with:*
  `PathA` is a pathname (encoded as an atom) (`pathnames:pathname/1`)
  `PathB` is a pathname (encoded as an atom) (`pathnames:pathname/1`)
  `Path` is a pathname (encoded as an atom) (`pathnames:pathname/1`)
- *The following properties should hold at call time:*
  `PathA` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `PathB` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

path_split/3: PREDICATE

`path_split(Path,Dir,Base)`

Given a pathname `Path`, `Base` is the last component of the path and `Dir` is the rest of the path. The following rules must hold:

- **All trailing slashes from `Dir` are removed (except if it is a root directory containing one or more slashes, e.g., `'/'`, `'//'`).**
- **If `Path` ends in a slash (`'.../'`, `Base` is empty (`''`).**
- **If `Path` is empty, both `Dir` and `Base` are empty.**
- **Concatenating `Dir` and `Base` results in pathname that is equivalent (modulo normalization) to `Path`. That is, for all `A`, `B`, `C`:**

```
path_split(A,B,C),
path_concat(B,C,D),
path_norm(A,An), path_norm(D,Dn), An = Dn.
```

Usage: `path_split(Path,Dir,Base)`

Split `Path` into the directory part `Dir` and the basename part `Base`.

- *Call and exit should be compatible with:*

`Path` **is a pathname (encoded as an atom)**	(`pathnames:pathname/1`)
`Dir` **is a pathname (encoded as an atom)**	(`pathnames:pathname/1`)
`Base` **is a pathname (encoded as an atom)**	(`pathnames:pathname/1`)

- *The following properties should hold at call time:*

`Path` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)
`Dir` **is currently a term which is not a free variable.**	(`term_typing:nonvar/1`)

path_norm/2: PREDICATE

`path_norm(Path,NormPath)`

`NormPath` is the normalized pathname version of `Path`. Normalization removes redundant separators (`//` into `/`), and collapses references to the parent (`..`) and current (`.`) level. The parent of a root path (`/` or `//`) is itself. Empty paths (`''`) or relative paths where all parent references are collapsed are normalized as `'.'`.

For compatibility, leading double-slashes are preserved in normalization (the POSIX.2 standard states that "a pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash."). Some systems (like Linux) ignore double-slashes while others (like Cygwin for `//hostname/path` SMB network drives) do not.

Note that `path_norm/2` does not access the filesystem, which may affect the semantics when symbolic links are used (and no path normalization is involved). E.g., the following query obtain different values for `S1` and `S2`:

```
?- P = '/usr/donotexists/..',
   path_norm(P, N),
   process_call(path(test), ['-e', P], [status(S1)]),
   process_call(path(test), ['-e', N], [status(S2)]).

N = '/usr',
P = '/usr/donotexists/..',
S1 = 1,
S2 = 0 ?

?- use_module(library(process)).
?- copy_file('/usr', 'u', [symlink]),
   P = 'u/../u',
   path_norm(P, N),
   process_call(path(test), ['-e', P], [status(S1)]),
   process_call(path(test), ['-e', N], [status(S2)]),
   delete_file('u').

N = u,
P = 'u/../u',
S1 = 1,
S2 = 0 ?

yes
```

Usage: `path_norm(Path,NormPath)`

`NormPath` **is the normalized pathname version of** `Path`.

- *Call and exit should be compatible with:*

  `Path` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

  `NormPath` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

- *The following properties should hold at call time:*

  `Path` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

path_splitext/3: PREDICATE

`path_splitext(Path,NoExt,Ext)`

The extension `Ext` **is the shortest suffix that begins with** `'.'` **of** `Path`**. The rest of the pathname is** `NoExt`**, which cannot be empty (**`''`**). The extension is** `''` **if the pathname has no extension.**

In this example, all the following goals succeed:

```
path_splitext('a/foo.', 'a/foo', '.')
path_splitext('a/foo.c', 'a/foo', '.c')
path_splitext('a/foo.c.d', 'a/foo.c', '.d')
path_splitext('a/.foo.', 'a/.foo', '.')
path_splitext('a/.foo.c', 'a/.foo', '.c')
path_splitext('a/.foo.c.d', 'a/.foo.c', '.d')
```

Usage 1: `path_splitext(Path,NoExt,Ext)`

Split `Path` **into its extension** `Ext` **and the rest of the pathname** `NoExt`.

- *Call and exit should be compatible with:*

  `Path` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

  `NoExt` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

  `Ext` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

- *The following properties should hold at call time:*

  `Path` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

Usage 2: `path_splitext(Path,NoExt,Ext)`

Compose `Path` **by concatenating the extension** `Ext` **to** `NoExt` **pathname.**

- *Call and exit should be compatible with:*

  `Path` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

  `NoExt` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

  `Ext` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

- *The following properties should hold at call time:*

  `NoExt` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Ext` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

path_basename/2: PREDICATE

Usage: `path_basename(Path,Base)`

`Base` **is the basename corresponding to the** `Path` **(equivalent to** `path_split(Path,_,Base)`**).**

- *Call and exit should be compatible with:*
  `Path` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
  `Base` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
- *The following properties should hold at call time:*
  `Path` **is currently a term which is not a free variable.** (term_typing:nonvar/1)

path_dirname/2: PREDICATE

Usage: `path_dirname(Path,Dir)`

`Dir` **is the directory part corresponding to the** `Path` **(equivalent to** `path_split(Path,Dir,_)`**).**

- *Call and exit should be compatible with:*
  `Path` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
  `Dir` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
- *The following properties should hold at call time:*
  `Path` **is currently a term which is not a free variable.** (term_typing:nonvar/1)

path_relocate/4: PREDICATE

Usage: `path_relocate(FromDir,ToDir,FromPath,ToPath)`

Replace `FromDir` **prefix by** `DestDir` **in** `FromDir` **to generate** `ToDir`

- *Call and exit should be compatible with:*
  `FromDir` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
  `ToDir` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
  `FromPath` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
  `ToPath` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
- *The following properties should hold at call time:*
  `FromDir` **is currently a term which is not a free variable.** (term_typing:nonvar/1)
  `ToDir` **is currently a term which is not a free variable.** (term_typing:nonvar/1)
  `FromPath` **is currently a term which is not a free variable.** (term_typing:nonvar/1)
  `ToPath` **is a free variable.** (term_typing:var/1)

path_get_relative/3: PREDICATE

Usage: `path_get_relative(BaseDir,Path,RelPath)`

Obtain path `RelPath` **such that** `path_concat(BaseDir, RelPath, Path)` `RelPath` **will not contain any trailing** `'/'`

- *Call and exit should be compatible with:*
  `BaseDir` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
  `Path` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
  `RelPath` **is a pathname (encoded as an atom)** (pathnames:pathname/1)
- *The following properties should hold at call time:*
  `BaseDir` **is currently a term which is not a free variable.** (term_typing:nonvar/1)
  `Path` **is currently a term which is not a free variable.** (term_typing:nonvar/1)
  `RelPath` **is a free variable.** (term_typing:var/1)

path_split_list/2: PREDICATE

Usage: `path_split_list(Path,Bases)`

Split `Path` **into its components** `Bases`**, calling** `path_split/3` **recursively.**

- *Call and exit should be compatible with:*

  `Path` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

  `Bases` **is a list of** `pathnames`**.** (`basic_props:list/2`)
- *The following properties should hold at call time:*

  `Path` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

path_concat_list/2: PREDICATE

Usage: `path_concat_list(Bases,Path)`

Concatenate all components in `Bases` **a new path** `Path`**, calling** `path_concat/3` **recursively.**

- *Call and exit should be compatible with:*

  `Bases` **is a list of** `pathnames`**.** (`basic_props:list/2`)

  `Path` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)
- *The following properties should hold at call time:*

  `Bases` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

161.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `native_props`, `lists`, `system`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `nativeprops`.

162 Processes (multitasking)

Author(s): Jose F. Morales.

This library offers predicates to create, communicate, and synchronize with child processes. The child processes are duplicates of the parent process, based on the `fork()` function from POSIX-compatible operating systems (see `man` page for a precise description of the non-inherited process attributes).

The child process can execute a given goal (in a clone of the parent process) or start an external program.

Contrary to threads, processes have a separate address space and communication must be performed via inter-process communication mechanisms. This is useful for executing external programs, or implementing coarse-grained parallelism and concurrency.

- Process creation via `fork()` is relatively costly. Use only when address separation is necessary. Consider other concurrency primitives otherwise.
- Channels connected to in-memory terms are transmited via pipes or temporary files (when needed to avoid deadlock problems).
- Deadlock problems may still appear if the user specifies two or more `pipe(_)` channels for the same process (data must be send/received concurrently).
- The current implementation is not protected against Prolog signals (it can leak resources, including zombie processes, if interrupted by signals).
- Communication is currently only supported via file system, file descriptors, and sockets.
- Other mechanisms like file locks, semaphores, message queues, shared memory, etc. are not yet implemented.
- Arguments do not accept wildcards. See predicates in `library(glob)` for glob expansions (shell wildcard patterns).

162.1 Usage and interface

- **Library usage:**

  `:- use_module(library(process)).`
- **Exports:**
  - *Predicates:*

    `process_call/3`, `process_pid/2`, `process_is_joined/1`, `process_join/1`, `process_terminate/1`, `process_kill/1`, `process_send_signal/2`, `process_fork/2`, `process_pipe/2`.
  - *Regular Types:*

    `process/1`, `process_option/1`, `process_cmd/1`, `process_arg/1`.

162.2 Documentation on exports

process/1: REGTYPE

Usage:

A process handler

process_option/1: REGTYPE

The predicate `process_call/3` waits for process completion, throwing an exception if the return code is different than `0`. This default exit behaviour can be controlled with the following options:

`status(ReturnCode)`
: unifies return code with `ReturnCode` upon process completion (no exception is thrown, may fail).

`background(Process)`
: execute asynchronously in background; most errors (including input/output) are delayed to `process_join/1`.

`setsid`
: call `setsid()` on the child to create a new session (useful to create daemon processes `http://www.netzmafia.de/skripten/unix/linux-daemon-howto.html`)

The process standard input, output, and error file descriptors (streams from the Prolog side) can be bound to several `process_channel/1`:

`stdin(Channel)`
: set channel for standard input

`stdout(Channel)`
: set channel for standard output

`stderr(Channel)`
: set channel for standard error

The environment of the process can be modified with the following options:

`cwd(Dir)`
: execute command at the `Dir` directory (does not affect relative path for input/output redirection).

`env(Env)`
: modify the specified environment variables.

`noenv(Env)`
: unset the specified environment variables.

Usage:

Options that control command execution

process_cmd/1: REGTYPE

Usage:

Command for `process_call/3`

process_arg/1: REGTYPE

Usage:

Argument for `process_call/3`

process_call/3: PREDICATE

Usage: `process_call(Cmd,Args,Opts)`

Execute a command in a child process, where `Cmd` is the executable path. Use `path(Exec)` for executing a program `Exec` reachable from the `PATH` environment variable

– *The following properties should hold at call time:*
  Command for `process_call/3` (`process:process_cmd/1`)
  `Args` **is a list of** `process_args`. (`basic_props:list/2`)
  `Opts` **is a list of** `process_options`. (`basic_props:list/2`)

process_pid/2: PREDICATE

Usage: `process_pid(Process,Pid)`

The POSIX PID of the process `Process`.

– *The following properties should hold at call time:*
  A process handler (`process:process/1`)
  `Pid` **is an integer.** (`basic_props:int/1`)

process_is_joined/1: PREDICATE

Usage: `process_is_joined(Process)`

The process has already been joined.

– *The following properties should hold at call time:*
  A process handler (`process:process/1`)

process_join/1: PREDICATE

Usage: `process_join(Process)`

Wait for completion of process `Process`.

– *The following properties should hold at call time:*
  A process handler (`process:process/1`)

process_terminate/1: PREDICATE

Usage: `process_terminate(Process)`

Sends POSIX signal SIGTERM to the process `Process`, **which asks politely for process termination.**

– *The following properties should hold at call time:*
  A process handler (`process:process/1`)

process_kill/1: PREDICATE

Usage: `process_kill(Process)`

Sends POSIX signal SIGKILL to the process `Process`, **which forces process termination.**

– *The following properties should hold at call time:*
  A process handler (`process:process/1`)

process_send_signal/2: PREDICATE

Usage: `process_send_signal(Process,Signal)`

Sends POSIX signal `Signal` to process `Process`.

– *The following properties should hold at call time:*

  A process handler (`process:process/1`)

  `Signal` **is an integer.** (`basic_props:int/1`)

process_fork/2: PREDICATE

Usage: `process_fork(Goal,Opts)`

Execute `Goal` in a forked process.

– *The following properties should hold at call time:*

  `Goal` **is a term which represents a goal, i.e., an atom or a structure.** (`basic_props:cgoal/1`)

  `Opts` **is a list of** `process_options`**.** (`basic_props:list/2`)

Meta-predicate **with arguments:** `process_fork(goal,?)`.

process_pipe/2: PREDICATE

Usage: `process_pipe(Cmd,Opts)`

Execute the list `Cmds` of `process_call/3`, connecting standard input and output of consecutive processes with pipes.

Options are passed as follows: input redirection options are applied to the first process; output redirection and status are applied to the last process; the rest of options are applied to all commands. Background execution is not currently supported (see internal documentation for details).

162.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

  `lists`, `port_reify`, `system`, `process_channel`.

– *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `isomodes`, `hiord`.

163 Process channels

Author(s): Jose F. Morales.

This module provides an abstraction for communication channels between processes (`process`) based on standard input/output/error streams.

163.1 Usage and interface

- **Library usage:**
  `:- use_module(library(process/process_channel)).`
- **Exports:**
  - *Predicates:*
    `channel_bindings/2`, `cleanup_binding/1`, `binding_port_call/1`, `send_input/2`, `receive_output/2`, `open_redirect/3`, `close_redirect/2`.
  - *Regular Types:*
    `process_channel/1`.

163.2 Documentation on exports

process_channel/1: REGTYPE

`default` the default descriptor

`null` the null stream (see `dev_null/1`)

`pipe(-Stream)`
a pipe (`Stream` unified with the parent's read/write end of the pipe)

`file(+File)`
a file `File`

`stream(+Stream)`
a stream `Stream` (must be opened with the right mode)

`file_append(+File)`
a file `File`, where writes are appended to the end

`string(?String)`
a string (list of codes)

`line(?Line)`
a string, ignoring trailing new line character

`atmlist(?Xs)`
a list of atoms (for each line)

`terms(?Xs)`
terms, ended in full-stop (using `read_term/3` and `write_canonical/2`)

`stdout` (only valid for `stderr(_)`) redirect to the same channel than stdout (useful for redirecting both standard output and standard error).

Usage: `process_channel(Channel)`

A communication channel for standard file descriptors

channel_bindings/2: PREDICATE

Usage: `channel_bindings(Channels,ChannelBinds)`

Create channel bindings (taking into account pipes)

cleanup_binding/1: PREDICATE

Usage: `cleanup_binding(ChannelB)`

Cleanup temporaries due to channel file-based bindings.

binding_port_call/1: PREDICATE

Usage: `binding_port_call(ChannelB)`

Do port_call/1 on the result of channel transfer (send or receive).

send_input/2: PREDICATE

Usage: `send_input(Mode,ChannelBinding)`

Send input through channel binding `ChannelBinding`. Transfer status is internally stored (see `binding_port_call/1`).

receive_output/2: PREDICATE

Usage: `receive_output(Mode,ChannelBinding)`

Receive output from channel binding `ChannelBinding`. Transfer status is internally stored (see `binding_port_call/1`).

open_redirect/3: PREDICATE

Usage: `open_redirect(ChannelB,Mode,S)`

Open stream file redirections (for internals:'$exec'/9).

close_redirect/2: PREDICATE

Usage: `close_redirect(ChannelB,S)`

Close stream file redirections (for internals:'$exec'/9).

163.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `lists`, `port_reify`, `system`, `read`, `write`, `stream_utils`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `isomodes`, `hiord`.

164 PART VIII - Additional libraries

Author(s): The Ciao Development Team.

This part documents several Ciao libraries which provide different useful additional functionality. Such functionality includes performing operating system calls, gathering statistics from the Ciao engine, file and filename manipulation, error and exception handling, fast reading and writing of terms (marshalling and unmarshalling), file locking, issuing program and error messages, pretty-printing programs and assertions, a browser of the system libraries, additional expansion utilities, concurrent aggregates, graph visualization, etc.

165 Accessing and redirecting the stream aliases

Author(s): Manuel Carro.

This library allows the redefinition of the files to which the special streams `user_input`, `user_output`, and `user_error` point to. On startup they point to the standard input, standard output, and standard error, in Unix style (Windows users may find that standard error stream does not work properly). Changing the file pointed to is useful for, e.g., redirecting the place to which the Prolog's standard error stream goes from within Prolog (e.g., to start a log file).

165.1 Usage and interface

- **Library usage:**
  `:- use_module(library(io_alias_redirection)).`
- **Exports:**
  - *Predicates:*
    `set_stream/3`, `get_stream/2`.

165.2 Documentation on exports

set_stream/3: PREDICATE

Usage: `set_stream(StreamAlias,NewStream,OldStream)`

Associate `StreamAlias` with an open stream `newStream`. Returns in `OldStream` the stream previously associated with the alias. The mode of `NewStream` must match the intended use of `StreamAlias`.

- *The following properties should hold at call time:*

  `StreamAlias` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `NewStream` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `StreamAlias` **is the alias of an open stream, i.e., an atom which represents a stream at Prolog level.** (`stream_basic:stream_alias/1`)

  `NewStream` **is an open stream.** (`stream_basic:stream/1`)

  `OldStream` **is an open stream.** (`stream_basic:stream/1`)

get_stream/2: PREDICATE

Usage: `get_stream(StreamAlias,Stream)`

Return in `Stream` the stream associated with `StreamAlias`.

- *The following properties should hold at call time:*

  `StreamAlias` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `StreamAlias` **is the alias of an open stream, i.e., an atom which represents a stream at Prolog level.** (`stream_basic:stream_alias/1`)

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

165.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  prelude, initial, condcomp, assertions, assertions/assertions_basic, basicmodes.

166 Call goals with reified (exit) ports.

Author(s): Jose F. Morales.

Predicates to delay the execution of the (exit) port of a goal. This is useful to introduce side-effects at selected points (e.g., clean-ups before goal exit in failure or exception conditions).

166.1 Usage and interface

- **Library usage:**
  `:- use_module(library(port_reify)).`
- **Exports:**
  - *Predicates:*
    `once_port_reify/2`, `port_call/1`.
  - *Regular Types:*
    `exit_port/1`.

166.2 Documentation on exports

exit_port/1: REGTYPE

`exit_port(Result)`

Exit port of a goal execution, defined as:

- `success`: **goal succeeded**
- `failure`: **goal failed**
- `exception(E)`: **goal threw an exception** `E`

once_port_reify/2: PREDICATE

Usage: `once_port_reify(Goal,Result)`

Execute `once(Goal)` **(alternatives are cut) and obtain its execution port** `Port`**, which can be continued with** `port_call/1`**.**

The sequence `(once_port_reify(Goal,P),Cleanup,port_call(P))` **is semantically equivalent to** `once(Goal)`**, but executes** `Cleanup` **goals in case of success, failure, and exception.**

- *The following properties should hold upon exit:*
  `port_reify:exit_port(Result)` (`port_reify:exit_port/1`)

Meta-predicate **with arguments:** `once_port_reify(goal,?)`.

port_call/1: PREDICATE

Usage: `port_call(Result)`

Succeed, fail, or throw the exception from `Result` **(see** `once_port_reify/2`**).**

- *The following properties should hold at call time:*
  `port_reify:exit_port(Result)` (`port_reify:exit_port/1`)

166.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `isomodes`, `hiord`.

167 Call goals with reified IO and (exit) ports

Author(s): Jose F. Morales.

Predicates to delay the execution of the (exit) port of a goal and capture the OS standard output/error streams (stdout,stderr).

167.1 Usage and interface

- **Library usage:**
  `:- use_module(library(io_port_reify)).`
- **Exports:**
  - *Predicates:*
    `io_once_port_reify/3`, `io_once_port_reify/4`, `open_std_redirect/3`, `close_std_redirect/1`.

167.2 Documentation on exports

io_once_port_reify/3: PREDICATE

Usage: `io_once_port_reify(Goal,Port,OutString)`

Executes `Goal`, `Port` is the state when the predicate finishes (true, fail). Its stdout is stored in `OutString`.

- *The following properties should hold upon exit:*
  `OutString` **is a string (a list of character codes).** (`basic_props:string/1`)

Meta-predicate **with arguments:** `io_once_port_reify(goal,?,?)`.

io_once_port_reify/4: PREDICATE

Usage: `io_once_port_reify(Goal,Port,OutString,ErrString)`

Executes `Goal`, `Port` is the state when the predicate finishes (true, fail). Its stdout and stderr are stored in `OutString` and `ErrString` respectively.

- *The following properties should hold upon exit:*
  `OutString` **is a string (a list of character codes).** (`basic_props:string/1`)
  `ErrString` **is a string (a list of character codes).** (`basic_props:string/1`)

Meta-predicate **with arguments:** `io_once_port_reify(goal,?,?,?)`.

open_std_redirect/3: PREDICATE

No further documentation available for this predicate.

close_std_redirect/1: PREDICATE

No further documentation available for this predicate.

167.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `system`, `system_extra`, `stream_utils`, `port_reify`, `io_alias_redirection`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `isomodes`, `hiord`.

168 ctrlcclean (library)

168.1 Usage and interface

- **Library usage:**
  `:- use_module(library(ctrlcclean)).`
- **Exports:**
  - *Predicates:*
    `ctrlc_clean/1`, `delete_on_ctrlc/2`, `ctrlcclean/0`.

168.2 Documentation on exports

ctrlc_clean/1: PREDICATE

No further documentation available for this predicate. *Meta-predicate* **with arguments:** `ctrlc_clean(goal)`.

delete_on_ctrlc/2: PREDICATE

No further documentation available for this predicate.

ctrlcclean/0: PREDICATE

No further documentation available for this predicate.

168.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`, `system`.
- *Packages:*
  `noprelude`, `initial`, `condcomp`, `datafacts`.

169 Default exception handler and pretty printer

Author(s): The Ciao Development Team, Jose F. Morales.

169.1 Usage and interface

- **Library usage:**
  `:- use_module(library(errhandle)).`
- **Exports:**
  - *Predicates:*
    `error_protect/2`, `default_error_message/1`.

169.2 Documentation on exports

error_protect/2: PREDICATE

Usage: `error_protect(Goal,OnError)`

Execute `Goal` with a default exception handler (which shows the exception and executes `OnError`).

Meta-predicate **with arguments:** `error_protect(goal,goal)`.

default_error_message/1: PREDICATE

Usage: `default_error_message(E)`

Default pretty printer for the exception term `E`.

169.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `system`, `rtchecks_pretty`.
- *Packages:*
  `noprelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

170 Term manipulation utilities

Author(s): The Ciao Development Team.

This module implements some utils to do term manipulation.

170.1 Usage and interface

- **Library usage:**
  `:- use_module(library(terms)).`
- **Exports:**
  - *Predicates:*
    `term_size/2`, `copy_args/3`, `arg/2`, `atom_concat/2`.

170.2 Documentation on exports

term_size/2: PREDICATE

Usage: `term_size(Term,N)`

Determines the size of a term.

- *The following properties should hold upon exit:*
  `N` **is a non-negative integer.** (`basic_props:nnegint/1`)

copy_args/3: PREDICATE

Usage: `copy_args(N,Term,Copy)`

`Term` **and** `Copy` **have the same first** `N` **arguments.**

- *The following properties should hold at call time:*
  `N` **is a non-negative integer.** (`basic_props:nnegint/1`)

Other properties:

`copy_args(N,Term,Copy)`

- *If the following properties hold at call time:*
  `N` **is currently ground (it contains no variables).** (`term_typing:ground/1`)
  `Term` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  then the following properties hold globally:
  `copy_args(N,Term,Copy)` **is evaluable at compile-time.** (`basic_props:eval/1`)

`copy_args(N,Term,Copy)`

- *The following properties hold globally:*
  `copy_args(N,Term,Copy)` **is side-effect** `free`**.** (`basic_props:sideff/2`)

arg/2: PREDICATE

Usage: `arg(Term,Arg)`

`Arg` **is an argument of** `Term`**. Gives each of the arguments on backtracking.**

atom_concat/2: PREDICATE

`atom_concat(Atms,Atm)`

`Atm` is the atom resulting from concatenating all atoms in the list `Atms` in the order in which they appear. If `Atm` is an atom at call then `Atms` can contain free variables.

Usage 1: `atom_concat(Atms,Atm)`

- *The following properties should hold at call time:*

  `Atms` is a list of `atms`. (`basic_props:list/2`)
- *The following properties should hold upon exit:*

  `Atm` is an atom. (`basic_props:atm/1`)

Usage 2: `atom_concat(Atms,Atm)`

- *The following properties should hold at call time:*

  `Atm` is an atom. (`basic_props:atm/1`)
- *The following properties should hold upon exit:*

  `Atms` is a list of `atms`. (`basic_props:list/2`)

170.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `native_props`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nativeprops`.

171 Term checking utilities

Author(s): The Ciao Development Team.

This module implements a basic set of term checking utilities.

171.1 Usage and interface

- **Library usage:**
  `:- use_module(library(terms_check)).`
- **Exports:**
  - *Predicates:*
    `variant/2`, `ask/2`, `subsumes_term/2`, `most_specific_generalization/3`, `most_general_instance/3`, `unifiable/3`.
  - *Properties:*
    `instance/2`.
  - *Regular Types:*
    `unifier/1`.

171.2 Documentation on exports

variant/2: PREDICATE

`variant(Term1,Term2)`

`Term1` **and** `Term2` **are identical up to renaming.**

ask/2: PREDICATE

`ask(Term1,Term2)`

`Term1` **and** `Term2` **unify without producing bindings for the variables of** `Term1`**. I.e.,** `instance(Term1,Term2)` **holds.**

instance/2: PROPERTY

Usage: `instance(Term1,Term2)`

`Term1` **is an instance of** `Term2`**.**

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`basic_props:native/1`)

subsumes_term/2: PREDICATE

Usage: `subsumes_term(Term1,Term2)` ⟨• ISO •⟩

`Term2` **is an instance of** `Term1`**.**

most_specific_generalization/3: PREDICATE

`most_specific_generalization(Term1,Term2,Term)`

`Term` satisfies `instance(Term1,Term)` and `instance(Term2,Term)` and there is no term less general than `Term` (modulo variants) that satisfies it.

most_general_instance/3: PREDICATE

`most_general_instance(Term1,Term2,Term)`

`Term` satisfies `instance(Term,Term1)` and `instance(Term,Term2)` and there is no term more general than `Term` (modulo variants) that satisfies it.

unifier/1: REGTYPE

```
unifier(Unifier) :-
    list(unifier_elem,Unifier).
```

Usage: `unifier(X)`

`X` is a unifier.

unifiable/3: PREDICATE

Usage: `unifiable(X,Y,Unifier)`

Unifies `Unifier` with the most general unifier between the terms `X` and `Y`. Fails if such unifier does not exit.

- *The following properties should hold at call time:*
  `X` is any term. (`basic_props:term/1`)
  `Y` is any term. (`basic_props:term/1`)
- *The following properties hold upon exit:*
  `Unifier` is a unifier. (`terms_check:unifier/1`)

171.3 Documentation on internals

unifier_elem/1: REGTYPE

A regular type, defined as follows:

```
unifier_elem(X=Term) :-
    var(X),
    term(Term).
```

171.4 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `noprelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `nortchecks`.

171.5 Other information

Currently, `ask/2` and `instance/2` are exactly the same. However, `ask/2` is intended to be more general, since it is also applicable to constraint domains (although not yet implemented): i.e., for the particular case of Herbrand terms, `ask/2` is just `instance/2`.

172 Sets of variables in terms

Author(s): The Ciao Development Team.

This module implements predicates to handle sets of variables in terms.

172.1 Usage and interface

- **Library usage:**
  `:- use_module(library(terms_vars)).`
- **Exports:**
  - *Predicates:*
    `varset/2`, `intersect_vars/3`, `member_var/2`, `diff_vars/3`, `varsbag/3`, `varset_in_args/2`, `term_variables/2`, `term_variables/3`.

172.2 Documentation on exports

varset/2: PREDICATE

`varset(Term,Xs)`

`Xs` is the sorted list of all the variables in `Term`.

intersect_vars/3: PREDICATE

No further documentation available for this predicate.

member_var/2: PREDICATE

No further documentation available for this predicate.

diff_vars/3: PREDICATE

No further documentation available for this predicate.

varsbag/3: PREDICATE

`varsbag(Term,Vs,Xs)`

`Vs` is the list of all the variables in `Term` ordered as they appear in `Term` right-to-left depth-first (including duplicates) plus `Xs`.

varset_in_args/2: PREDICATE

Usage: `varset_in_args(T,LL)`

Each list of `LL` contains the variables of an argument of `T`, for each argument, and in left to right order.

- *The following properties should hold at call time:*
  `T` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
- *The following properties should hold upon exit:*
  `LL` is a list of `list(var)`s. (`basic_props:list/2`)

term_variables/2: PREDICATE

Usage: `term_variables(Term,Vars)` ⟨• ISO •⟩

`Vars` is the list of all the variables in `Term`, ordered as they appear in `Term` right-to-left depth-first (without duplicates).

term_variables/3: PREDICATE

`term_variables(Term,Vars,Tail)`

`Vars-Tail` is the difference list of all the variables in `Term`, ordered as they appear in `Term` right-to-left depth-first (without duplicates).

172.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `idlists`, `sort`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

173 Cyclic terms handling

Author(s): Daniel Cabeza, Remy Haemmerle.

This module implements predicates related to cyclic terms. Cyclic (or infinite) terms are produced when unifying a variable with a term which contains that variable.

173.1 Usage and interface

- **Library usage:**
  `:- use_module(library(cyclic_terms)).`
- **Exports:**
  - *Predicates:*
    `acyclic_term/1`, `uncycle_term/2`, `recycle_term/2`, `cyclic_term/1`, `cyclic_term/1`, `cyclic_term/1`.

173.2 Documentation on exports

acyclic_term/1: PREDICATE

Usage: `acyclic_term(T)` ⟨• ISO •⟩

True if `T` is acyclic (finite).

uncycle_term/2: PREDICATE

Usage: `uncycle_term(T,U)`

Given a term `T`, `U` is a finite representation of `T` as an acyclic term. This representation can be converted back to `T` using `recycle_term/2`.

recycle_term/2: PREDICATE

Usage: `recycle_term(U,T)`

Given `U`, a finite representation of a term as an acyclic term as `uncycle_term/2` produces, `T` is the represented term. `U` is modified by the predicate, thus to maintain it untouched `copy_term/2` should be used.

cyclic_term/1: PREDICATE

Usage: `cyclic_term(T)`

True if `T` is cyclic (infinite).

cyclic_term/1: PREDICATE

Usage: `cyclic_term(T)`

True if `T` is cyclic (infinite).

cyclic_term/1: PREDICATE

Usage: `cyclic_term(T)`

True if `T` is cyclic (infinite).

173.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`lists`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

174 Fast reading and writing of terms

Author(s): Daniel Cabeza, Oscar Portela Arjona.

This library provides predicates to support reading / writing of terms on a format designed to be:

- **Simple to implement at the engine level, with very few dependencies.**
- **Faster to handle than the standard representation.**

174.1 Usage and interface

- **Library usage:**
  `:- use_module(library(fastrw)).`
- **Exports:**
  - *Predicates:*
    `fast_read/1`, `fast_read/2`, `fast_write/1`, `fast_write/2`.

174.2 Documentation on exports

fast_read/1: PREDICATE

`fast_read(Term)`

The next term is read from current standard input and is unified with `Term`. The syntax of the term must agree with fast_read / fast_write format. If the end of the input has been reached, `Term` is unified with the term 'end_of_file'. Further calls to `fast_read/1` will then cause an error.

fast_read/2: PREDICATE

`fast_read(Stream,Term)`

The next term is read from `Stream` and unified with `Term`. The syntax of the term must agree with fast_read / fast_write format. If the end of the input has been reached, `Term` is unified with the term 'end_of_file'. Further calls to `fast_read/2` will then cause an error.

Usage:

- *Call and exit should be compatible with:*
  `Term` **is any term.** (`basic_props:term/1`)
- *The following properties should hold at call time:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold upon exit:*
  `Term` **is any term.** (`basic_props:term/1`)

fast_write/1: PREDICATE

`fast_write(Term)`

Output `Term` in a way that `fast_read/1` and `fast_read/2` will be able to read it back.

fast_write/2: PREDICATE

`fast_write(Stream,Term)`

Output `Term` to `Stream` in a way that `fast_read/1` and `fast_read/2` will be able to read it back.

Usage:

- *The following properties should hold at call time:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

  `Term` **is any term.** (`basic_props:term/1`)
- *The following properties should hold upon exit:*

  `Term` **is any term.** (`basic_props:term/1`)
- *The following properties should hold globally:*

  `Term` **is not further instantiated.** (`basic_props:not_further_inst/2`)

174.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

175 Modules as blobs

Author(s): Isabel Garcia-Contreras, Jose F. Morales.

This module implements (temporary) modules-as-blobs facilities. They allow the runtime creation of modules from strings and terms.

175.1 Implementation

Run-time creation of modules is currently implemented by writing the module code into a unique temporary directory (using the system `mktemp_in_tmp/2` predicate).

Currently they need to be deleted explicitly.

Note that this is a temporary solution before read/write stream operations can be work on in-memory objects.

175.2 Examples

The predicate `new_modblob/4` creates a new temporary module, that can be deleted with `delete_modblob/1`.

Module creation requires the module name, export list, program clauses, and returns the module object (currently a path). If a `module` directive is specified, the export list is ignored.

Example:

```
?- new_modblob([(p(A) :- q(A)), (q(b))], [p/1], my_tmp_mod, ModBlob), modblob_path(ModBlob, Path).
Path = '/tmp/tmp_modsfbMY9Y/my_tmp_mod.pl' ?
```

where the contents of `/tmp/tmp_modsfbMY9Y/my_tmp_mod.pl` are:

```
:-module(_,[p/1],[]).
p(A) :-
    q(A).
q(b).
```

175.3 Usage and interface

- **Library usage:**
  `:- use_module(library(modblobs)).`
- **Exports:**
  - *Predicates:*
    `new_modblob/4`, `delete_modblob/1`, `modblob_path/2`.

175.4 Documentation on exports

new_modblob/4: PREDICATE

Usage: `new_modblob(Clauses,ExportedPreds,ModName,ModBlob)`

Write clauses `Clauses` in a temporary module identified by `ModBlob`.

- *The following properties should hold at call time:*
  `Clauses` **is a list.** (`basic_props:list/1`)
  `ExportedPreds` **is a list.** (`basic_props:list/1`)
  `ModName` **is an atom.** (`basic_props:atm/1`)
  `ModBlob` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Clauses` **is a list.** (`basic_props:list/1`)
  `ExportedPreds` **is a list.** (`basic_props:list/1`)
  `ModName` **is an atom.** (`basic_props:atm/1`)
  `ModBlob` **is a is a temporary module object** (`modblobs:modblob/1`)

delete_modblob/1: PREDICATE

Usage: `delete_modblob(ModBlob)`

Removes the data associated to `ModBlob`

- *The following properties should hold at call time:*
  `ModBlob` **is a is a temporary module object** (`modblobs:modblob/1`)
- *The following properties should hold globally:*
  All calls of the form `delete_modblob(ModBlob)` **are deterministic.** (`native_props:is_det/1`)

modblob_path/2: PREDICATE

Usage: `modblob_path(ModBlob,ModPath)`

Obtain the temporary file associated to `ModBlob`

- *The following properties should hold at call time:*
  `ModBlob` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `ModPath` **is an atom.** (`basic_props:atm/1`)

175.5 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `native_props`, `datafacts_rt`, `system`, `system_extra`, `write`, `pathnames`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `modes`, `regtypes`, `nativeprops`, `datafacts`.

176 Lists and conjunctions and disjunctions

176.1 Usage and interface

- **Library usage:**
  `:- use_module(library(formulae)).`
- **Exports:**
  - *Predicates:*
    `list_to_conj/3`, `list_to_conj/2`, `conj_to_list/2`, `list_to_disj/2`, `disj_to_list/2`, `conj_to_llist/2`, `llist_to_conj/2`, `disj_to_llist/2`, `llist_to_disj/2`, `body2list/2`, `asbody_to_conj/2`, `list_to_disj2/2`.
  - *Properties:*
    `assert_body_type/1`.
  - *Regular Types:*
    `conj_disj_type/1`, `t_conj/1`, `t_disj/1`.

176.2 Documentation on exports

list_to_conj/3: PREDICATE

`list_to_conj(List,Conj,End)`

`Conj` is the conjunction made up of the elements of `List` plus a final element `End`.

list_to_conj/2: PREDICATE

`list_to_conj(A,B)`

Examples:

```
?- list_to_conj(A, a).

A = [a] ? ;

no
?- list_to_conj(A, (a,V,b)).

A = [a,V,b] ? ;

no
?- list_to_conj([A], B).

B = A ? ;

no
?- list_to_conj([a,A,b], B).

B = (a,A,b) ? ;
```

```
no
?- list_to_conj([], B).

B = true ? ;

no
```

Usage 1: `list_to_conj(A,B)`

`Conj` **is the conjunction made up of the elements of** `List`**. The empty list is (**`[]`**) is** `true`**).**

- *The following properties should hold at call time:*

  `A` **is a list.** (`basic_props:list/1`)

  `B` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*

  Conjuntions (`formulae:t_conj/1`)

Usage 2: `list_to_conj(A,B)`

- *The following properties should hold at call time:*

  `A` **is a free variable.** (`term_typing:var/1`)

  Conjuntions (`formulae:t_conj/1`)
- *The following properties should hold upon exit:*

  `A` **is a list.** (`basic_props:list/1`)

conj_to_list/2: PREDICATE

`conj_to_list(Conj,List)`

`List` **is the list made up of the elements of conjunction** `Conj` **(**`true` **is** `[]`**).**

list_to_disj/2: PREDICATE

`list_to_disj(A,B)`

Examples:

```
?- list_to_disj([a], A).

A = a ? ;

no
?- list_to_disj([a,B,b], A).

A = (a;B;b) ? ;

no
?- list_to_disj(A, (a)).

A = [a] ? ;

no
?- list_to_disj(A, (a;B;b)).
```

```
A = [a,B,b] ? ;

no
?-
```

Usage 1: `list_to_disj(A,B)`

`Disj` **is the disjunction made up of the elements of** `List`. **(**`[]` **is** `false`**).**

- *The following properties should hold at call time:*
  `A` **is a list.** (`basic_props:list/1`)
  `B` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  Disjunctions (`formulae:t_disj/1`)

Usage 2: `list_to_disj(A,B)`

- *The following properties should hold at call time:*
  `A` **is a free variable.** (`term_typing:var/1`)
  Disjunctions (`formulae:t_disj/1`)
- *The following properties should hold upon exit:*
  `A` **is a list.** (`basic_props:list/1`)

disj_to_list/2: PREDICATE

`disj_to_list(Disj,List)`

`List` **is the list made up of the elements of disjunction** `Disj` **(**`true` **is** `[]`**).**

conj_to_llist/2: PREDICATE

Turns a conjunctive (normal form) formula into a list (of lists of ...). As a side-effect, inner conjunctions get flattened. No special care for `true`**.**

llist_to_conj/2: PREDICATE

Inverse of `conj_to_llist/2`**. No provisions for anything else than a non-empty list on input (i.e., they will go 'as are' in the output.**

disj_to_llist/2: PREDICATE

Turns a disjunctive (normal form) formula into a list (of lists of ...). As a side-effect, inner disjunctions get flattened. No special care for `true`**.**

llist_to_disj/2: PREDICATE

Inverse of `disj_to_llist/2`**. No provisions for anything else than a non-empty list on input (i.e., they will go 'as are' in the output.**

body2list/2: PREDICATE

No further documentation available for this predicate.

asbody_to_conj/2: PREDICATE

Usage 1: `asbody_to_conj(A,B)`

Transforms assertion body `A` into a conjuntion (`B`). It runs in both ways

- *The following properties should hold at call time:*
  `formulae:assert_body_type(A)` (`formulae:assert_body_type/1`)
  `B` is a free variable. (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  The usual prolog way of writing conjuntions and disjuntions in a body using ',' and ';' (`formulae:conj_disj_type/1`)

Usage 2: `asbody_to_conj(A,B)`

- *The following properties should hold at call time:*
  `A` is a free variable. (`term_typing:var/1`)
  The usual prolog way of writing conjuntions and disjuntions in a body using ',' and ';' (`formulae:conj_disj_type/1`)
- *The following properties should hold upon exit:*
  `formulae:assert_body_type(A)` (`formulae:assert_body_type/1`)

assert_body_type/1: PREDICATE

No further documentation available for this predicate.

conj_disj_type/1: REGTYPE

Usage:

The usual prolog way of writing conjuntions and disjuntions in a body using ',' and ';'

t_conj/1: REGTYPE

Usage:

Conjuntions

t_disj/1: REGTYPE

Usage:

Disjunctions

list_to_disj2/2: PREDICATE

No further documentation available for this predicate.

176.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

177 counters (library)

177.1 Usage and interface

- **Library usage:**
  `:- use_module(library(counters)).`
- **Exports:**
  - *Predicates:*
    `setcounter/2`, `getcounter/2`, `inccounter/2`, `deccounter/2`.

177.2 Documentation on exports

setcounter/2: PREDICATE

No further documentation available for this predicate.

getcounter/2: PREDICATE

No further documentation available for this predicate.

inccounter/2: PREDICATE

No further documentation available for this predicate.

deccounter/2: PREDICATE

No further documentation available for this predicate.

177.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `datafacts`.

178 Enumeration of integers inside a range

Author(s): The Ciao Development Team.

This modules enumerates integers between two numbers, or checks that an integer lies within a range

178.1 Usage and interface

- **Library usage:**
  `:- use_module(library(between)).`
- **Exports:**
  - *Predicates:*
    `between/3`.

178.2 Documentation on exports

between/3: PREDICATE

Usage: `between(Min,Max,N)`

`N` **is a number which is greater than or equal to** `Min` **and smaller than or equal to** `Max`**. Both** `Min` **and** `Max` **can be either integer or real numbers.**

- *The following properties should hold at call time:*
  `Min` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Max` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Min` **is a number.** (`basic_props:num/1`)
  `Max` **is a number.** (`basic_props:num/1`)
- *The following properties should hold upon exit:*
  `N` **is an integer.** (`basic_props:int/1`)

178.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`.

179 A simple pretty-printer for Ciao programs

Author(s): The Ciao Development Team.

This library module writes out to standard output a clause or a list of clauses.

179.1 Usage and interface

- **Library usage:**
  `:- use_module(library(pretty_print)).`
- **Exports:**
  - *Predicates:*
    `pretty_print/2`, `pretty_print/3`, `pretty_print/4`.

179.2 Documentation on exports

pretty_print/2: PREDICATE

Usage: `pretty_print(Cls,Flags)`

Prints each clause in the list `Cls` after numbering its variables.

- *The following properties should hold at call time:*

  `pretty_print:clauses(Cls)` (`pretty_print:clauses/1`)

  `Flags` **is a list of** `flags`. (`basic_props:list/2`)

pretty_print/3: PREDICATE

Usage: `pretty_print(Cls,Flags,Ds)`

Prints each clause in the list `Cls` after using the corresponding variable names dictionary in `Ds` to name its variables.

- *The following properties should hold at call time:*

  `pretty_print:clauses(Cls)` (`pretty_print:clauses/1`)

  `Flags` **is a list of** `flags`. (`basic_props:list/2`)

  `Ds` **is a dictionary of variable names.** (`vndict:varnamedict/1`)

pretty_print/4: PREDICATE

No further documentation available for this predicate.

179.3 Documentation on internals

clauses/1: REGTYPE

A regular type, defined as follows:

```
clauses([]).
clauses([_1|_2]) :-
    clause(_1),
    clauses(_2).
clauses(_1) :-
    clause(_1).
```

clause/1: REGTYPE

A regular type, defined as follows:

```
clause(_1) :-
    clterm(_1).
clause((_1,_2)) :-
    clterm(_1),
    term(_2).
```

clterm/1: REGTYPE

A regular type, defined as follows:

```
clterm(clause(_1,_2)) :-
    cgoal(_1),
    body(_2).
clterm(directive(_1)) :-
    body(_1).
clterm((_1:-_2)) :-
    cgoal(_1),
    body(_2).
clterm(_1) :-
    cgoal(_1).
```

body/1: REGTYPE

A well formed body, including cge expressions and &-concurrent expressions. The atomic goals may or may not have a key in the form `^(goal:any)`, and may or may not be module qualified, but if they are it has to be in the form `^(^(moddesc:goal):any)`.

Usage: `body(X)`

`X` is a printable body.

flag/1: REGTYPE

A keyword `ask/1` flags whether to output *asks* or *whens* and `nl/1` whether to separate clauses with a blank line or not.

Usage: `flag(X)`

`X` is a flag for the pretty-printer.

179.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `operators`, `vndict`, `streams`, `write`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `fsyntax`.

180 Pretty-printing assertions

Author(s): Francisco Bueno.

This module defines some predicates which are useful for writing assertions in a readable form.

180.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assertions/assrt_write)).`
- **Exports:**
  - *Predicates:*
    `write_assertion/6`, `write_assertion/7`, `write_assertion_as_comment/6`, `write_assertion_as_comment/7`, `write_assertion_as_double_comment/6`, `write_assertion_as_double_comment/7`.

180.2 Documentation on exports

write_assertion/6: PREDICATE

Usage: `write_assertion(Goal,Status,Type,Body,Dict,Flag)`

Writes the (normalized) assertion to current output.

- *Call and exit should be compatible with:*
  `Status` **is an acceptable status for an assertion.** (`assertions_props:assrt_status/1`)
  `Type` **is an admissible kind of assertion.** (`assertions_props:assrt_type/1`)
  `Body` **is a normalized assertion body.** (`assertions_props:nabody/1`)
  `Dict` **is a dictionary of variable names.** (`assertions_props:dictionary/1`)
  `Flag` **is** `status` **or** `nostatus`**.** (`assrt_write:status_flag/1`)

write_assertion/7: PREDICATE

Usage: `write_assertion(Stream,Goal,Status,Type,Body,Dict,Flag)`

Writes the (normalized) assertion to stream `Stream`**.**

- *Call and exit should be compatible with:*
  `Status` **is an acceptable status for an assertion.** (`assertions_props:assrt_status/1`)
  `Type` **is an admissible kind of assertion.** (`assertions_props:assrt_type/1`)
  `Body` **is a normalized assertion body.** (`assertions_props:nabody/1`)
  `Dict` **is a dictionary of variable names.** (`assertions_props:dictionary/1`)
  `Flag` **is** `status` **or** `nostatus`**.** (`assrt_write:status_flag/1`)

write_assertion_as_comment/6: PREDICATE

Usage: `write_assertion_as_comment(Goal,Status,Type,Body,Dict,Flag)`

Writes the (normalized) assertion to current output as a Prolog comment.

– *Call and exit should be compatible with:*

`Status` **is an acceptable status for an assertion.** (`assertions_props:assrt_status/1`)

`Type` **is an admissible kind of assertion.** (`assertions_props:assrt_type/1`)

`Body` **is a normalized assertion body.** (`assertions_props:nabody/1`)

`Dict` **is a dictionary of variable names.** (`assertions_props:dictionary/1`)

`Flag` **is** `status` **or** `nostatus`**.** (`assrt_write:status_flag/1`)

write_assertion_as_comment/7: PREDICATE

Usage: `write_assertion_as_comment(Stream,Goal,Status,Type,Body,Dict,Flag)`

Writes the (normalized) assertion to stream `Stream` **as a Prolog comment.**

– *Call and exit should be compatible with:*

`Status` **is an acceptable status for an assertion.** (`assertions_props:assrt_status/1`)

`Type` **is an admissible kind of assertion.** (`assertions_props:assrt_type/1`)

`Body` **is a normalized assertion body.** (`assertions_props:nabody/1`)

`Dict` **is a dictionary of variable names.** (`assertions_props:dictionary/1`)

`Flag` **is** `status` **or** `nostatus`**.** (`assrt_write:status_flag/1`)

write_assertion_as_double_comment/6: PREDICATE

Usage: `write_assertion_as_double_comment(Goal,Status,Type,Body,Dict,Flag)`

Writes the (normalized) assertion to current output as a Prolog double comment.

– *Call and exit should be compatible with:*

`Status` **is an acceptable status for an assertion.** (`assertions_props:assrt_status/1`)

`Type` **is an admissible kind of assertion.** (`assertions_props:assrt_type/1`)

`Body` **is a normalized assertion body.** (`assertions_props:nabody/1`)

`Dict` **is a dictionary of variable names.** (`assertions_props:dictionary/1`)

`Flag` **is** `status` **or** `nostatus`**.** (`assrt_write:status_flag/1`)

write_assertion_as_double_comment/7: PREDICATE

Usage:

`write_assertion_as_double_comment(Stream,Goal,Status,Type,Body,Dict,Flag)`

Writes the (normalized) assertion to stream `Stream` **as a Prolog double comment.**

– *Call and exit should be compatible with:*

`Status` **is an acceptable status for an assertion.** (`assertions_props:assrt_status/1`)

`Type` **is an admissible kind of assertion.** (`assertions_props:assrt_type/1`)

`Body` **is a normalized assertion body.** (`assertions_props:nabody/1`)

`Dict` **is a dictionary of variable names.** (`assertions_props:dictionary/1`)

`Flag` **is** `status` **or** `nostatus`**.** (`assrt_write:status_flag/1`)

180.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `format`, `assrt_lib`, `messages`, `assertions_props`, `vndict`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

181 A syntax highlighter

This module implements a syntax highlighter for different source code languages. Currently it depends on external tools like `emacs`.

181.1 Usage and interface

- **Library usage:**
  `:- use_module(library(syntax_highlight)).`
- **Exports:**
  - *Predicates:*
    `can_highlight/1`, `highlight_file_to_html_string/3`, `highlight_string_to_html_string/3`, `highlight_file_to_html_term/3`, `detect_language/2`.
  - *Regular Types:*
    `lang/1`.

181.2 Documentation on exports

lang/1: REGTYPE

Usage: `lang(L)`

`L` is a language for syntax highlight

can_highlight/1: PREDICATE

Usage: `can_highlight(Lang)`

Check if `Lang` is a supported language (silently fails if any dependency for highlighting, e.g., emacs, ciao-mode, is not installed)

- *The following properties should hold at call time:*
  `Lang` is an atom. (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Lang` is a language for syntax highlight (`syntax_highlight:lang/1`)

highlight_file_to_html_string/3: PREDICATE

Usage: `highlight_file_to_html_string(Lang,Input,Output)`

Produce HTML `Output` string with syntax highlight from `Input` file (see `highlight_to_html/3`)

- *Call and exit should be compatible with:*
  `Lang` is a language for syntax highlight (`syntax_highlight:lang/1`)
  `Input` is an atom. (`basic_props:atm/1`)
  `Output` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Lang` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Input` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

highlight_string_to_html_string/3: PREDICATE

Usage: `highlight_string_to_html_string(Lang,Input,Output)`

Produce HTML `Output` string with syntax highlight from `Input` string (see `highlight_to_html/3`)

- *Call and exit should be compatible with:*
  - `Lang` is a language for syntax highlight (`syntax_highlight:lang/1`)
  - `Input` is a string (a list of character codes). (`basic_props:string/1`)
  - `Output` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold at call time:*
  - `Lang` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  - `Input` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

highlight_file_to_html_term/3: PREDICATE

Usage: `highlight_file_to_html_term(Lang,Input,Output)`

Produce HTML `Output` term with syntax highlight from `Input` file (see `highlight_to_html/3`)

- *Call and exit should be compatible with:*
  - `Lang` is a language for syntax highlight (`syntax_highlight:lang/1`)
  - `Input` is an atom. (`basic_props:atm/1`)
  - `Output` is a term representing HTML code. (`html:html_term/1`)
- *The following properties should hold at call time:*
  - `Lang` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  - `Input` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

detect_language/2: PREDICATE

Usage: `detect_language(File,Lang)`

Detect language `Lang` of file `File` (may look at contents)

- *Call and exit should be compatible with:*
  - `File` is an atom. (`basic_props:atm/1`)
  - `Lang` is a language for syntax highlight (`syntax_highlight:lang/1`)
- *The following properties should hold at call time:*
  - `File` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  - `Lang` is a free variable. (`term_typing:var/1`)

181.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `datafacts_rt`, `lists`, `system`, `bundle_paths`, `process`, `emacs_batch`, `system_extra`, `port_reify`, `stream_utils`, `html`, `pathnames`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `doccomments`, `isomodes`, `regtypes`, `fsyntax`, `datafacts`.

182 Operations on source trees

Author(s): Jose F. Morales, The Ciao Development Team.

This module defines predicates to operate on directory trees containing source code (source trees).

The possible operations include enumerating, copying, and cleaning files or selection of files based on filters. See `source_filter/1` for a description of the available filters.

182.1 Details of the walk algorithm

The walk algorithm (see internal `walk/3`) traverses a filtered directory structure and invokes actions (see internal `action_hook/3`, `walk_action/1`) parameterized by walk events (see internal `walk_event/1`) before entering a directory (`enter` event), when a file is processed (`file` event), and once the files in a directory are processed (`exit` event).

Filters restrict the walk to selected parts of the file tree. Filters are decomposed into formulas of basic filters, where a basic filter is a property of a regular file or directory (e.g., based on its name or path).

A normalized filter (see internal `norm_filter/2`) is given by the tuple of formulas (`WalkP`, `DirP`, `FileP`), where each component represents the following conditions for the walk algorithm:

`WalkP` walk into the directory

`DirP` treat directory files

`FileP` treat the file (a regular file or a directory not marked for walk)

Given a base directory, the walk algorithm will perform the following operation on each file `F`:

```
if F is a directory and WalkP(F):
   if DirP(F):
  Action 'enter' on F
  call recursively for each file in F
  Action 'exit' on F
else:
   if FileP(F):
  Action 'file' on F
```

182.2 Efficiency and memory usage

- Evaluation of filter checks is optimized to minimize accesses to the filesystem.
- This code depends on the regular expression library, which is currently not optimized for performance.
- Since file names are encoded as atoms, this code may suffer from exhaustion of the atom table on large file trees.

182.3 Examples

Find cleanable files in the current directory (recursively):

```
?- current_file_find([cleanable(src)], '.', X).
```

Find distributable packages:

```
?- current_file_find([proj(distributable), srctype(package)], '.', X).
```

182.4 Usage and interface

- **Library usage:**

  `:- use_module(library(source_tree)).`
- **Exports:**
  - *Predicates:*

    `current_file_find/3`, `copy_file_tree/4`, `copy_file_tree/5`, `clean_file_tree/2`, `copy_file_or_dir/2`, `remove_dir/1`, `remove_file_or_dir/1`, `delete_glob/2`, `remove_glob/2`, `match_def/3`, `get_file_srctype/2`.
  - *Regular Types:*

    `source_filter/1`, `precomp_level/1`, `file_srctype/1`.

182.5 Documentation on exports

current_file_find/3: PREDICATE

Usage: `current_file_find(Filter,BaseDir,FileName)`

Enumerates recursively all files `FileName` (absolute file name) in `BaseDir` directory files that match the corresponding `Filter`.

- *Call and exit should be compatible with:*

  `Filter` **is a source file filter** (`source_tree:source_filter/1`)

  `BaseDir` **is an atom.** (`basic_props:atm/1`)

  `FileName` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold at call time:*

  `Filter` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `BaseDir` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `FileName` **is a free variable.** (`term_typing:var/1`)

copy_file_tree/4: PREDICATE

Usage: `copy_file_tree(Filter,SrcDir,DestDir,Perms)`

Copy the file tree from `SrcDir` to `DestDir` with permissions `Perms`

copy_file_tree/5: PREDICATE

Usage: `copy_file_tree(Filter,SrcDir,DestDir,Perms,Owner)`

Copy the file tree from `SrcDir` to `DestDir` with permissions `Perms` and owner `Owner`

clean_file_tree/2: PREDICATE

Usage: `clean_file_tree(PrecompLevel,Dir)`

Unwind the precompilation level at directory `Dir`, cleaning the contents recursively (see `untainted` and `cleanable(PrecompLevel)` `source_filter/1`).

copy_file_or_dir/2: PREDICATE

Usage: `copy_file_or_dir(FileName,DestDir)`

Copy `FileName` file into `DestDir` directory. The path for `DestDir` is created if it does not exists. If `FileName` is a directory, all its contents are copied recursively.

remove_dir/1: PREDICATE

Usage: `remove_dir(Dir)`

Delete the directory `Dir` and all its contents recursively. Throws exception if file does not exist.

remove_file_or_dir/1: PREDICATE

Usage: `remove_file_or_dir(FileName)`

Delete `FileName`. If `FileName` is a directory, all its contents are deleted recursively. Ignore errors if file does not exist.

delete_glob/2: PREDICATE

Usage: `delete_glob(Dir,Pattern)`

Delete each file in directory `Dir` (non-recursively) that matches the glob pattern `Pattern`

remove_glob/2: PREDICATE

Usage: `remove_glob(Dir,Pattern)`

Like `delete_glob/2`, but uses `remove_dir/1` for each matching directory in `Dir`.

source_filter/1: REGTYPE

A filter for source files. A filter is a basic filter, an extended filter (defined on terms on other filters), and a list of basic or extended filters.

Basic filters:

`true` Enumerate all files (recursively).

`nonrec` Do not recurse into directories (directories will be treated as files).

`untainted`
Enumerate all files, except those that are backups or hold repository metadata.

`srctype(SrcTypes)`
Enumerate source files of any of the specified `SrcTypes`, a source type or list of source types. A source type (`file_srctype/1`) is any of types`module`, `package`, or `include` files (for user or included files).

`proj(TreeProj)`
Obtain a tree projection based on marks. Projections allow discarding whole subtrees or individual files. The valid projection names and their marks are:

`compilable`
: for each directory, discard the subtree if it contains the `NOCOMPILE` file, or ignore the files whose name match with any of the patterns listed in the file `NOCOMPILEFILES`.

`testable` same with `NOTEST` and `NOTESTFILES` (implies `compilable`).

`distributable`
: same with `NODISTRIBUTE` and `NODISTRIBUTEFILES`.

`installable`
: same with `NOINSTALL` and `NOINSTALLFILES`.

`precomp(PrecompLevel)`
: Enumerate files for the specified precompilation level `PrecompLevel` (see `precomp_level/1`).

`cleanable(PrecompLevel)`
: Enumerate the files resulting from compilation of source files that must be cleaned in order reduce the precompilation to `PrecompLevel`. E.g., `PrecompLevel=src` cleans all compiler output, `PrecompLevel=noa` cleans only platform dependant files.

Extended filters:

`compilable_module`
: Enumerate Ciao modules that are suitable for automatic compilation during bundle build (in the untainted `proj(compilable)` projection).

`testable_module`
: Enumerate Ciao modules that can be compiled and tested (in the untainted `proj(compilable)` and `proj(testable)` projection).

`distributable_precomp(PrecompLevel)`
: Enumerate files that can be distributed (in the untainted `proj(distributable)` projection and for the given precompilation level `PrecompLevel`).

  Additionally, excludes the build directory and some temporary files.

`installable_precomp(PrecompLevel)`
: Enumerate (source) files that can be installed (in the untainted `proj(installable)` projection and for the given precompilation level `PrecompLevel`).

  Additionally, excludes the build directory and some temporary files.

Usage: `source_filter(Filter)`

`Filter` is a source file filter

precomp_level/1: REGTYPE

The valid precompilation levels are:

`src` source files only

`noa` sources and portable objects

`bin` sources, portable, and architecture dependant objects (except special third-party files like elisp objects and Java classes)

`full` like `bin`, including special third-party files.

Usage: `precomp_level(PrecompLevel)`

`PrecompLevel` **is a pre-compilation level**

match_def/3: PREDICATE

No further documentation available for this predicate.

file_srctype/1: REGTYPE

Source file types, defined as:

```
file_srctype(module).
file_srctype(package).
file_srctype(include).
```

Usage: `file_srctype(T)`

`T` **is a source file type**

get_file_srctype/2: PREDICATE

This operation should be relatively fast, since we only need to read the first term in the file. Modules start with a `:- module/3` (or `:- module/2`) or `:- package/1` directive.

Usage: `get_file_srctype(FileName,Type)`

`FileName` **is a file of type** `Type` **(**`module`**,** `package`**, or** `include` **for included or user sources). Fails if the file does not seem Ciao code**

- *Call and exit should be compatible with:*

  `FileName` **is an atom.** (`basic_props:atm/1`)

  `Type` **is a source file type** (`source_tree:file_srctype/1`)
- *The following properties should hold at call time:*

  `FileName` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

182.6 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `system`, `pathnames`, `lists`, `sort`, `aggregates`, `regexp_code`, `system_extra`, `bundle_paths`, `stream_utils`, `read`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `fsyntax`, `hiord`, `regtypes`, `isomodes`.

183 Version string parsing and comparison

Author(s): Jose F. Morales.

This module contains operations to work with version strings (as atoms). The specification of the format and operations is partially inspired on Semantic Versioning (`http://semver.org/`).

Normal versions are specified as strings of the form `X.Y.Z`, where `X`, `Y`, and `Z` are non-negative integers. The components denote the major, minor, and patch version, respectively. Missing version numbers are assumed to be 0.

Pre-release versions include an optional *pre-release* identifier (`X.Y.Z-Prerelease`), which itself may contain many dot separated components.

Versions are compared by comparing each component from left to right. Each component is compared numerically (if numbers) or lexicographically (if not numbers). Numeric components have lower precedence than non-numeric components. A pre-release version has lower precedence than its corresponding normal version.

183.1 Usage and interface

- **Library usage:**
  `:- use_module(library(version_strings)).`
- **Exports:**
  - *Predicates:*
    `version_parse/4`, `version_split_patch/3`, `version_compare/3`.

183.2 Documentation on exports

version_parse/4: PREDICATE

Usage: `version_parse(VerAtm,Major,Minor,Patch)`

Parse a version string into numeric values for major, minor, and patch numbers

version_split_patch/3: PREDICATE

Usage: `version_split_patch(VerAtm,VerNopatchAtm,PatchAtm)`

Split version `VerAtm` into major and minor `VerNopatchAtm`, and patch and prerelease `PatchAtm`)

version_compare/3: PREDICATE

Usage: `version_compare(C,A,B)`

Compare versions `A` and `B` (see `compare/3` for values of `C`)

183.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `lists`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

184 The Ciao library browser

Author(s): Angel Fernandez Pineda, Isabel Garcia-Contreras, Jose F. Morales (minor).

The `librowser` library provides a set of predicates wich enable the user to interactively find Ciao libraries and/or any predicate exported by them.

This is a simple example:

```
?- apropos(aggregates:'.*find.*').

aggregates:findnsols/5
aggregates:findnsols/4
aggregates:findall/4
aggregates:findall/3

yes
?-
```

`librowser` is specially useful when inside GNU Emacs: just place the cursor over a librowser response and press C-cTAB in order to get help on the related predicate. Refer to the "Using Ciao inside GNU Emacs" chapter for further information.

184.1 Usage and interface

- **Library usage:**

  It is not necesary to use this library at user programs. It is designed to be used at the Ciao *toplevel* shell: `ciaosh`. In order to do so, just make use of `use_module/1` as follows:

  `use_module(library(librowser)).`

  Then, the library interface must be read. This is automatically done when calling any predicate at librowser, and the entire process will take a little moment.So, you should want to perform such a process after loading the Ciao toplevel:

  ```
  ?- use_module(library(librowser)).

  yes
  ?- update.
  ```

  Whether you want this process to be automatically performed when loading `ciaosh`, you may include those lines in your *.ciaorc* personal initialization file.

- **Exports:**
  - *Predicates:*

    `update/0`, `browse/2`, `where/1`, `describe/1`, `apropos/1`.

184.2 Documentation on exports

update/0: PREDICATE

This predicate will scan the Ciao libraries for predicate definitions. This may be done once time before calling any other predicate at this library.

`update/0` will also be automatically called (once) when calling any other predicate at librowser.

Usage:

Creates an internal database of modules at Ciao libraries.

browse/2: PREDICATE

This predicate is fully reversible, and is provided to inspect concrete predicate specifications. For example:

```
?- browse(M,findall/A).

A = 3,
M = conc_aggregates ? ;

A = 4,
M = aggregates ? ;

A = 3,
M = aggregates ? ;

no
?-
```

Usage: `browse(Module,Spec)`

Asocciates the given `Spec` **predicate specification with the** `Module` **which exports it.**

– *The following properties should hold at call time:*

`Module` **is a module name (an atom)** (`librowser:module_name/1`)

`Spec` **is a Functor/Arity predicate specification** (`librowser:pred_spec/1`)

where/1: PREDICATE

This predicate will print at the screen the module needed in order to import a given predicate specification. For example:

```
?- where(findall/A).
findall/3 exported at module conc_aggregates
findall/4 exported at module aggregates
findall/3 exported at module aggregates

yes
?-
```

Usage: `where(Spec)`

Display what module to load in order to import the given `Spec`.

– *The following properties should hold at call time:*

`Spec` **is a Functor/Arity predicate specification** (`librowser:pred_spec/1`)

describe/1: PREDICATE

This one is used to find out which predicates were exported by a given module. Very useful when you know the library, but not the concrete predicate. For example:

```
?- describe(librowser).
Predicates at library librowser :

apropos/1
describe/1
where/1
browse/2
update/0

yes
?-
```

Usage: `describe(Module)`

Display a list of exported predicates at the given `Module`

– *The following properties should hold at call time:*

`Module` **is a module name (an atom)** (`librowser:module_name/1`)

apropos/1: PREDICATE

This tool makes use of regular expresions in order to find predicate specifications. It is very useful whether you can't remember the full name of a predicate. Regular expresions take the same format as described in library `patterns`**. Example:**

```
?- apropos('write.').

write:writeq/1
write:writeq/2

yes
?- apropos('write.*'/2).

dht_misc:write_pr/2
profiler_auto_conf:write_cc_assertions/2
mtree:write_mforest/2
transaction_concurrency:write_lock/2
transaction_logging:write/2
provrml_io:write_vrml_file/2
provrml_io:write_terms_file/2
unittest_base:write_data/2
write:write_canonical/2
write:writeq/2
write:write/2
write:write_term/2
stream_utils:write_string/2
res_exectime_hlm_gen:write_hlm_indep_each/2
res_exectime_hlm_gen:write_hlm_indep_2/2
res_exectime_hlm_gen:write_hlm_dep/2
oracle_calibration:write_conf/2
bshare_utils:write_string/2
bshare_utils:write_string_list/2
bshare_utils:write_length/2
bshare_utils:write_neg_db_stream/2
```

```
bshare_utils:write_neg_db/2
bshare_utils:write_pos_db/2

yes
```

When no predicates are found with the exact search, this predicate will perform a fuzzy search which will find predicates at a distance of one edit, swap, deletion or insertion.

```
?- apropos('wirte').
Predicate wirte not found. Similar predicates:

transaction_logging:write/2
write:write/1
write:write/2

yes
?- apropos(apend).
Predicate apend not found. Similar predicates:

hprolog:append/2
lists:append/3
llists:append/2

yes
?-
```

Usage: `apropos(RegSpec)`

This will search any predicate specification `Spec` which matches the given `RegSpec` incomplete predicate specification.

– *The following properties should hold at call time:*

  `RegSpec` is a **predicate specification** `Pattern`, `Pattern/Arity`, `Module:Pattern`, `Module:Pattern/Arity`. (`librowser:apropos_spec/1`)

184.3 Documentation on internals

apropos_spec/1: REGTYPE

Defined as:

```
apropos_spec(Pattern) :-
    atm(Pattern).
apropos_spec(Pattern/Arity) :-
    atm(Pattern),
    int(Arity).
apropos_spec(Module:Pattern/Arity) :-
    atm(Pattern),
    atm(Module),
    int(Arity).
apropos_spec(Module:Pattern) :-
    atm(Pattern),
    atm(Module).
```

Usage: `apropos_spec(S)`

S **is a predicate specification** Pattern, Pattern/Arity, Module:Pattern, Module:Pattern/Arity.

184.4 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

  regexp_code, datafacts_rt, read, fastrw, system, stream_utils, lists, fuzzy_search, pathnames, bundle_paths, write.

– *Packages:*

  prelude, initial, condcomp, assertions, assertions/assertions_basic, regexp, datafacts, dcg.

185 Random numbers

Author(s): Daniel Cabeza.

This module provides predicates for generating pseudo-random numbers

185.1 Usage and interface

- **Library usage:**
  `:- use_module(library(random)).`
- **Exports:**
  - *Predicates:*
    `random/1`, `random/3`, `srandom/1`.

185.2 Documentation on exports

random/1: PREDICATE

`random(Number)`

`Number` **is a (pseudo-) random number in the range [0.0,1.0]**

Usage:

- *The following properties should hold at call time:*
  `Number` **is a free variable.** (`term_typing:var/1`)
- *The following properties hold upon exit:*
  `Number` **is currently instantiated to a float.** (`term_typing:float/1`)
- *The following properties hold globally:*
  The Prolog predicate `PrologName` **is implemented using the function** `ForeignName`. **The same considerations as above example are to be applied.** (`foreign_interface_properties:foreign_low/2`)

random/3: PREDICATE

`random(Low,Up,Number)`

`Number` **is a (pseudo-) random number in the range [**`Low`, `Up`**]**

Usage 1:

If `Low` **and** `Up` **are integers,** `Number` **is an integer.**

- *The following properties should hold at call time:*
  `Low` **is an integer.** (`basic_props:int/1`)
  `Up` **is an integer.** (`basic_props:int/1`)
  `Number` **is a free variable.** (`term_typing:var/1`)
- *The following properties hold upon exit:*
  `Number` **is an integer.** (`basic_props:int/1`)
- *The following properties hold globally:*
  The Prolog predicate `PrologName` **is implemented using the function** `ForeignName`. **The same considerations as above example are to be applied.** (`foreign_interface_properties:foreign_low/2`)

Usage 2:

- *The following properties should hold at call time:*

  `Low` **is a float.** (`basic_props:flt/1`)

  `Up` **is a number.** (`basic_props:num/1`)

  `Number` **is a free variable.** (`term_typing:var/1`)

- *The following properties hold upon exit:*

  `Number` **is a float.** (`basic_props:flt/1`)

Usage 3:

- *The following properties should hold at call time:*

  `Low` **is an integer.** (`basic_props:int/1`)

  `Up` **is a float.** (`basic_props:flt/1`)

  `Number` **is a free variable.** (`term_typing:var/1`)

- *The following properties hold upon exit:*

  `Number` **is a float.** (`basic_props:flt/1`)

srandom/1: PREDICATE

`srandom(Seed)`

Changes the sequence of pseudo-random numbers according to `Seed`**. The starting sequence of numbers generated can be duplicated by calling the predicate with** `Seed` **unbound (the sequence depends on the OS).**

Usage:

- *Calls should, and exit will be compatible with:*

  `Seed` **is an integer.** (`basic_props:int/1`)

- *The following properties hold upon exit:*

  `Seed` **is an integer.** (`basic_props:int/1`)

- *The following properties hold globally:*

  The Prolog predicate `PrologName` **is implemented using the function** `ForeignName`**. The same considerations as above example are to be applied.** (`foreign_interface_properties:foreign_low/2`)

185.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `foreign_interface_properties`.

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `foreign_interface`, `basicmodes`, `regtypes`, `engine(foreign_types)`.

186 Randomized aggregates

Author(s): Jose F. Morales.

This module implements randomized version of aggregates predicates.

Example 1:

```
?- random_findall(4, X, between(1,10000,X), Xs).

Xs = [6659,7135,7871,9864] ?

yes
```

Example 2:

```
?- random_findall(4, X, member(X, [the,sequel,will,not,happen]), Xs).

Xs = [the,sequel,will,happen] ?

yes
```

186.1 Usage and interface

- **Library usage:**
  `:- use_module(library(random_aggregates)).`
- **Exports:**
  - *Predicates:*
    `random_findall/4`.

186.2 Documentation on exports

random_findall/4: PREDICATE

`random_findall(K,X,Goal,Ys)`

This predicate implements the reservoir sampling algorithm (`https://en.wikipedia.org/wiki/Reservoir_sampling`). It needs to evaluate all solutions to `Goal` but only need to keeps `K` of those solutions in memory simultaneously.

Usage: `random_findall(K,X,Goal,Ys)`

Obtains a list `Ys` of length `K` of random solutions uniformly distributed from `X` for all solutions to `Goal`. Fails if there are less than `K` solutions.

- *The following properties should hold at call time:*
  `K` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `X` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Goal` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

Meta-predicate **with arguments:** `random_findall(?,?,goal,?)`.

186.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  native_props, datafacts_rt, aggregates, random.
- *Packages:*

  prelude, initial, condcomp, assertions, assertions/assertions_basic, isomodes, nativeprops, hiord, datafacts.

187 The socket interface

Author(s): Manuel Carro, Daniel Cabeza.

This module defines primitives to open sockets, send, and receive data from them. This allows communicating with other processes, on the same machine or across the Internet. The reader should also consult standard bibliography on the topic for a proper use of these primitives.

187.1 Usage and interface

- **Library usage:**
  `:- use_module(library(sockets)).`
- **Exports:**
  - *Predicates:*
    `connect_to_socket_type/4`, `connect_to_socket/3`, `bind_socket/3`, `socket_accept/2`, `select_socket/5`, `socket_send/3`, `socket_sendall/2`, `socket_send_stream/2`, `socket_recv/3`, `socket_shutdown/2`, `hostname_address/2`, `socket_getpeername/2`.
  - *Regular Types:*
    `socket_type/1`, `shutdown_type/1`.

187.2 Documentation on exports

connect_to_socket_type/4: PREDICATE

Usage: `connect_to_socket_type(Hostname,Port,Type,Stream)`

Returns a `Stream` which connects to `Hostname`. The `Type` of connection can be defined. A `Stream` is returned, which can be used to `write/2` to, to `read/2`, to `socket_send/3` to, or to `socket_recv/3` from the socket.

- *Calls should, and exit will be compatible with:*
  `Hostname` is currently instantiated to an atom. (`term_typing:atom/1`)
  `Port` is an integer. (`basic_props:int/1`)
  `Type` is a valid socket type. (`sockets:socket_type/1`)
  `Stream` is an open stream. (`stream_basic:stream/1`)
- *The following properties should hold at call time:*
  `Hostname` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Port` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Type` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Stream` is a free variable. (`term_typing:var/1`)
- *The following properties hold globally:*
  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

connect_to_socket/3: PREDICATE

Usage: `connect_to_socket(Hostname,Port,Stream)`

Calls `connect_to_socket_type/4` with SOCK_STREAM connection type. This is the connection type you want in order to use the `write/2` and `read/2` predicates (and other stream IO related predicates).

- *Call and exit should be compatible with:*

  `Hostname` **is an atom.** (`basic_props:atm/1`)

  `Port` **is an integer.** (`basic_props:int/1`)

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

- *The following properties should hold at call time:*

  `Hostname` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Port` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Stream` **is a free variable.** (`term_typing:var/1`)

bind_socket/3: PREDICATE

Usage: `bind_socket(Port,Length,Socket)`

Returns an AF_INET `Socket` bound to `Port` (which may be assigned by the OS or defined by the caller), and listens to it (hence no listen call in this set of primitives). `Length` specifies the maximum number of pending connections.

- *Calls should, and exit will be compatible with:*

  `Port` **is an integer.** (`basic_props:int/1`)

  `Length` **is an integer.** (`basic_props:int/1`)

  `Socket` **is an integer.** (`basic_props:int/1`)

- *The following properties should hold at call time:*

  `Length` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Socket` **is a free variable.** (`term_typing:var/1`)

- *The following properties hold globally:*

  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

socket_accept/2: PREDICATE

Usage: `socket_accept(Sock,Stream)`

Creates a new `Stream` connected to `Sock`.

- *Calls should, and exit will be compatible with:*

  `Sock` **is an integer.** (`basic_props:int/1`)

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

- *The following properties should hold at call time:*

  `Sock` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Stream` **is a free variable.** (`term_typing:var/1`)

- *The following properties hold globally:*

  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

select_socket/5: PREDICATE

Usage: `select_socket(Socket,NewStream,TO_ms,Streams,ReadStreams)`

Wait for data available in a list of `Streams` and in a `Socket`. `Streams` is a list of Prolog streams which will be tested for reading. `Socket` is a socket (i.e., an integer denoting the O.S. port number) or a free variable. `TO_ms` is a number denoting a timeout. Within this timeout the `Streams` and the `Socket` are checked for the availability of data to be read. `ReadStreams` is the list of streams belonging to `Streams` which have data pending to be read. If `Socket` was a free variable, it is ignored, and `NewStream` is not checked. If `Socket` was instantiated to a port number and there are connections pending, a connection is accepted and connected with the Prolog stream in `NewStream`.

- *Calls should, and exit will be compatible with:*

  `Socket` **is an integer.** (`basic_props:int/1`)

  `NewStream` **is an open stream.** (`stream_basic:stream/1`)

  `TO_ms` **is an integer.** (`basic_props:int/1`)

  `Streams` **is a list of** `streams`**.** (`basic_props:list/2`)

  `ReadStreams` **is a list of** `streams`**.** (`basic_props:list/2`)

- *The following properties should hold at call time:*

  `Socket` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `NewStream` **is a free variable.** (`term_typing:var/1`)

  `TO_ms` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Streams` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `ReadStreams` **is a free variable.** (`term_typing:var/1`)

- *The following properties hold globally:*

  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

socket_send/3: PREDICATE

Usage: `socket_send(Stream,Bytes,Sent)`

Sends `Bytes` to the socket associated to `Stream`, return in `Sent` the number of sent bytes. The socket has to be in connected state.

- *Calls should, and exit will be compatible with:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

  `Bytes` **is list of bytes.** (`basic_props:bytelist/1`)

  `Sent` **is an integer.** (`basic_props:int/1`)

- *The following properties should hold at call time:*

  `Stream` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Bytes` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

- *The following properties hold globally:*

  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

socket_sendall/2: PREDICATE

Usage: `socket_sendall(Stream,Bytes)`

Sends all `Bytes` to the socket associated to `Stream`. The socket has to be in connected state.

- *Calls should, and exit will be compatible with:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
  `Bytes` **is list of bytes.** (`basic_props:bytelist/1`)
- *The following properties should hold at call time:*
  `Stream` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `Bytes` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties hold globally:*
  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

socket_send_stream/2: PREDICATE

Usage: `socket_send_stream(Stream,FromStream)`

Sends all bytes from stream `FromStream` to the socket associated to `Stream`. The socket has to be in connected state. `FromStream` cannot be a socket stream

- *Calls should, and exit will be compatible with:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
  `FromStream` **is an open stream.** (`stream_basic:stream/1`)
- *The following properties should hold at call time:*
  `Stream` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
  `FromStream` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties hold globally:*
  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

socket_recv/3: PREDICATE

Usage: `socket_recv(Stream,Bytes,Length)`

Receives a byte list `Bytes` from the socket associated to `Stream`, and returns its `Length`. For TCP, `Length` is 0 if the peer has performed an orderly shutdown on the socket.

- *Calls should, and exit will be compatible with:*
  `Stream` **is an open stream.** (`stream_basic:stream/1`)
  `Bytes` **is list of bytes.** (`basic_props:bytelist/1`)
  `Length` **is an integer.** (`basic_props:int/1`)
- *The following properties should hold at call time:*
  `Stream` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties hold globally:*
  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

socket_shutdown/2: PREDICATE

Usage: `socket_shutdown(Stream,How)`

Shut down a duplex communication socket with which `Stream` is associated. All or part of the communication can be shutdown, depending on the value of `How`. The atoms `read`, `write`, or `read_write` should be used to denote the type of closing required.

- *Calls should, and exit will be compatible with:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

  `How` **is a valid shutdown type.** (`sockets:shutdown_type/1`)
- *The following properties should hold at call time:*

  `Stream` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `How` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties hold globally:*

  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

hostname_address/2: PREDICATE

Usage: `hostname_address(Hostname,Address)`

`Address` **is unified with the atom representing the address (in AF_INET format) corresponding to `Hostname`.**

- *Calls should, and exit will be compatible with:*

  `Hostname` **is an atom.** (`basic_props:atm/1`)

  `Address` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold at call time:*

  `Hostname` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties hold globally:*

  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

socket_getpeername/2: PREDICATE

Usage: `socket_getpeername(Stream,Address)`

`Address` **is unified with the atom representing the address (in AF_INET or AF_INET6 format) of the peer connected to the socket associated to `Stream`.**

- *Calls should, and exit will be compatible with:*

  `Stream` **is an open stream.** (`stream_basic:stream/1`)

  `Address` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold at call time:*

  `Stream` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)
- *The following properties hold globally:*

  The Prolog predicate `PrologName` is implemented using the function `ForeignName`. The same considerations as above example are to be applied. (`foreign_interface_properties:foreign_low/2`)

socket_type/1: REGTYPE

Defines the atoms which can be used to specify the socket type recognized by `connect_to_socket_type/4`. Defined as follows:

```
socket_type(stream).
socket_type(dgram).
socket_type(raw).
socket_type(seqpacket).
socket_type(rdm).
```

Usage: `socket_type(T)`

`T` is a valid socket type.

shutdown_type/1: REGTYPE

Usage: `shutdown_type(T)`

`T` is a valid shutdown type.

187.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `foreign_interface_properties`.

- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `regtypes`, `foreign_interface`, `basicmodes`, `engine(foreign_types)`.

188 Sockets I/O

Author(s): Francisco Bueno, Jose F. Morales (minor changes, queue version).

This module implements a socket-based message reading loop.

188.1 Usage and interface

- **Library usage:**
  `:- use_module(library(sockets/sockets_io)).`
- **Exports:**
  - *Predicates:*
    `serve_socket/3`, `socket_send_term/2`, `socket_recv_term/2`, `socket_send_fastrw/2`, `socket_recv_fastrw/2`.

188.2 Documentation on exports

serve_socket/3: PREDICATE

Usage: `serve_socket(Socket,Serve,Handler)`

Handles the streams associated to `Socket` calling `Serve` on one request of each stream (as `Serve(Stream,Unwatch)`), and `Handler(Stream)` if the stream is empty (connection closed). It is expected that `Unwatch` is unified with `yes` or `no` after calls to `Serve`. If `Unwatch=yes` then the `Stream` is removed from the watch list (e.g., useful when the application will perform IO on that stream from a separate thread)

- *Call and exit should be compatible with:*
  `Socket` is a socket id. (`sockets_io:socket/1`)
  `Serve` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)
  `Handler` is a term which represents a goal, i.e., an atom or a structure. (`basic_props:cgoal/1`)

Meta-predicate with arguments: `serve_socket(?,pred(2),pred(1))`.

socket_send_term/2: PREDICATE

Usage: `socket_send_term(Stream,Term)`

Writes `Term` to `Stream` in a way that it is safe for a socket connection on `Stream`.

- *Call and exit should be compatible with:*
  `Stream` is an open stream. (`stream_basic:stream/1`)
  `Term` is any term. (`basic_props:term/1`)

socket_recv_term/2: PREDICATE

No further documentation available for this predicate.

socket_send_fastrw/2: PREDICATE

No further documentation available for this predicate.

socket_recv_fastrw/2: PREDICATE

No further documentation available for this predicate.

188.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `lists`, `sockets`, `terms_io`, `read`, `fastrw`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `hiord`, `regtypes`.

189 HTTP client/server libraries

Author(s): The Ciao Development Team.

This is a collection of modules that implement a client and server for the HTTP protocol, and related technology like CGI programming.

Note: Parts of this code are based on the original PiLLoW library [CHV96]), which can be downloaded from `http://clip.dia.fi.upm.es/Software/pillow/pillow.html`**.**

189.1 Usage and interface

- **Library usage:**
  `:- use_package(http).`
  or
  `:- module(...,...,[http]).`
- **Implicit imports:**
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `doccomments`.

190 HTTP client

Author(s): Daniel Cabeza, Jose F. Morales (fixes, improvements).

This module implements a client the HTTP protocol.

190.1 Usage and interface

- **Library usage:**
  `:- use_module(library(http/http_client)).`
- **Exports:**
  - *Predicates:*
    `fetch_url/3`.

190.2 Documentation on exports

fetch_url/3: PREDICATE

`fetch_url(URL,Request,Response)`

Fetches the document pointed to by `URL` from Internet, using request parameters `Request` (`get` method by default), and unifies `Response` with the parameters of the response. Fails on timeout. Note that redirections are not handled automatically, that is, if `Response` contains terms of the form `status(redirection,301,_)` and `location(NewURL)`, the program should in most cases access location `NewURL`.

Usage: `fetch_url(URL,Request,Response)`

- *The following properties should hold at call time:*
  `URL` **specifies a URL.** (`url:url_term/1`)
  `Request` **is a list of** `http_request_params`**.** (`basic_props:list/2`)
- *The following properties should hold upon exit:*
  `Response` **is a list of** `http_response_params`**.** (`basic_props:list/2`)

http_request_param/1: (UNDOC_REEXPORT)

Imported from `http_messages` **(see the corresponding documentation for details).**

http_request_param/1: (UNDOC_REEXPORT)

Imported from `http_messages` **(see the corresponding documentation for details).**

http_response_param/1: (UNDOC_REEXPORT)

Imported from `http_messages` **(see the corresponding documentation for details).**

http_response_param/1: (UNDOC_REEXPORT)

Imported from `http_messages` **(see the corresponding documentation for details).**

190.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `lists`, `url`, `http_messages`, `sockets`, `stream_utils`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `regtypes`, `dcg`, `hiord`, `doccomments`.

191 HTTP server

Author(s): The Ciao Development Team, Jose F. Morales (multifile-based HTTP simple server).

This module implements a simple HTTP server.

It can be used to handle individual HTTP requests (`http_serve_fetch/2`**) or for implementing a simple HTTP server (see** `http_bind/1`, `http_loop/1`, `http_shutdown/1`**).**

Clients of this module must use the `http_server_hooks` **package and implement the multifile** `httpserv.handle/3` **(see** `http_loop/1` **for details).**

191.1 Usage and interface

- **Library usage:**

  `:- use_module(library(http/http_server)).`
- **Exports:**
  - *Predicates:*

    `http_serve_fetch/2`, `http_bind/1`, `http_loop/1`, `http_shutdown/1`, `http_protect/4`.
  - *Multifiles:*

    `httpserv.handle/3`, `httpserv.file_path/2`.

191.2 Documentation on exports

http_serve_fetch/2: PREDICATE

Usage: `http_serve_fetch(Stream,Serve)`

Read a HTTP request from `Stream`**, obtain the response calling** `Serve` **predicate, and write the response to the socket stream.**

Meta-predicate **with arguments:** `http_serve_fetch(?,pred(2))`.

http_bind/1: PREDICATE

Usage: `http_bind(Port)`

Bind socket to the port `Port` **(use with** `http_loop/1`**)**

http_loop/1: PREDICATE

Usage: `http_loop(ExitCode)`

Listen and handle HTTP requests in a loop. You can terminate this loop by `http_shutdown/1` **predicate. Requests are handled by multifile** `httpserv.handle/3` **and** `httpserv.file_path/2` **predicates (declared in** `http_server_hooks` **file).**

http_shutdown/1: PREDICATE

Usage: `http_shutdown(ExitCode)`

`ExitCode` **mark that we are not going to process further requests**

http_protect/4: PREDICATE

Usage: `http_protect(Handler,ErrHandler,Request,Response)`

Execute `Handler` on `Request` to obtain `Response`. If `Handler` fails or raises some exception `E`, a response is obtained calling `ErrHandler` on `E`. This is useful, for example, to produce an HTML page informing about the incident

Meta-predicate with arguments: `http_protect(pred(2),pred(2),?,?)`.

191.3 Documentation on multifiles

httpserv.handle/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

httpserv.file_path/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

191.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `datafacts_rt`, `lists`, `port_reify`, `pathnames`, `http_messages`, `sockets`, `stream_utils`, `sockets_io`, `system`, `terms`, `mimetypes`, `write`, `format`, `http_date`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `dcg`, `hiord`, `doccomments`, `datafacts`.

192 URL encoding/decoding

Author(s): The Ciao Development Team.

This module implements URL encoding/decoding predicates.

192.1 Usage and interface

- **Library usage:**
  `:- use_module(library(http/url)).`
- **Exports:**
  - *Predicates:*
    `url_info/2`, `url_info_relative/3`.
  - *Regular Types:*
    `url_term/1`.

192.2 Documentation on exports

url_term/1: REGTYPE

A term specifying an Internet Uniform Resource Locator. Currently only HTTP URLs are supported. Example: `http('ciao-lang.org',80,"/ciao/")`. **Defined as**

```
url_term(http(Hostname,Port,URIStr)) :-
    atm(Hostname),
    int(Port),
    string(URIStr).
```

Usage: `url_term(URL)`

`URL` **specifies a URL.**

url_info/2: PREDICATE

`url_info(URL,URLTerm)`

Translates a URL `URL` **to a Prolog structure** `URLTerm` **which details its various components, and vice-versa. For now non-HTTP URLs make the predicate fail.**

Usage 1:

- *Call and exit should be compatible with:*
  `URLTerm` **specifies a URL.** (`url:url_term/1`)
- *The following properties should hold at call time:*
  `URL` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `URLTerm` **specifies a URL.** (`url:url_term/1`)

Usage 2:

- *Call and exit should be compatible with:*
  `URLTerm` **specifies a URL.** (`url:url_term/1`)

- *The following properties should hold at call time:*
  URL **is a string (a list of character codes).** (basic_props:string/1)
- *The following properties should hold upon exit:*
  URLTerm **specifies a URL.** (url:url_term/1)

Usage 3:
- *The following properties should hold at call time:*
  URL **is a free variable.** (term_typing:var/1)
  URLTerm **specifies a URL.** (url:url_term/1)
- *The following properties should hold upon exit:*
  URL **is a string (a list of character codes).** (basic_props:string/1)

url_info_relative/3: PREDICATE

`url_info_relative(URL,BaseURLTerm,URLTerm)`

Translates a relative URL URL **which appears in the HTML page refered to by** BaseURLTerm **into** URLTerm**, a Prolog structure containing its absolute parameters. Absolute URLs are translated as with** url_info/2**. E.g.**

```
url_info_relative("dadu.html",
                  http('www.foo.com',80,"/bar/scoob.html"), Info)
```

gives `Info = http('www.foo.com',80,"/bar/dadu.html")`.

Usage 1:
- *Call and exit should be compatible with:*
  URLTerm **specifies a URL.** (url:url_term/1)
- *The following properties should hold at call time:*
  URL **is an atom.** (basic_props:atm/1)
  BaseURLTerm **specifies a URL.** (url:url_term/1)
- *The following properties should hold upon exit:*
  URLTerm **specifies a URL.** (url:url_term/1)

Usage 2:
- *Call and exit should be compatible with:*
  URLTerm **specifies a URL.** (url:url_term/1)
- *The following properties should hold at call time:*
  URL **is a string (a list of character codes).** (basic_props:string/1)
  BaseURLTerm **specifies a URL.** (url:url_term/1)
- *The following properties should hold upon exit:*
  URLTerm **specifies a URL.** (url:url_term/1)

192.3 Documentation on imports

This module has the following direct dependencies:
- *System library modules:*
  lists, http_grammar.
- *Packages:*
  prelude, initial, condcomp, assertions, assertions/assertions_basic, isomodes, regtypes, dcg, doccomments.

193 Form Data and Query Strings

Author(s): The Ciao Development Team.

This module implements a printer/parser for query strings and multipart/form-data contents of HTTP requests.

Query strings are typically encoded in the URLs of `GET` messages. Form-data is encoded in the contents of `POST` messages. Both are useful to send list of name=value information in HTTP requests (originally from HTML forms, lately for many other uses).

193.1 Usage and interface

- **Library usage:**

  `:- use_module(library(http/http_forms)).`
- **Exports:**
  - *Predicates:*

    `get_form_value/3`, `get_form_value_string/3`, `get_form_value_atm/3`, `form_empty_value/1`, `form_default/3`, `url_query_values/2`, `form_encode_value/3`, `form_decode_value/3`, `lines_to_value/2`, `http_parse_form/2`.
  - *Regular Types:*

    `form_dict/1`, `form_assignment/1`, `form_value/1`, `value_dict/1`.
  - *Multifiles:*

    `define_flag/3`.

193.2 Documentation on exports

form_dict/1: REGTYPE

Usage: `form_dict(Dict)`

`Dict` **is a dictionary of values of the attributes of a form. It is a list of** `form_assignment`

form_assignment/1: REGTYPE

Usage: `form_assignment(Eq)`

`Eq` **is an assignment of value of an attribute of a form. It is defined by:**

```
form_assignment(A=V) :-
    atm(A),
    form_value(V).
form_value(A) :-
    atm(A).
form_value(N) :-
    num(N).
form_value(L) :-
    list(string,L).
```

form_value/1: REGTYPE

Usage: `form_value(V)`

`V` **is a value of an attribute of a form.**

value_dict/1: REGTYPE

Usage: `value_dict(Dict)`

`Dict` **is a dictionary of values. It is a list of pairs** *atom=constant***.**

get_form_value/3: PREDICATE

`get_form_value(Dict,Var,Val)`

Unifies `Val` **with the value for attribute** `Var` **in dictionary** `Dict`**. Does not fail: value is** `''` **if not found (this simplifies the programming of form handlers when they can be accessed directly).**

Usage:

- *Call and exit should be compatible with:*
  `Val` **is a value of an attribute of a form.** (`http_forms:form_value/1`)
- *The following properties should hold at call time:*
  `Dict` **is a dictionary of values of the attributes of a form. It is a list of** `form_assignment` (`http_forms:form_dict/1`)
  `Var` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Val` **is a value of an attribute of a form.** (`http_forms:form_value/1`)

get_form_value_string/3: PREDICATE

`get_form_value_string(Dict,Var,Val)`

Like `get_form_value/3` **but obtain** `Val` **as a string.**

Usage:

- *Call and exit should be compatible with:*
  `Val` **is a string (a list of character codes).** (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Dict` **is a dictionary of values of the attributes of a form. It is a list of** `form_assignment` (`http_forms:form_dict/1`)
  `Var` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*
  `Val` **is a string (a list of character codes).** (`basic_props:string/1`)

get_form_value_atm/3: PREDICATE

`get_form_value_atm(Dict,Var,Val)`

Like `get_form_value/3` **but obtain** `Val` **as an atom.**

Usage:

- *Call and exit should be compatible with:*
  `Val` **is an atom.** (`basic_props:atm/1`)

- *The following properties should hold at call time:*

  `Dict` **is a dictionary of values of the attributes of a form. It is a list of** `form_assignment` (`http_forms:form_dict/1`)

  `Var` **is an atom.** (`basic_props:atm/1`)
- *The following properties should hold upon exit:*

  `Val` **is an atom.** (`basic_props:atm/1`)

form_empty_value/1: PREDICATE

Usage: `form_empty_value(Term)`

Checks that `Term`**, a value comming from a text area is empty (can have spaces, newlines and linefeeds).**

form_default/3: PREDICATE

Usage: `form_default(Val,Default,NewVal)`

Useful when a form is only partially filled, or when the executable can be invoked either by a link or by a form, to set form defaults. If the value of `Val` **is empty then** `NewVal=Default`**, else** `NewVal=Val`**.**

- *The following properties should hold at call time:*

  `Val` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `Default` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

  `NewVal` **is a free variable.** (`term_typing:var/1`)

url_query_values/2: PREDICATE

`url_query_values(URLArgs,Dict)`

`Dict` **is a dictionary of parameter values and** `URLArgs` **is the URL-encoded string of those assignments, which may appear after an URL pointing to a CGI script preceded by a '?'.** `Dict` **is computed according to the** `raw_form_values` **flag. The use of this predicate is reversible.**

Usage 1:

- *The following properties should hold at call time:*

  `URLArgs` **is a free variable.** (`term_typing:var/1`)

  `Dict` **is a dictionary of values. It is a list of pairs** *atom=constant***.** (`http_forms:value_dict/1`)
- *The following properties should hold upon exit:*

  `URLArgs` **is a string (a list of character codes).** (`basic_props:string/1`)

Usage 2:

- *The following properties should hold at call time:*

  `URLArgs` **is a string (a list of character codes).** (`basic_props:string/1`)

  `Dict` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*

  `Dict` **is a dictionary of values. It is a list of pairs** *atom=constant***.** (`http_forms:value_dict/1`)

form_encode_value/3: PREDICATE

No further documentation available for this predicate.

form_decode_value/3: PREDICATE

No further documentation available for this predicate.

lines_to_value/2: PREDICATE

No further documentation available for this predicate.

http_parse_form/2: PREDICATE

Usage: `http_parse_form(Request,Dic)`

Get form data `Dic` from HTTP request `Request`

193.3 Documentation on multifiles

define_flag/3: PREDICATE

Defines a flag as follows:

```
define_flag(raw_form_values,[on,off],off).
```

(See Chapter 155 [Runtime system control and flags], page 683).

If flag is `on`, values returned by `http_parse_form/2` are always atoms, unchanged from its original value.

Usage: `define_flag(Flag,FlagValues,Default)`

- *The following properties hold upon exit:*

  `Flag` is an atom. (`basic_props:atm/1`)

  Define the valid flag values (`basic_props:flag_values/1`)

The predicate is *multifile*.

193.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `strings`, `lists`, `http_grammar`, `multipart_form_data`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `dcg`, `hiord`, `doccomments`, `define_flag`.

194 CGI programming

Author(s): The Ciao Development Team.

This module implements the predicates for CGI processes. It is based on translating the CGI environment to HTTP request terms (see `http/http_server`).

194.1 Usage and interface

- **Library usage:**
  `:- use_module(library(http/cgi)).`
- **Exports:**
  - *Predicates:*
    `cgi_read_request/1`, `my_url/1`, `set_cookie/2`, `get_cookies/1`, `cgi_write_response/1`.
  - *Multifiles:*
    `define_flag/3`.

194.2 Documentation on exports

cgi_read_request/1: PREDICATE

Usage: `cgi_read_request(Request)`

Obtain a HTTP request from the CGI environment variables (for headers) and input stream data (for contents). This request can be processed with other predicates in the `http_server` libraries.

my_url/1: PREDICATE

`my_url(URL)`

Unifies `URL` with the Uniform Resource Locator (WWW address) of this cgi executable.

Usage:

- *Call and exit should be compatible with:*
  `URL` **is a string (a list of character codes).** (`basic_props:string/1`)
- *The following properties should hold upon exit:*
  `URL` **is a string (a list of character codes).** (`basic_props:string/1`)

set_cookie/2: PREDICATE

`set_cookie(Name,Value)`

Sets a cookie of name `Name` and value `Value` (it must be called before `cgi_write_response/1`)

Usage:

- *The following properties should hold at call time:*
  `Name` **is an atom.** (`basic_props:atm/1`)
  `Value` **is an atomic term (an atom or a number).** (`basic_props:constant/1`)

get_cookies/1: PREDICATE

`get_cookies(Cookies)`

Unifies `Cookies` **with a dictionary of** *attribute=value* **pairs of the active cookies for this URL. If the flag** `raw_form_values` **is** `on`, *value***s are always atoms even if they could be interpreted as numbers.**

Usage:

- *The following properties should hold at call time:*
  `Cookies` **is a free variable.** (`term_typing:var/1`)
- *The following properties should hold upon exit:*
  `Cookies` **is a dictionary of values. It is a list of pairs** *atom=constant***.** (`http_forms:value_dict/1`)

cgi_write_response/1: PREDICATE

Usage: `cgi_write_response(Response)`

Writes the CGI response (similar to a HTTP response but sent through stdout)

194.3 Documentation on multifiles

define_flag/3: PREDICATE

Usage: `define_flag(Flag,FlagValues,Default)`

- *The following properties hold upon exit:*
  `Flag` **is an atom.** (`basic_props:atm/1`)
  Define the valid flag values (`basic_props:flag_values/1`)

The predicate is *multifile***.**

194.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `strings`, `stream_utils`, `lists`, `system`, `http_grammar`, `http_forms`, `multipart_form_data`, `http_messages`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `dcg`, `hiord`, `doccomments`, `define_flag`.

195 HTTP messages (response and request)

Author(s): The Ciao Development Team.

Parser/printer for HTTP response and request messages.

See RFC2616 (`https://www.w3.org/Protocols/rfc2616/rfc2616.html`) for a reference of HTTP/1.1 protocol.

The request and response objects are represented as list of terms, which represent HTTP header fields and other elements (such as the status, and contents).

The current list of request headers is:

- **method**(*Method*): *Method* is the HTTP method in lowercase (`head`, `get`, `post`, etc.).
- **timeout**(*T*): *T* specifies the time in seconds to wait for the response. Default is 300 seconds.
- **if_modified_since**(*Date*): Get document only if newer than *Date*. *Date* has the format defined by `http_date/1`.
- **user_agent**(*Agent*): Provides a user-agent field, *Agent* is an atom.
- **authorization**(*Scheme*,*Params*): To provide credentials. See RFC 1945 for details.
- *option*(*Value*): Any unary term, being *Value* an atom, can be used to provide another valid option (e.g. `from('user@machine')`).

The current list of response headers is:

- **status**(*Type,Code,Reason*): *Type* is an atom denoting the response type, *Code* is the status code (an integer), and *Reason* is a string holding the reason phrase.
- **date**(*Date*): *Date* is the date of the response, with format defined by `http_date/1`.
- **location**(*Loc*): This parameter appears when the document has moved, *Loc* is an atom holding the new location.
- **server**(*Server*): *Server* is the server responding, as an atom.
- **www_authenticate**(*Params*): Returned if document is protected, *Params* is a list of chagenges. See RFC 1945 for details.
- **allow**(*Methods*): *Methods* are the methods allowed by the server, as a list of atoms.
- **content_encoding**(*Encoding*): *Encoding* is an atom defining the encoding.
- **expires**(*Date*): *Date* is the date after which the entity should be considered stale. Format defined by `http_date/1`.
- **last_modified**(*Date*): *Date* is the date at which the sender believes the resource was last modified. Format defined by `http_date/1`.
- **pragma**(*String*): Miscellaneous data.
- *header*(*String*): Any other functor *header*/1 is an extension header.

The current list of some common headers is:

- **content_length**(*Length*): *Length* is the length of the document (an integer).
- **content_type**(*Type,Subtype,Params*): Specifies the document content type, *Type* and *Subtype* are atoms, *Params* a list of parameters (e.g. `content_type(text,html,[])`).
- **content**(*Bytes*): *Bytes* is the document content (bytelist/1). If `method(head)` of the HTTP request is used, an empty list is get here.

195.1 Usage and interface

- **Library usage:**
  `:- use_module(library(http/http_messages)).`
- **Exports:**
  - *Predicates:*
    `http_request_str/4`, `name_to_method/2`, `http_response_str/3`.
  - *Regular Types:*
    `http_request_param/1`, `http_response_param/1`.

195.2 Documentation on exports

http_request_param/1: REGTYPE

Usage: `http_request_param(Request)`

`Request` is a parameter of an HTTP request.

http_request_str/4: PREDICATE

Usage 1: `http_request_str(RequestURI,Request,RequestBytes,RequestBytesTail)`

Parse a string into an HTTP request

- *The following properties should hold at call time:*
  `RequestURI` is a free variable. (`term_typing:var/1`)
  `Request` is a free variable. (`term_typing:var/1`)
  `RequestBytes` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `RequestBytesTail` is currently a term which is not a free variable. (`term_typing:nonvar/1`)

Usage 2: `http_request_str(RequestURI,Request,RequestBytes,RequestBytesTail)`

Generate an HTTP request from a list of parameters

- *The following properties should hold at call time:*
  `RequestURI` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `Request` is currently a term which is not a free variable. (`term_typing:nonvar/1`)
  `RequestBytes` is a free variable. (`term_typing:var/1`)
  `RequestBytesTail` is a free variable. (`term_typing:var/1`)

name_to_method/2: PREDICATE

No further documentation available for this predicate.

http_response_param/1: REGTYPE

Usage: `http_response_param(Response)`

`Response` is a parameter of an HTTP response.

http_response_str/3: PREDICATE

No further documentation available for this predicate.

195.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`lists`, `http_grammar`, `http_date`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `isomodes`, `dcg`, `hiord`, `doccomments`.

196 HTTP dates

Author(s): The Ciao Development Team.

Parser/printer for dates in the HTTP protocol.

The format of dates in HTTP/1.1 (RFC1123 and RFC850) (see RFC2616 (`https://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.3`)) is the following:

```
HTTP-date    = rfc1123-date | rfc850-date | asctime-date
rfc1123-date = wkday "," SP date1 SP time SP "GMT"
rfc850-date  = weekday "," SP date2 SP time SP "GMT"
asctime-date = wkday SP date3 SP time SP 4DIGIT
date1        = 2DIGIT SP month SP 4DIGIT
               ; day month year (e.g., 02 Jun 1982)
date2        = 2DIGIT "-" month "-" 2DIGIT
               ; day-month-year (e.g., 02-Jun-82)
date3        = month SP ( 2DIGIT | ( SP 1DIGIT ))
               ; month day (e.g., Jun  2)
time         = 2DIGIT ":" 2DIGIT ":" 2DIGIT
               ; 00:00:00 - 23:59:59
wkday        = "Mon" | "Tue" | "Wed"
             | "Thu" | "Fri" | "Sat" | "Sun"
weekday      = "Monday" | "Tuesday" | "Wednesday"
             | "Thursday" | "Friday" | "Saturday" | "Sunday"
month        = "Jan" | "Feb" | "Mar" | "Apr"
             | "May" | "Jun" | "Jul" | "Aug"
             | "Sep" | "Oct" | "Nov" | "Dec"
```

196.1 Usage and interface

- **Library usage:**
  `:- use_module(library(http/http_date)).`
- **Exports:**
  - *Predicates:*
    `http_date_str/3`.
  - *Regular Types:*
    `http_date/1`, `weekday/1`, `month/1`, `hms_time/1`.

196.2 Documentation on exports

http_date/1: REGTYPE

`http_date(Date)`

`Date` is a term defined as

```
http_date(date(WeekDay,Day,Month,Year,Time)) :-
    weekday(WeekDay),
    int(Day),
    month(Month),
    int(Year),
```

```
hms_time(Time).
```

.

Usage: `http_date(Date)`

`Date` **is a term denoting a date.**

weekday/1: REGTYPE

Usage: `weekday(WeekDay)`

`WeekDay` **is a term denoting a weekday.**

month/1: REGTYPE

Usage: `month(Month)`

`Month` **is a term denoting a month.**

hms_time/1: REGTYPE

Usage: `hms_time(Time)`

`Time` **is an atom of the form** `hh:mm:ss`

http_date_str/3: PREDICATE

No further documentation available for this predicate.

196.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `http_grammar`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `regtypes`, `dcg`, `hiord`, `doccomments`.

197 Common grammar definitions for HTTP

Author(s): The Ciao Development Team.
Some grammar definitions for the HTTP protocol.

197.1 Usage and interface

- **Library usage:**
  `:- use_module(library(http/http_grammar)).`
- **Exports:**
  - *Predicates:*
    `PARSING/2`, `PRINTING/2`, `loalpha/3`, `upalpha/3`, `digit/3`, `http_sp/2`, `http_lws/2`, `http_lws0/2`, `http_crlf/2`, `integer_str/3`, `parse_integer/3`, `http_token/3`, `http_lo_up_token/3`, `http_line/3`, `http_line_atm/3`, `http_lines/3`, `http_field/3`, `http_auth_params/3`, `http_http/4`, `http_status_line/3`, `http_media_type/5`, `http_type_params/3`.

197.2 Documentation on exports

PARSING/2: PREDICATE
No further documentation available for this predicate.

PRINTING/2: PREDICATE
No further documentation available for this predicate.

loalpha/3: PREDICATE
No further documentation available for this predicate.

upalpha/3: PREDICATE
No further documentation available for this predicate.

digit/3: PREDICATE
No further documentation available for this predicate.

http_sp/2: PREDICATE
No further documentation available for this predicate.

http_lws/2: PREDICATE
No further documentation available for this predicate.

http_lws0/2: PREDICATE

No further documentation available for this predicate.

http_crlf/2: PREDICATE

No further documentation available for this predicate.

integer_str/3: PREDICATE

No further documentation available for this predicate.

parse_integer/3: PREDICATE

No further documentation available for this predicate.

http_token/3: PREDICATE

No further documentation available for this predicate.

http_lo_up_token/3: PREDICATE

No further documentation available for this predicate.

http_line/3: PREDICATE

No further documentation available for this predicate.

http_line_atm/3: PREDICATE

No further documentation available for this predicate.

http_lines/3: PREDICATE

No further documentation available for this predicate.

http_field/3: PREDICATE

No further documentation available for this predicate.

http_auth_params/3: PREDICATE

No further documentation available for this predicate.

http_http/4: PREDICATE

No further documentation available for this predicate.

http_status_line/3: PREDICATE

No further documentation available for this predicate.

http_media_type/5: PREDICATE

No further documentation available for this predicate.

http_type_params/3: PREDICATE

No further documentation available for this predicate.

197.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `strings`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `dcg`, `doccomments`.

198 Web programming libraries (PiLLoW)

Author(s): Daniel Cabeza, Manuel Hermenegildo, The Ciao Development Team.

This package implements a collection of libraries for HTML/XML/JSON processing.

See `http_doc` **libraries for the protocol and programming.**

198.1 Other PiLLoW releases

This code is is based on the original PiLLoW library [CHV96, CH01]), which can be downloaded from `http://clip.dia.fi.upm.es/Software/pillow/pillow.html`**. There is also a** *PiLLoW on-line tutorial* **(slides) at** `http://clip.dia.fi.upm.es/logalg/slides/C_pillow/C_pillow.pdf` **which illustrates the basic features and provides a number of examples of PiLLoW use.**

198.2 Usage and interface

- **Library usage:**
  `:- use_package(pillow).`
  or
  `:- module(...,...,[pillow]).`
- **New operators defined:**
  `$/2` [**150,xfx**], `$/1` [**150,fx**].
- **Implicit imports:**
  - *System library modules:*
    `html`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

199 HTML/XML parser and generator

Author(s): Daniel Cabeza, Manuel Hermenegildo, Sacha Varma, The Ciao Development Team.

This module implements predicates for HTML/ XML generation and parsing.

199.1 Usage and interface

- **Library usage:**
  `:- use_module(library(pillow/html)).`
- **Exports:**
  - *Predicates:*
    `output_html/1`, `html2terms/2`, `xml2terms/2`, `html_template/3`.
  - *Regular Types:*
    `canonic_html_term/1`, `canonic_xml_term/1`, `html_term/1`.
  - *Multifiles:*
    `html_expansion/2`.

199.2 Documentation on exports

canonic_html_term/1: REGTYPE

A term representing HTML code in canonical, structured way. It is a list of terms defined by the following predicate:

```
canonic_html_item(comment(S)) :-
    string(S).
canonic_html_item(declare(S)) :-
    string(S).
canonic_html_item(env(Tag,Atts,Terms)) :-
    atm(Tag),
    list(tag_attrib,Atts),
    canonic_html_term(Terms).
canonic_html_item($(Tag,Atts)) :-
    atm(Tag),
    list(tag_attrib,Atts).
canonic_html_item(S) :-
    string(S).

tag_attrib(Att) :-
    atm(Att).
tag_attrib(Att=Val) :-
    atm(Att),
    string(Val).
```

Each structure represents one HTML construction:

env(*tag*,*attribs*,*terms*)
: **An HTML environment, with name** *tag***, list of attributes** *attribs* **and contents** *terms***.**

$(*tag*,*attribs*)
: An HTML element of name *tag* and list of attributes *attribs*. `($)/2` is defined by the pillow package as an infix, binary operator.

comment(*string*)
: An HTML comment (translates to/from `<!--`*string*`-->`).

declare(*string*)
: An HTML declaration, they are used only in the header (translates to/from `<!`*string*`>`).

string
: Normal text is represented as a list of character codes.

For example, the term

```
env(a,[href="www.therainforestsite.com"],
      ["Visit ",img$[src="TRFS.gif"]])
```

is output to (or parsed from):

```
<a href="www.therainforestsite.com">Visit <img src="TRFS.gif"></a>
```

Usage: `canonic_html_term(HTMLTerm)`

`HTMLTerm` is a term representing HTML code in canonical form.

canonic_xml_term/1: REGTYPE

A term representing XML code in canonical, structured way. It is a list of terms defined by the following predicate (see `tag_attrib/1` definition in `canonic_html_term/1`):

```
canonic_xml_item(Term) :-
    canonic_html_item(Term).
canonic_xml_item(xmldecl(Atts)) :-
    list(tag_attrib,Atts).
canonic_xml_item(env(Tag,Atts,Terms)) :-
    atm(Tag),
    list(tag_attrib,Atts),
    canonic_xml_term(Terms).
canonic_xml_item(elem(Tag,Atts)) :-
    atm(Tag),
    list(tag_attrib,Atts).
```

In addition to the structures defined by `canonic_html_term/1` (the `($)/2` structure appears only in malformed XML code), the following structures can be used:

elem(*tag*,*atts*)
: Specifies an XML empty element of name *tag* and list of attributes *atts*. For example, the term

```
elem(arc,[weigh="3",begin="n1",end="n2"])
```

is output to (or parsed from):

```
<arc weigh="3" begin="n1" end="n2"/>
```

xmldecl(*atts*)
: Specifies an XML declaration with attributes *atts* (translates to/from `<?xml` *atts*`?>`)

Usage: `canonic_xml_term(XMLTerm)`

`XMLTerm` **is a term representing XML code in canonical form.**

html_term/1: REGTYPE

A term which represents HTML or XML code in a structured way. In addition to the structures defined by `canonic_html_term/1` or `canonic_xml_term/1`, the following structures can be used:

begin(*tag*,*atts*)
: **It translates to the start of an HTML environment of name *tag* and attributes *atts*. There exists also a begin(*tag*) structure. Useful, in conjunction with the next structure, when including in a document output generated by an existing piece of code (e.g. *tag* = `pre`). Its use is otherwise discouraged.**

end(*tag*)
: **Translates to the end of an HTML environment of name *tag*.**

start
: **Used at the beginning of a document (translates to `<html>`).**

end
: **Used at the end of a document (translates to `</html>`).**

`--`
: **Produces a horizontal rule (translates to `<hr>`).**

\
: **Produces a line break (translates to `<br>`).**

$
: **Produces a paragraph break (translates to `<p>`).**

image(*address*)
: **Used to include an image of address (URL) *address* (equivalent to `img$[src=`*address*`]`).**

image(*address*,*atts*)
: **As above with the list of attributes *atts*.**

ref(*address*,*text*)
: **Produces a hypertext link, *address* is the URL of the referenced resource, *text* is the text of the reference (equivalent to `a([href=`*address*`],`*text*`)`).**

label(*name*,*text*)
: **Labels *text* as a target destination with label *name* (equivalent to `a([name=`*name*`],`*text*`)`).**

heading(*n*,*text*)
: **Produces a heading of level *n* (between 1 and 6), *text* is the text to be used as heading. Useful when one wants a heading level relative to another heading (equivalent to `h`*n*`(`*text*`)`).**

itemize(*items*)
: **Produces a list of bulleted items, *items* is a list of corresponding HTML terms (translates to a `<ul>` environment).**

enumerate(*items*)
: **Produces a list of numbered items, *items* is a list of corresponding HTML terms (translates to a `<ol>` environment).**

description(*defs*)
: **Produces a list of defined items, *defs* is a list whose elements are definitions, each of them being a Prolog sequence (composed by `','/2` operators). The last element of the sequence is the definition, the other (if any) are the defined terms (translates to a `<dl>` environment).**

preformatted(*text*)
: Used to include preformatted text, *text* is a list of HTML terms, each element of the list being a line of the resulting document (translates to a <pre> environment).

verbatim(*text*)
: Used to include text verbatim, special HTML characters (<,>,&," and space) are translated into its quoted HTML equivalent.

prolog_term(*term*)
: Includes any prolog term *term*, represented in functional notation. Variables are output as _.

nl
: Used to include a newline in the HTML source (just to improve human readability).

entity(*name*)
: Includes the entity of name *name* (ISO-8859-1 special character).

start_form(*addr*,*atts*)
: Specifies the beginning of a form. *addr* is the address (URL) of the program that will handle the form, and *atts* other attributes of the form, as the method used to invoke it. If *atts* is not present (there is only one argument) the method defaults to POST.

start_form
: Specifies the beginning of a form without assigning address to the handler.

end_form
: Specifies the end of a form.

checkbox(*name*,*state*)
: Specifies an input of type `checkbox` with name *name*, *state* is `on` if the checkbox is initially checked.

radio(*name*,*value*,*selected*)
: Specifies an input of type `radio` with name *name* (several radio buttons which are interlocked must share their name), *value* is the the value returned by the button, if *selected*=*value* the button is initially checked.

input(*type*,*atts*)
: Specifies an input of type *type* with a list of attributes *atts*. Possible values of *type* are `text`, `hidden`, `submit`, `reset`, ldots

textinput(*name*,*atts*,*text*)
: Specifies an input text area of name *name*. *text* provides the default text to be shown in the area, *atts* a list of attributes.

option(*name*,*val*,*options*)
: Specifies a simple option selector of name *name*, *options* is the list of available options and *val* is the initial selected option (if *val* is not in *options* the first item is selected by default) (translates to a <select> environment).

menu(*name*,*atts*,*items*)
: Specifies a menu of name *name*, list of attributes *atts* and list of options *items*. The elements of the list *items* are marked with the prefix operator `$` to indicate that they are selected (translates to a <select> environment).

name(*text*)
: A term with functor *name*/1, different from the special functors defined herein, represents an HTML environment of name *name* and included text *text*. For example, the term

```
address('clip@clip.dia.fi.upm.es')
```

is translated into the HTML source

```
<address>clip@clip.dia.fi.upm.es</address>
```

name(*atts*,*text*)
A term with functor *name*/2, different from the special functors defined herein, represents an HTML environment of name *name*, attributes *atts* and included text *text*. For example, the term

```
a([href='http://www.clip.dia.fi.upm.es/'],"Clip home")
```

represents the HTML source

```
<a href="http://www.clip.dia.fi.upm.es/">Clip home</a>
```

Usage: `html_term(HTMLTerm)`

`HTMLTerm` **is a term representing HTML code.**

output_html/1: PREDICATE

`output_html(HTMLTerm)`

Outputs `HTMLTerm`**, interpreted as an** `html_term/1`**, to current output stream.**

Usage:

- *The following properties should hold at call time:*
  `HTMLTerm` **is a term representing HTML code.** (`html:html_term/1`)

html2terms/2: PREDICATE

`html2terms(String,Terms)`

`String` **is a character list containing HTML code and** `Terms` **is its prolog structured representation.**

Usage 1:

Translates an HTML-term into the HTML code it represents.

- *The following properties should hold at call time:*
  `String` **is a free variable.** (`term_typing:var/1`)
  `Terms` **is a term representing HTML code.** (`html:html_term/1`)
- *The following properties should hold upon exit:*
  `String` **is a string (a list of character codes).** (`basic_props:string/1`)

Usage 2:

Translates HTML code into a structured HTML-term.

- *Call and exit should be compatible with:*
  `Terms` **is a term representing HTML code in canonical form.** (`html:canonic_html_term/1`)
- *The following properties should hold at call time:*
  `String` **is a string (a list of character codes).** (`basic_props:string/1`)
- *The following properties should hold upon exit:*
  `Terms` **is a term representing HTML code in canonical form.** (`html:canonic_html_term/1`)

xml2terms/2: PREDICATE

`xml2terms(String,Terms)`

`String` is a character list containing XML code and `Terms` is its prolog structured representation.

Usage 1:

Translates a XML-term into the XML code it represents.

- *The following properties should hold at call time:*
  `String` is a free variable. (`term_typing:var/1`)
  `Terms` is a term representing HTML code. (`html:html_term/1`)
- *The following properties should hold upon exit:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)

Usage 2:

Translates XML code into a structured XML-term.

- *Call and exit should be compatible with:*
  `Terms` is a term representing XML code in canonical form. (`html:canonic_xml_term/1`)
- *The following properties should hold at call time:*
  `String` is a string (a list of character codes). (`basic_props:string/1`)
- *The following properties should hold upon exit:*
  `Terms` is a term representing XML code in canonical form. (`html:canonic_xml_term/1`)

html_template/3: PREDICATE

`html_template(Chars,Terms,Dict)`

Interprets `Chars` as an HTML template returning in `Terms` the corresponding structured HTML-term, which includes variables, and unifying `Dict` with a dictionary of those variables (an incomplete list of *name*=*Var* pairs). An HTML template is standard HTML code, but in which "slots" can be defined and given an identifier. These slots represent parts of the HTML code in which other HTML code can be inserted, and are represented in the HTML-term as free variables. There are two kinds of variables in templates:

- Variables representing page contents. A variable with name *name* is defined with the special tag `<V>`*name*`</V>`.
- Variables representing tag attributes. They occur as an attribute or an attribute value starting with `_`, followed by its name, which must be formed by alphabetic characters.

As an example, suposse the following HTML template:

```
<html>
<body bgcolor=_bgcolor>
<v>content</v>
</body>
</html>
```

The following query in the Ciao toplevel shows how the template is parsed, and the dictionary returned:

```
?- file_to_string('template.html',_S), html_template(_S,Terms,Dict).

Dict = [bgcolor=_A,content=_B|_],
Terms = [env(html,[],["
",env(body,[bgcolor=_A],["
",_B,"
"]),"
"]),"
"] ?

yes
```

If a dictionary with values is supplied at call time, then variables are unified accordingly inside the template:

```
?- file_to_string('template.html',_S),
   html_template(_S,Terms,[content=b("hello world!"),bgcolor="white"]).

Terms = [env(html,[],["
",env(body,[bgcolor="white"],["
",b("hello world!"),"
"]),"
"]),"
"] ?

yes
```

Usage:

– *Call and exit should be compatible with:*
  `Terms` **is a term representing HTML code in canonical form.** (`html:canonic_html_term/1`)
  `Dict` **is a list.** (`basic_props:list/1`)
– *The following properties should hold at call time:*
  `Chars` **is a string (a list of character codes).** (`basic_props:string/1`)
– *The following properties should hold upon exit:*
  `Terms` **is a term representing HTML code in canonical form.** (`html:canonic_html_term/1`)
  `Dict` **is a list.** (`basic_props:list/1`)

199.3 Documentation on multifiles

html_expansion/2: PREDICATE

Usage: `html_expansion(Term,Expansion)`

Hook predicate to define macros. Expand occurrences of `Term` **into** `Expansion`**, in** `output_html/1`**. Take care to not transform something into itself!**

The predicate is ***multifile*****.**

199.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `stream_utils`, `strings`, `lists`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `isomodes`, `dcg`.

199.5 Other information

The code uses input from from L. Naish's forms and Francisco Bueno's previous Chat interface. Other people who have contributed are (please inform us if we leave out anybody): Markus Fromherz.

200 JSON encoder and decoder

Author(s): Jose F. Morales.

Stability: [devel] Currently the subject of active development and/or research. Functionality may be limited and API and/or functionality may change without warning or deprecation period. Not recommended yet for use in production.

This module defines a term representation for JSON (`http://json.org/`) (JavaScript Object Notation), as well as encoders and decoders.

200.1 Usage and interface

- **Library usage:**
  `:- use_module(library(pillow/json)).`
- **Exports:**
  - *Predicates:*
    `json_to_string/2`, `string_to_json/2`, `json_get/3`, `json_get_atm/3`, `atomiclst_to_json_strlist/2`, `atomic_to_json_str/2`, `json_as_atm/2`, `json_as_num/2`.
  - *Regular Types:*
    `json/1`, `json_attrs/1`, `json_attr/1`, `json_val/1`, `json_list/1`.

200.2 Documentation on exports

json/1: REGTYPE

Usage:

A JSON object.

```
json(json(Attrs)) :-
    json_attrs(Attrs).
```

json_attrs/1: REGTYPE

Usage:

Attributes (pairs of key/value) of a JSON object.

```
json_attrs([]).
json_attrs([X|Xs]) :-
    json_attr(X),
    json_attrs(Xs).
```

json_attr/1: REGTYPE

A regular type, defined as follows:

```
json_attr(Id=Val) :-
    atm(Id),
    json_val(Val).
```

json_val/1: REGTYPE

A regular type, defined as follows:

```
json_val(string(X)) :-
    string(X).
json_val(X) :-
    number(X).
json_val(X) :-
    json(X).
json_val(X) :-
    json_list(X).
json_val(true).
json_val(false).
json_val(null).
```

json_list/1: REGTYPE

Usage:

A list of JSON elements

json_to_string/2: PREDICATE

Usage: `json_to_string(Term,String)`

Encode a JSON value `Term` as a character list.

- *Call and exit should be compatible with:*
  `json:json_val(Term)` (`json:json_val/1`)
  `String` **is a string (a list of character codes).** (`basic_props:string/1`)
- *The following properties should hold at call time:*
  `Term` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

string_to_json/2: PREDICATE

Usage: `string_to_json(String,Term)`

Decode a character list as a JSON value `Term`.

- *Call and exit should be compatible with:*
  `String` **is a string (a list of character codes).** (`basic_props:string/1`)
  `json:json_val(Term)` (`json:json_val/1`)
- *The following properties should hold at call time:*
  `String` **is currently a term which is not a free variable.** (`term_typing:nonvar/1`)

json_get/3: PREDICATE

No further documentation available for this predicate.

json_get_atm/3: PREDICATE

No further documentation available for this predicate.

atomiclst_to_json_strlist/2: PREDICATE

No further documentation available for this predicate.

atomic_to_json_str/2: PREDICATE

No further documentation available for this predicate.

json_as_atm/2: PREDICATE

No further documentation available for this predicate.

json_as_num/2: PREDICATE

No further documentation available for this predicate.

200.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  `strings`, `lists`.
- *Packages:*

  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `basicmodes`, `dcg`, `fsyntax`.

201 Color space transformations

Author(s): Jose F. Morales.

This module defines color space transformations.

201.1 Usage and interface

- **Library usage:**
  `:- use_module(library(pillow/color_space)).`
- **Exports:**
  - *Predicates:*
    `hsv_to_rgb/2`, `hsl_to_hsv/2`, `hsl_to_rgb/2`, `rgb_to_hex/2`, `color_to_hex/2`.
  - *Properties:*
    `rgb/1`, `hsv/1`, `hsv/1`, `color/1`, `hex/1`.

201.2 Documentation on exports

rgb/1: PROPERTY

Usage: `rgb(C)`

`C` **is a color in RGB space (Red-Green-Blue)**

hsv/1: PROPERTY

Usage: `hsv(C)`

`C` **is a color in HSV space (Hue-Saturation-Value)**

hsv/1: PROPERTY

Usage: `hsv(C)`

`C` **is a color in HSV space (Hue-Saturation-Value)**

color/1: PROPERTY

Usage: `color(C)`

`C` **is a color in any color space**

hex/1: PROPERTY

Usage: `hex(Hex)`

`Hex` **is a color in** `#RRGGBB` **hexadecimal representation**

hsv_to_rgb/2: PREDICATE

Usage: `hsv_to_rgb(HSV,RGB)`

Transform the color `HSV` in `hsv/1` space to `RGB` in `rgb/1` space

– *Call and exit should be compatible with:*

`HSV` **is a color in HSV space (Hue-Saturation-Value)** (`color_space:hsv/1`)

`RGB` **is a color in RGB space (Red-Green-Blue)** (`color_space:rgb/1`)

hsl_to_hsv/2: PREDICATE

No further documentation available for this predicate.

hsl_to_rgb/2: PREDICATE

Usage 1: `hsl_to_rgb(HSL,HSV)`

Transform the color `HSL` in `hsl/1` space to `HSV` in `hsv/1` space

– *Call and exit should be compatible with:*

`HSL` **is a color in HSL space (Hue-Saturation-Lightness)** (`color_space:hsl/1`)

`HSV` **is a color in HSV space (Hue-Saturation-Value)** (`color_space:hsv/1`)

Usage 2: `hsl_to_rgb(HSL,RGB)`

Transform the color `HSL` in `hsl/1` space to `RGB` in `rgb/1` space

– *Call and exit should be compatible with:*

`HSL` **is a color in HSL space (Hue-Saturation-Lightness)** (`color_space:hsl/1`)

`RGB` **is a color in RGB space (Red-Green-Blue)** (`color_space:rgb/1`)

rgb_to_hex/2: PREDICATE

Usage: `rgb_to_hex(RGB,Hex)`

Transform the color `RGB` in `rgb/1` space to `Hex` in `hex/1` representation

– *Call and exit should be compatible with:*

`RGB` **is a color in RGB space (Red-Green-Blue)** (`color_space:rgb/1`)

`Hex` **is a color in `#RRGGBB` hexadecimal representation** (`color_space:hex/1`)

color_to_hex/2: PREDICATE

Usage: `color_to_hex(Color,Hex)`

Obtain the atom `Hex` representation (`hex/1`) of color `Color`

– *Call and exit should be compatible with:*

`Color` **is a color in any color space** (`color_space:color/1`)

`Hex` **is a color in `#RRGGBB` hexadecimal representation** (`color_space:hex/1`)

201.3 Documentation on imports

This module has the following direct dependencies:

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

202 Shell-style pathname pattern expansion

Author(s): Manuel Hermenegildo (original `file_find.pl`), Jose F. Morales.

This module provides file searching predicates to locate pathnames matching a *globbing* (`http://en.wikipedia.org/wiki/Glob_(programming)`) pattern (shell wildcard expansion).

202.1 Usage and interface

- **Library usage:**
  `:- use_module(library(glob)).`
- **Exports:**
  - *Predicates:*
    `glob/3`, `glob/2`.
  - *Regular Types:*
    `glob_pattern/1`.

202.2 Documentation on exports

glob_pattern/1: REGTYPE

Usage: `glob_pattern(Pattern)`

`Pattern` **is a pathname pattern whose components may contain shell-style wildcards.**

glob/3: PREDICATE

`glob(Directory,Pattern,FileList)`

Search the list of pathnames `FileList` matching the specified pathname pattern `Pattern`. If `Pattern` is an absolute path, `Directory` is ignored. Otherwise, all matches are relative to `Directory`. If `Directory` does not exist `FileList` is empty. The shortest version `glob/2` can be used when `Directory` is `'.'` (the current directory).

For example, `glob/2` and `glob/3` will give the following results in a typical Unix installation:

```
?- use_module(library(glob)).

yes
?- cd('/bin').

yes
?- glob('e*', F).

F = [expr,ed,echo] ?

yes
?- glob('/', 'bin/e*', F).

F = ['bin/expr','bin/ed','bin/echo'] ?
```

```
yes
?- glob('/tmp', '../bin/e*', F).

F = ['../bin/expr','../bin/ed','../bin/echo'] ?

yes
?- cd('/tmp').

yes
?- glob('/bin/e*', F).

F = ['/bin/expr','/bin/ed','/bin/echo'] ?

yes
```

Usage: `glob(Directory,Pattern,FileList)`

`FileList` **is the list of pathnames matching the specified pathname pattern** `Pattern`**, relative to** `Directory`**.**

– *Call and exit should be compatible with:*

`Directory` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

`Pattern` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

`FileList` **is a list of** `pathnames`**.** (`basic_props:list/2`)

glob/2: PREDICATE

Usage: `glob(Pattern,FileList)`

Like `glob/3`**, relative to the current directory (equivalent to** `glob('.', Pattern, FileList)`**)**

– *Call and exit should be compatible with:*

`Pattern` **is a pathname (encoded as an atom)** (`pathnames:pathname/1`)

`FileList` **is a list of** `pathnames`**.** (`basic_props:list/2`)

202.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`regexp_code`, `system`, `pathnames`, `lists`, `aggregates`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `isomodes`, `hiord`.

203 Pattern (regular expression) matching

Author(s): The Ciao Development Team.

This library provides facilities for matching strings and terms against *patterns*. Some prolog flags are available to control its behavior.

- **The flag**

  `controls whether to perform case-insensitive (on) or case-sensitive (off) match. The default value is off.`
- `There is a syntax facility that allows using this regular expression matching in a similar way to unification. You can use =~ "regexp" as an argument of a predicate, and then that argument must match with the regexp. For example:`

  ```
  pred ( =~ "ab*c", B) :- ...
  ```

  `is equivalent to`

  ```
  pred (X,B) :- match_posix("ab*c",X,R), ...
  ```

  `Two additional flags control this matching. The first one is`

  `. Its values are shell, posix, list and pred. Their effect is as if changing in the example above the call to match_posix/3 by a call to, respectively, match_shell/2, match_posix/3, match_struct/3, and match_pred/3. The default value is posix. The other prolog flag is`

  `. Its values are on and off. The off value means replacing in the example R with []. If the value is on, then R is a variable. The default value is on.`

203.1 Usage and interface

- **Library usage:**

  `:- use_package(regexp).`

  or

  `:- module(...,...,[regexp]).`
- **New operators defined:**

  `=~/1 [200,fy].`
- **Implicit imports:**
  - *System library modules:*

    `regexp_code.`
  - *Packages:*

    `prelude, initial, condcomp, assertions, assertions/assertions_basic.`

203.2 Documentation on internals

match_shell/3: PREDICATE

Usage: `match_shell(Exp,IN,Rest)`

Matches IN against Exp. Rest is the longest remainder of the string after the match. For example, match_shell("??*","foo.pl",Tail) succeeds, instantiating Tail to "o.pl".

– *The following properties should hold at call time:*

Exp is a shell regular expression to match against. (regexp_code:shell_regexp/1)

IN is a string (a list of character codes). (basic_props:string/1)

Rest is a string (a list of character codes). (basic_props:string/1)

match_shell/2: PREDICATE

Usage: match_shell(Exp,IN)

Matches completely IN (no tail can remain unmatched) against Exp similarly to match_shell/3.

– *The following properties should hold at call time:*

Exp is a shell regular expression to match against. (regexp_code:shell_regexp/1)

IN is a string (a list of character codes). (basic_props:string/1)

match_posix/2: PREDICATE

Usage: match_posix(Exp,IN)

Matches completely IN (no tail can remain unmatched) against Exp similarly to match_posix/3.

– *The following properties should hold at call time:*

Exp is a shell regular expression to match against. (regexp_code:shell_regexp/1)

IN is a string (a list of character codes). (basic_props:string/1)

match_posix/4: PREDICATE

Usage: match_posix(Exp,In,Match,Rest)

– *The following properties should hold at call time:*

Exp is a shell regular expression to match against. (regexp_code:shell_regexp/1)

In is a string (a list of character codes). (basic_props:string/1)

Match is a list of strings. (basic_props:list/2)

Rest is a string (a list of character codes). (basic_props:string/1)

match_posix_rest/3: PREDICATE

Usage: match_posix_rest(Exp,IN,Rest)

Matches IN against Exp. Tail is the remainder of the string after the match. For example, match_posix("ab*c","abbbbcdf",Tail) succeeds, instantiating Tail to "df".

- *The following properties should hold at call time:*

  Exp is a posix regular expression to match against. (regexp_code:posix_regexp/1)

  IN is a string (a list of character codes). (basic_props:string/1)

  Rest is a string (a list of character codes). (basic_props:string/1)

match_posix_matches/3: PREDICATE

Usage: match_posix_matches(Exp,IN,Matches)

Matches completely IN against Exp. Exp can contain *anchored expressions* of the form \(regexp\). Matches will contain a list of the *anchored expression* which were matched on success. Note that since POSIX expressions are being read inside a string, backslashes will have to be doubled. For example,

```
?- match_posix_matches("\(aa|bb\)\(bb|aa\)", "bbaa", M).
M = ["bb","aa"] ? ;
no

?- match_posix_matches("\(aa|bb\)\(bb|aa\)", "aabb", M).
M = ["aa","bb"] ? ;
no
```

- *The following properties should hold at call time:*

  Exp is a shell regular expression to match against. (regexp_code:shell_regexp/1)

  IN is a string (a list of character codes). (basic_props:string/1)

  Matches is a list of strings. (basic_props:list/2)

match_struct/4: PREDICATE

Usage: match_struct(Exp,IN,Rest,Tail)

Matches IN against Exp. Tail is the remainder of the list of atoms IN after the match. For example, match_struct([a,*(b),c],[a,b,b,b,c,d,e],Tail) succeeds, instantiating Tail to [d,e].

- *Call and exit should be compatible with:*

  Exp is a struct regular expression to match against. (regexp_code:struct_regexp/1)

  IN is a string (a list of character codes). (basic_props:string/1)

  Rest is a string (a list of character codes). (basic_props:string/1)

match_pred/2: PREDICATE

No further documentation available for this predicate.

replace_first/4: PREDICATE

Usage: `replace_first(IN,Old,New,Resul)`

Replace the first occurrence of the Old by New in IN and copy the result in Resul.

- *The following properties should hold at call time:*

  IN is a string (a list of character codes). (basic_props:string/1)

  Old is a posix regular expression to match against. (regexp_code:posix_regexp/1)

  New is a string (a list of character codes). (basic_props:string/1)

  Resul is a string (a list of character codes). (basic_props:string/1)

replace_all/4: PREDICATE

Usage: `replace_all(IN,Old,New,Resul)`

Replace all occurrences of the Old by New in IN and copy the result in Resul.

- *The following properties should hold at call time:*

  IN is a string (a list of character codes). (basic_props:string/1)

  Old is a posix regular expression to match against. (regexp_code:posix_regexp/1)

  New is a string (a list of character codes). (basic_props:string/1)

  Resul is a string (a list of character codes). (basic_props:string/1)

204 regexp_code (library)

204.1 Usage and interface

- **Library usage:**
  `:- use_module(library(regexp/regexp_code)).`
- **Exports:**
  - *Predicates:*
    `match_shell/3`, `match_shell/2`, `match_posix/2`, `match_posix/4`, `match_posix_rest/3`, `match_posix_matches/3`, `match_struct/4`, `match_term/2`, `replace_first/4`, `replace_all/4`.
  - *Regular Types:*
    `shell_regexp/1`, `posix_regexp/1`, `struct_regexp/1`.

204.2 Documentation on exports

match_shell/3: PREDICATE

Usage: `match_shell(Exp,IN,Rest)`

Matches IN against Exp. Rest is the longest remainder of the string after the match. For example, `match_shell("??*","foo.pl",Tail)` succeeds, instantiating Tail to "o.pl".

- *The following properties should hold at call time:*
  Exp is a shell regular expression to match against. (regexp_code:shell_regexp/1)
  IN is a string. (regexp_code:string/1)
  Rest is a string. (regexp_code:string/1)

match_shell/2: PREDICATE

Usage: `match_shell(Exp,IN)`

Matches completely IN (no tail can remain unmatched) against Exp similarly to match_shell/3.

- *The following properties should hold at call time:*
  Exp is a shell regular expression to match against. (regexp_code:shell_regexp/1)
  IN is a string. (regexp_code:string/1)

match_posix/2: PREDICATE

Usage: `match_posix(Exp,IN)`

Matches completely IN (no tail can remain unmatched) against Exp similarly to match_posix/3.

– *The following properties should hold at call time:*

Exp is a shell regular expression to match against. (regexp_code:shell_regexp/1)

IN is a string. (regexp_code:string/1)

match_posix/4: PREDICATE

Usage: match_posix(Exp,In,Match,Rest)

– *The following properties should hold at call time:*

Exp is a shell regular expression to match against. (regexp_code:shell_regexp/1)

In is a string. (regexp_code:string/1)

Match is a list of strings. (basic_props:list/2)

Rest is a string. (regexp_code:string/1)

match_posix_rest/3: PREDICATE

Usage: match_posix_rest(Exp,IN,Rest)

Matches IN against Exp. Tail is the remainder of the string after the match. For example, match_posix("ab*c","abbbbcdf",Tail) succeeds, instantiating Tail to "df".

– *The following properties should hold at call time:*

Exp is a posix regular expression to match against. (regexp_code:posix_regexp/1)

IN is a string. (regexp_code:string/1)

Rest is a string. (regexp_code:string/1)

match_posix_matches/3: PREDICATE

Usage: match_posix_matches(Exp,IN,Matches)

Matches completely IN against Exp. Exp can contain *anchored expressions* of the form \(regexp\). Matches will contain a list of the *anchored expression* which were matched on success. Note that since POSIX expressions are being read inside a string, backslashes will have to be doubled. For example,

```
?- match_posix_matches("\(aa|bb\)\(bb|aa\)", "bbaa", M).
M = ["bb","aa"] ? ;
no

?- match_posix_matches("\(aa|bb\)\(bb|aa\)", "aabb", M).
M = ["aa","bb"] ? ;
no
```

– *The following properties should hold at call time:*

Exp is a shell regular expression to match against. (regexp_code:shell_regexp/1)

IN is a string. (regexp_code:string/1)

Matches is a list of strings. (basic_props:list/2)

match_struct/4: PREDICATE

Usage: `match_struct(Exp,IN,Rest,Tail)`

Matches `IN` against `Exp`. `Tail` is the remainder of the list of atoms `IN` after the match. For example, `match_struct([a,*(b),c],[a,b,b,b,c,d,e],Tail)` succeeds, instantiating `Tail` to `[d,e]`.

– *Call and exit should be compatible with:*

`Exp` is a struct regular expression to match against. (`regexp_code:struct_regexp/1`)

`IN` is a string. (`regexp_code:string/1`)

`Rest` is a string. (`regexp_code:string/1`)

match_term/2: PREDICATE

Usage: `match_term(Term1,Term2)`

Tests if two terms `Term1` and `Term2` match using shell regular expressions.

replace_first/4: PREDICATE

Usage: `replace_first(IN,Old,New,Resul)`

Replace the first occurrence of the `Old` by `New` in `IN` and copy the result in `Resul`.

– *The following properties should hold at call time:*

`IN` is a string. (`regexp_code:string/1`)

`Old` is a posix regular expression to match against. (`regexp_code:posix_regexp/1`)

`New` is a string. (`regexp_code:string/1`)

`Resul` is a string. (`regexp_code:string/1`)

replace_all/4: PREDICATE

Usage: `replace_all(IN,Old,New,Resul)`

Replace all occurrences of the `Old` by `New` in `IN` and copy the result in `Resul`.

– *The following properties should hold at call time:*

`IN` is a string. (`regexp_code:string/1`)

`Old` is a posix regular expression to match against. (`regexp_code:posix_regexp/1`)

`New` is a string. (`regexp_code:string/1`)

`Resul` is a string. (`regexp_code:string/1`)

shell_regexp/1: REGTYPE

Usage: `shell_regexp(P)`

`P` is a shell regular expression to match against.

posix_regexp/1: REGTYPE

Usage: `posix_regexp(P)`

`P` is a posix regular expression to match against.

struct_regexp/1: REGTYPE

Usage: `struct_regexp(P)`

`P` is a struct regular expression to match against.

204.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `lists`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `dcg`, `regtypes`, `define_flag`.

205 Text templates

Author(s): Jose F. Morales, The Ciao Development Team.

A template is a text marked-up with special tags that contain expressions. Template evaluation replaces tags with the evaluation of such expressions, w.r.t. a substitution of parameters.

Tags must have the form {{*Expr*}}, where *Expr* is a parameter name.

Example: given the file n.txt.skel:

```
Dear {{Author}},

on behalf of the program committee of {{Conf}}, we are pleased to
inform you that your paper

{{Title}}

has been accepted for inclusion in the post-conference proceedings.
```

then the query:

```
?- eval_template_file('n.txt.skel', [
     'Author' = 'Alain',
     'Title' = 'The Birth of Prolog',
     'Conf' = 'CACM 93'
   ], 'n.txt').
```

produces the following text:

```
Dear Alain,

on behalf of the program committee of CACM 93, we are pleased to
inform you that your paper

The Birth of Prolog

has been accepted for inclusion in the post-conference proceedings.
```

205.1 Usage and interface

- **Library usage:**
  `:- use_module(library(text_template)).`
- **Exports:**
  - *Predicates:*
    `eval_template_file/3`, `eval_template_string/3`.

205.2 Documentation on exports

eval_template_file/3: PREDICATE

Usage: `eval_template_file(InFile,Subst,OutFile)`

Like `eval_template_string/3`, using files.

eval_template_string/3: PREDICATE

Usage: `eval_template_string(In,Subst,Out)`

Evaluate the template string `In` to generate `Out` using the values in `Subst`.

205.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `regexp_code`, `lists`, `stream_utils`, `dict`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`, `isomodes`, `hiord`, `regexp`.

206 Printing status and error messages

Author(s): The Ciao Development Team.

This is a very simple library for printing status and error messages to the console.

206.1 Usage and interface

- **Library usage:**

  `:- use_module(library(messages)).`
- **Exports:**
  - *Predicates:*

    `show_message/2`, `show_message/3`, `show_message/4`, `error_message/1`, `error_message/2`, `error_message/3`, `warning_message/1`, `warning_message/2`, `warning_message/3`, `note_message/1`, `note_message/2`, `note_message/3`, `simple_message/1`, `simple_message/2`, `optional_message/2`, `optional_message/3`, `debug_message/1`, `debug_message/2`.
  - *Regular Types:*

    `message_t/1`.
  - *Multifiles:*

    `issue_debug_messages/1`.

206.2 Documentation on exports

show_message/2: PREDICATE

Usage: `show_message(Type,Text)`

The text provided in Text is printed as a message of type Type.

- *The following properties should hold at call time:*

  The types of messages supported by the message predicate. (messages:message_t/1)

  Text is a string (a list of character codes). (basic_props:string/1)

show_message/3: PREDICATE

Usage: `show_message(Type,Text,ArgList)`

The text provided in Text is printed as a message of type Type, using the arguments in ArgList to interpret any variable-related formatting commands embedded in Text.

- *The following properties should hold at call time:*

  The types of messages supported by the message predicate. (messages:message_t/1)

  Text is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with name/2. (format:format_control/1)

  ArgList is a list. (basic_props:list/1)

Meta-predicate with arguments: `show_message(?,?,addmodule(?))`.

show_message/4: PREDICATE

Usage: `show_message(Type,Lc,Text,ArgList)`

The text provided in `Text` is printed as a message of type `Type`, using the arguments in `ArgList` to interpret any variable-related formatting commands embedded in `Text`, and reporting error location `Lc` (file and line numbers).

– *The following properties should hold at call time:*

The types of messages supported by the message predicate. (messages:message_t/1)

Identifies a source line range in a file.

```
location_t(loc(File,L1,L2)) :-
    atm(File),
    int(L1),
    int(L2).
```

(c_itf:location_t/1)

`Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (format:format_control/1)

`ArgList` is a list. (basic_props:list/1)

Meta-predicate with arguments: `show_message(?,?,?,addmodule(?))`.

message_t/1: REGTYPE

This type defines the types of messages supported by the message priting predicates.

```
message_t(error).
message_t(warning).
message_t(note).
message_t(simple).
message_t(debug).
```

Usage:

The types of messages supported by the message predicate.

error_message/1: PREDICATE

Usage: `error_message(Text)`

Same as `message(error,Text)`.

– *The following properties should hold at call time:*

`Text` is a string (a list of character codes). (basic_props:string/1)

error_message/2: PREDICATE

Usage: `error_message(Text,ArgList)`

Same as `message(error,Text,ArgList)`.

– *The following properties should hold at call time:*

`Text` is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with `name/2`. (format:format_control/1)

`ArgList` is a list. (basic_props:list/1)

Meta-predicate with arguments: `error_message(?,addmodule(?))`.

error_message/3: PREDICATE

Usage: `error_message(Lc,Text,ArgList)`

Same as `message(error,Lc,Text,ArgList)`.

– *The following properties should hold at call time:*

Identifies a source line range in a file.

```
location_t(loc(File,L1,L2)) :-
    atm(File),
    int(L1),
    int(L2).
```

(c_itf:location_t/1)

Text is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with name/2. (format:format_control/1)

ArgList is a list. (basic_props:list/1)

Meta-predicate with arguments: `error_message(?,?,addmodule(?))`.

warning_message/1: PREDICATE

Usage: `warning_message(Text)`

Same as `message(warning,Text)`.

– *The following properties should hold at call time:*

Text is a string (a list of character codes). (basic_props:string/1)

warning_message/2: PREDICATE

Usage: `warning_message(Text,ArgList)`

Same as `message(warning,Text,ArgList)`.

– *The following properties should hold at call time:*

Text is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with name/2. (format:format_control/1)

ArgList is a list. (basic_props:list/1)

Meta-predicate with arguments: `warning_message(?,addmodule(?))`.

warning_message/3: PREDICATE

Usage: `warning_message(Lc,Text,ArgList)`

Same as `message(warning,Lc,Text,ArgList)`.

– *The following properties should hold at call time:*

Identifies a source line range in a file.

```
location_t(loc(File,L1,L2)) :-
    atm(File),
    int(L1),
    int(L2).
```

(c_itf:location_t/1)

Text is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with name/2. (format:format_control/1)

ArgList is a list. (basic_props:list/1)

Meta-predicate with arguments: warning_message(?,?,addmodule(?)).

note_message/1: PREDICATE

Usage: note_message(Text)

Same as message(note,Text).

– *The following properties should hold at call time:*

Text is a string (a list of character codes). (basic_props:string/1)

note_message/2: PREDICATE

Usage: note_message(Text,ArgList)

Same as message(note,Text,ArgList).

– *The following properties should hold at call time:*

Text is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with name/2. (format:format_control/1)

ArgList is a list. (basic_props:list/1)

Meta-predicate with arguments: note_message(?,addmodule(?)).

note_message/3: PREDICATE

Usage: note_message(Lc,Text,ArgList)

Same as message(note,Lc,Text,ArgList).

– *The following properties should hold at call time:*

Identifies a source line range in a file.

```
location_t(loc(File,L1,L2)) :-
    atm(File),
    int(L1),
    int(L2).
```

(c_itf:location_t/1)

Text is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with name/2. (format:format_control/1)

ArgList is a list. (basic_props:list/1)

Meta-predicate with arguments: note_message(?,?,addmodule(?)).

simple_message/1: PREDICATE

Usage: `simple_message(Text)`

The text provided in Text is printed.

– *The following properties should hold at call time:*

Text is a string (a list of character codes). (basic_props:string/1)

simple_message/2: PREDICATE

Usage: `simple_message(Text,ArgList)`

The text provided in Text is printed as a message, using the arguments in ArgList.

– *The following properties should hold at call time:*

Text is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with name/2. (format:format_control/1)

ArgList is a list. (basic_props:list/1)

optional_message/2: PREDICATE

Usage: `optional_message(Text,Opts)`

The text provided in Text is printed as a message, but only if the atom -v is a member of Opts. These predicates are meant to be used for optional messages, which are only to be printed when *verbose* output is requested explicitly.

– *The following properties should hold at call time:*

Text is a string (a list of character codes). (basic_props:string/1)

Opts is a list of atms. (basic_props:list/2)

optional_message/3: PREDICATE

Usage: `optional_message(Text,ArgList,Opts)`

The text provided in Text is printed as a message, using the arguments in ArgList, but only if the atom -v is a member of Opts. These predicates are meant to be used for optional messages, which are only to be printed when *verbose* output is requested explicitly.

– *The following properties should hold at call time:*

Text is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with name/2. (format:format_control/1)

ArgList is a list. (basic_props:list/1)

Opts is a list of atms. (basic_props:list/2)

debug_message/1: PREDICATE

Usage: `debug_message(Text)`

The text provided in Text is printed as a debugging message. These messages are turned on by defining a fact of issue_debug_messages/1 with the module name as argument.

- *The following properties should hold at call time:*

  Text is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with name/2. (format:format_control/1)

debug_message/2: PREDICATE

Usage: debug_message(Text,ArgList)

The text provided in Text is printed as a debugging message, using the arguments in ArgList to interpret any variable-related formatting commands embedded in Text. These messages are turned on by defining a fact of issue_debug_messages/1 which the module name as argument.

- *The following properties should hold at call time:*

  Text is an atom or string describing how the arguments should be formatted. If it is an atom it will be converted into a string with name/2. (format:format_control/1)

  ArgList is a list. (basic_props:list/1)

Meta-predicate with arguments: debug_message(?,addmodule(?)).

location_t/1: (UNDOC_REEXPORT)

Imported from c_itf (see the corresponding documentation for details).

206.3 Documentation on multifiles

issue_debug_messages/1: PREDICATE

Usage: issue_debug_messages(Module)

Printing of debugging messages is enabled for module Module.

- *The following properties hold upon exit:*

  Module is an atom. (basic_props:atm/1)

The predicate is *multifile*.

The predicate is of type *data*.

206.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  datafacts_rt, lists, streams, write, format, pathnames, c_itf.
- *Packages:*

  prelude, initial, condcomp, assertions, assertions/assertions_basic, regtypes, isomodes, datafacts.

207 Interactive menus

Author(s): The Ciao Development Team.

This library package allows definition interactive menus

207.1 Usage and interface

- **Library usage:**

  `:- use_package(menu).`

  or

  `:- module(...,...,[menu]).`
- **New operators defined:**

  `::/2 [970,xfx]`, `<-/2 [971,xfx]`, `guard/1 [900,fy]`, `$/2 [150,xfx]`, `=>/2 [950,xfx]`, `argnames/1 [1150,fx]`.
- **Implicit imports:**
  - *System library modules:*

    `menu_generator`, `menu_rt`.
  - *Packages:*

    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `argnames`.

207.2 Documentation on multifiles

menu_default/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

menu_opt/6: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

hook_menu_flag_values/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

hook_menu_check_flag_value/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

hook_menu_flag_help/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

hook_menu_default_option/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

208 menu_generator (library)

208.1 Usage and interface

- **Library usage:**
  `:- use_module(library(menu/menu_generator)).`
- **Exports:**
  - *Predicates:*
    `menu/1`, `menu/2`, `menu/3`, `menu/4`, `exists_menu_flag/2`, `get_menu_flag/3`, `set_menu_flag/3`, `space/1`, `get_menu_configs/1`, `save_menu_config/1`, `remove_menu_config/1`, `restore_menu_config/1`, `show_menu_configs/0`, `show_menu_config/1`, `get_menu_flags/1`, `restore_menu_flags_list/1`, `get_menu_flags/2`, `restore_menu_flags/2`, `generate_offline_menu/2`, `eq/3`, `neq/3`, `uni_type/2`, `vmember/2`, `decomp_menu_node/3`, `comp_menu_node/3`.
  - *Regular Types:*
    `menu_flag_values/1`.
  - *Multifiles:*
    `$is_persistent/2`, `persistent_dir/2`, `persistent_dir/4`, `menu_default/3`, `menu_opt/6`, `hook_menu_flag_values/3`, `hook_menu_check_flag_value/3`, `hook_menu_flag_help/3`, `hook_menu_default_option/3`.

208.2 Documentation on exports

menu/1: PREDICATE

Usage: `menu(M)`

Like `menu(M,true)`.

menu/2: PREDICATE

Usage: `menu(M,Bool)`

Like menu/4 with no selected options, taking the menu level from the term M (example: ana(1) is expert, ana is naive), and using Bool to decide whether print help message or not.

menu/3: PREDICATE

Usage: `menu(M,Level,Bool)`

Like menu/4 with no selected options.

menu/4: PREDICATE

Usage: `menu(M,Level,Bool,AlreadySelectedOpts)`

Execute the menu X. Level specifies the menu level. Bool decides whether print the help message. AlreadySelectedOpts is a list with the selected options.

exists_menu_flag/2: PREDICATE

Usage: `exists_menu_flag(M,F)`

The flag F in the menu (-branch) M has a default value.

– *The following properties should hold at call time:*

M is an atom. (basic_props:atm/1)

F is an atom. (basic_props:atm/1)

get_menu_flag/3: PREDICATE

Usage: `get_menu_flag(M,F,V)`

Returns the value in V of the flag F in the menu (-branch) M.

– *The following properties should hold at call time:*

M is an atom. (basic_props:atm/1)

F is an atom. (basic_props:atm/1)

V is a free variable. (term_typing:var/1)

set_menu_flag/3: PREDICATE

Usage: `set_menu_flag(M,F,V)`

Set the value V of the flag F in the menu (-branch) M.

– *The following properties should hold at call time:*

M is an atom. (basic_props:atm/1)

F is an atom. (basic_props:atm/1)

V is a free variable. (term_typing:var/1)

space/1: PREDICATE

Usage: `space(N)`

prints N spaces.

– *The following properties should hold at call time:*

N is a number. (basic_props:num/1)

get_menu_configs/1: PREDICATE

Usage: `get_menu_configs(X)`

Returns a list of atoms in X with the name of stored configurations.

– *The following properties should hold at call time:*

X is a free variable. (term_typing:var/1)

– *The following properties should hold upon exit:*

X is a list of atms. (basic_props:list/2)

save_menu_config/1: PREDICATE

Usage: `save_menu_config(Name)`

Save the current flags configuration under the Name key.

– *The following properties should hold at call time:*

Name is an atom. (basic_props:atm/1)

remove_menu_config/1: PREDICATE

Usage: `remove_menu_config(Name)`

Remove the configuration stored with the Name key (the same provided in save_menu_config/1).

- *The following properties should hold at call time:*

  Name is an atom. (basic_props:atm/1)

restore_menu_config/1: PREDICATE

Usage: `restore_menu_config(Name)`

Restore the configuration saved with the Name key (the same provided in save_menu_config/1).

- *The following properties should hold at call time:*

  Name is an atom. (basic_props:atm/1)

show_menu_configs/0: PREDICATE

Usage:

Show all stored configurations.

show_menu_config/1: PREDICATE

Usage: `show_menu_config(C)`

Show specific configuration values pointed by C key (the same provided in save_menu_config/1).

- *The following properties should hold at call time:*

  C is an atom. (basic_props:atm/1)

get_menu_flags/1: PREDICATE

Usage: `get_menu_flags(L)`

Return a list L of all current menu flags, composed by terms with the form (M,F,V), where M is the menu, F the flag, and V the value. This list can be used as argument of restore_flags_list/1

- *The following properties should hold at call time:*

  L is a free variable. (term_typing:var/1)

- *The following properties should hold upon exit:*

  L is a list. (basic_props:list/1)

restore_menu_flags_list/1: PREDICATE

Usage: `restore_menu_flags_list(L)`

Restores menu flags. L is a list of tuple (M,F,V) where M is the menu, F is the flag, and V is the value of the flag F in the menu M.

- *The following properties should hold at call time:*

  L is a list. (basic_props:list/1)

get_menu_flags/2: PREDICATE

Usage: `get_menu_flags(M,L)`

Return a list `L` of the current menu `M` composed by terms with the form (`F=V`), `F` the flag, and `V` the value. This list can be used as argument of `restore_menu_flags/2`

- *The following properties should hold at call time:*

  `M` is any term. (basic_props:term/1)

  `L` is a free variable. (term_typing:var/1)

- *The following properties should hold upon exit:*

  `L` is a list. (basic_props:list/1)

restore_menu_flags/2: PREDICATE

Usage: `restore_menu_flags(M,F)`

Restore the flag of the menu `M`. `F` is a list of terms `F=V`, which indicate the flag (`F`) and the value (`V`). `M` is the target menu to which those flags "belong". Additionally, `F` can contains terms like `changed_to_menu(NM)` that will put `NM` as the new target menu.

- *The following properties should hold at call time:*

  `M` is an atom. (basic_props:atm/1)

  `F` is a list. (basic_props:list/1)

generate_offline_menu/2: PREDICATE

Usage: `generate_offline_menu(EntryMenu,MenuItems)`

Generate an offline menu `MenuItems` for the given `EntryMenu`

- *The following properties should hold at call time:*

  `EntryMenu` is any term. (basic_props:term/1)

  `MenuItems` is any term. (basic_props:term/1)

eq/3: PREDICATE

Usage: `eq(Type,A,B)`

`Type` is the value returned by the 2nd arg of `uni_type`. `A` and `B` are whatever terms. This predicate success if they are equal (like `A=B`).

neq/3: PREDICATE

Usage: `neq(Type,A,B)`

`Type` is the value returned by the 2nd arg of `uni_type`. `A` and `B` are whatever terms. The semantic is similar to `A == B`.

uni_type/2: PREDICATE

Usage: `uni_type(Var,Type)`

`Var` should be the argument passed to the menu guard. `Type` is an abstract type that decides how unifications should be done in `eq/3` and `neq/3`.

vmember/2: PREDICATE

Usage: `vmember(Var,List)`

It is member equivalent predicate to be used in guards.

menu_flag_values/1: REGTYPE

Usage: `menu_flag_values(X)`

Flag values

decomp_menu_node/3: PREDICATE

No further documentation available for this predicate.

comp_menu_node/3: PREDICATE

No further documentation available for this predicate.

208.3 Documentation on multifiles

$is_persistent/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

The predicate is of type *data*.

persistent_dir/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

The predicate is of type *data*.

persistent_dir/4: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

The predicate is of type *data*.

menu_default/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

menu_opt/6: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

hook_menu_flag_values/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

hook_menu_check_flag_value/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

hook_menu_flag_help/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

hook_menu_default_option/3: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

208.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  datafacts_rt, persdb_rt, aggregates, streams, write, messages, prompt, lists, pathnames.
- *Packages:*

  prelude, initial, condcomp, hiord, assertions, assertions/assertions_basic, regtypes, argnames, persdb, datafacts, persdb/persdb_decl, nortchecks.

209 Parse and return command-line options

Author(s): Manuel Carro.

209.1 Usage and interface

- **Library usage:**
  `:- use_module(library(getopts)).`
- **Exports:**
  - *Predicates:*
    `getopts/4`, `cl_option/2`.

209.2 Documentation on exports

getopts/4: PREDICATE

Usage: `getopts(Arguments,Opts,Matched,Rest)`

Ciao Prolog parses the command-line arguments of its executables and passes them as a list of atoms to the main/1 predicate. Thus, a shell invocation such as

```
./my_program -file input.txt -file input2.txt --output_file out.txt
-create-dir --decode --unsorte
```

makes main/1 receive an argument such as

```
['-file', 'input.txt', '-file', 'input2.txt', '--output_file',
'out.txt', '-create-dir', '--decode', '--unsorte']
```

getopts/4 can be used to parse such an command-line option list. passed in the Arguments parameter. Opts is a list of expected options, each option being an *option spec*, i.e., a term of the form atom/arity. For every atom a command-line option of the form '--atom' or '-atom' is expected, with arity arguments following it. An arity of zero can be omitted. For each matched option spec, the list Matched will contain a term of the form atom(Arg1, Arg2, ..., Argn), where n = arity. The list Rest will contain the unmatched element in Arguments.

Rest will respect the relative order of the elements in Arguments. The matching elements in Matched appear in the same order as the options in Opts, and for every option in Opts, its matches appear in the order as they came in Arguments.

Assuming Arguments is ['-file', 'input.txt', '-file', 'input2.txt', '--output_file', 'out.txt', '-create-dir', '--decode', '--unsorte'], some possible uses of getopts/4 follow.

- Check that a simple option has been selected:

```
?- getopts(Args, ['create-dir'], M, R).
Args = ...
M = ['create-dir'],
R = ['-file','input.txt','-file','input2.txt','--output_
file',
     'out.txt','--decode','--unsorte']
```

- Which argument was given to an option expecting an additional value?

```
1 ?- getopts(Args, [output_file/1], M, R).
Args = ...
M = [output_file('out.txt')],
R = ['-file','input.txt','-file','input2.txt','-create-dir',
     '--decode','--unsorte']

1 ?- getopts(Args, [output_file/1], [output_file(F)], R).
Args = ..
F = 'out.txt',
R = ['-file','input.txt','-file','input2.txt','-create-dir',
     '--decode','--unsorte']
```

- Extract options (and associated values) which can appear several times.

```
1 ?- getopts(Args, [file/1], M, R).
Args = ...
M = [file('input.txt'),file('input2.txt')],
R = ['--output_file','out.txt','-create-dir','--decode',
     '--unsorte']
```

- Was decoding selected?

```
1 ?- getopts(Args, [decode], [_], R).
Args = ...
R = ['-file','input.txt','-file','input2.txt','--output_
file',
     'out.txt', '-create-dir','--unsorte']
```

- Was encoding selected?

```
1 ?- getopts(Args, [encoding], [_], R).
no
```

- Was decoding **not** selected?

```
1 ?- getopts(Args, [decode], [], R).
no
```

- Are all the options passed to the program legal options? If this is not the case, which option(s) is/are not legal?

```
1 ?- getopts(Args, [file/1, output_file/1, 'create-dir',
                encode, decode, unsorted], _, R).
Args = ...
R = ['--unsorte'] ?
```

The complexity of getopts/1 is currently *O(La x Lo)*, where *La* is the length of the argument list and *Lo* is the length of the option list.

- *Call and exit should be compatible with:*

  Arguments is a list of atoms. (basic_props:list/2)

  Opts is a list of specs. (basic_props:list/2)

  Matched is a list of terms. (basic_props:list/2)

  Rest is a list of terms. (basic_props:list/2)

- *The following properties should hold at call time:*

  Arguments is currently a term which is not a free variable. (term_typing:nonvar/1)

  Opts is currently a term which is not a free variable. (term_typing:nonvar/1)

cl_option/2: PREDICATE

Usage: `cl_option(Arguments,Option)`

Check that Option is an option in Arguments.

- *Call and exit should be compatible with:*

  Arguments is a list of atoms. (basic_props:list/2)

  Option is AtomName/Arity (getopts:spec/1)
- *The following properties should hold at call time:*

  Arguments is currently a term which is not a free variable. (term_typing:nonvar/1)

  Option is currently a term which is not a free variable. (term_typing:nonvar/1)

209.3 Documentation on internals

spec/1: REGTYPE

Usage: `spec(Spec)`

Spec is AtomName/Arity

209.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  lists.
- *Packages:*

  prelude, initial, condcomp, assertions, assertions/assertions_basic, basicmodes, regtypes.

210 Additional operating system utilities

Author(s): Manuel Hermenegildo, Jose F. Morales.

This module groups some extensions to library system that have been found convenient, but which are still in development, their interface has not been fixed, etc.

210.1 Usage and interface

- **Library usage:**
  `:- use_module(library(system_extra)).`
- **Exports:**
  - *Predicates:*
    `warn_on_nosuccess/1`, `ignore_nosuccess/1`, `del_dir_if_empty/1`, `empty_dir/1`, `move_files/2`, `move_file/2`, `copy_files/2`, `copy_files/3`, `copy_files_nofail/3`, `del_file_nofail/1`, `del_files_nofail/1`, `file_to_line/2`, `replace_strings_in_file/3`, `backup_file/1`, `move_if_diff/3`, `set_file_owner/2`, `set_file_perms/2`, `set_exec_perms/2`, `mkpath/1`, `mkpath/2`, `mkpath/3`, `mkpath_mode/3`, `mkpath_mode/2`, `mkpath_mode/1`, `mktempdir_in_tmp/2`, `rmtempdir/1`, `create_rel_link/2`, `create_link/2`, `relpath/3`, `istty/1`, `datime_atom/1`, `datime_atom/2`, `datime_string/1`, `datime_string/2`, `datime_to_string/2`, `replace_strings/3`.
  - *Regular Types:*
    `perms_term/1`, `valid_mode/1`.

210.2 Documentation on exports

warn_on_nosuccess/1: PREDICATE

Usage: `warn_on_nosuccess(G)`

Call G (cutting solutions, i.e., as once/1) and show warning messages if something went wrong (failure and exceptions).

Meta-predicate with arguments: `warn_on_nosuccess(goal)`.

ignore_nosuccess/1: PREDICATE

Usage: `ignore_nosuccess(G)`

Call G and ignore if something went wrong (failure and exceptions).

Meta-predicate with arguments: `ignore_nosuccess(goal)`.

del_dir_if_empty/1: PREDICATE

Usage: `del_dir_if_empty(D)`

Delete D if it is an empty directory.

- *The following properties should hold at call time:*
  D is an atom. (basic_props:atm/1)

empty_dir/1: PREDICATE

Usage: `empty_dir(D)`

`D` is an empty directory

– *The following properties should hold at call time:*

`D` is an atom. (basic_props:atm/1)

move_files/2: PREDICATE

Usage: `move_files(Files,Dir)`

Move `Files` to directory `Dir` (note that to move only one file to a directory, `rename_file/2` can be used).

– *The following properties should hold at call time:*

`Files` is a list of `atm`s. (basic_props:list/2)

`Dir` is an atom. (basic_props:atm/1)

move_file/2: PREDICATE

Usage: `move_file(File,Dir)`

Move `File` to directory `Dir`

– *The following properties should hold at call time:*

`File` is an atom. (basic_props:atm/1)

`Dir` is an atom. (basic_props:atm/1)

copy_files/2: PREDICATE

Usage: `copy_files(Files,Dir)`

Like `copy_files/3`, with empty options list.

– *The following properties should hold at call time:*

`Files` is a list of `atm`s. (basic_props:list/2)

`Dir` is an atom. (basic_props:atm/1)

copy_files/3: PREDICATE

Usage: `copy_files(Files,Dir,Opts)`

Copy `Files` to directory `Dir`, using `Opts` as the option list for copy. See `copy_file/3` for the list of options. Note that to move only one file to a directory, `rename_file/2` can be used.

– *The following properties should hold at call time:*

`Files` is a list of `atm`s. (basic_props:list/2)

`Dir` is an atom. (basic_props:atm/1)

`Opts` is a list of file copy options. (system:copy_options/1)

copy_files_nofail/3: PREDICATE

Usage: `copy_files_nofail(Files,Dir,Opts)`

Like `copy_files/3`, but do not fail in case of errors.

– *The following properties should hold at call time:*

`Files` is a list of `atm`s. (`basic_props:list/2`)

`Dir` is an atom. (`basic_props:atm/1`)

`Opts` is a list of file copy options. (`system:copy_options/1`)

del_file_nofail/1: PREDICATE

Usage: `del_file_nofail(File)`

Like `delete_file/1`, but do not fail in case of errors.

– *The following properties should hold at call time:*

`File` is an atom. (`basic_props:atm/1`)

del_files_nofail/1: PREDICATE

Usage: `del_files_nofail(Files)`

Like `del_file_nofail/1`, but takes list of files in `Files`.

– *The following properties should hold at call time:*

`Files` is a list of `atm`s. (`basic_props:list/2`)

file_to_line/2: PREDICATE

Usage: `file_to_line(File,Str)`

– *The following properties should hold at call time:*

`File` is a list of `atm`s. (`basic_props:list/2`)

`Str` is a string (a list of character codes). (`basic_props:string/1`)

replace_strings_in_file/3: PREDICATE

Usage: `replace_strings_in_file(Ss,F1,F2)`

Like `replace_strings/3` but from file `F1` to file `F2`.

– *The following properties should hold at call time:*

`Ss` is a list of `string`s. (`basic_props:list/2`)

`F1` is an atom. (`basic_props:atm/1`)

`F2` is an atom. (`basic_props:atm/1`)

backup_file/1: PREDICATE

Usage: `backup_file(FileName)`

Save a backup copy of file `FileName`

– *The following properties should hold at call time:*

`FileName` is an atom. (`basic_props:atm/1`)

move_if_diff/3: PREDICATE

Usage: `move_if_diff(From,To,NewOrOld)`

If To does not exist, or its contents are different from From, delete To and rename From to To. NewOrOld is unified with new or old depending on whether the new or the old file is preserved.

- *The following properties should hold at call time:*

  From is an atom. (basic_props:atm/1)

  To is an atom. (basic_props:atm/1)
- *The following properties should hold upon exit:*

  NewOrOld is an atom. (basic_props:atm/1)

set_file_owner/2: PREDICATE

Usage: `set_file_owner(File,Owner)`

Set user/group of a file.

- *The following properties should hold at call time:*

  File is currently a term which is not a free variable. (term_typing:nonvar/1)

  Owner is currently a term which is not a free variable. (term_typing:nonvar/1)

set_file_perms/2: PREDICATE

Usage: `set_file_perms(File,Perms)`

Set permissions of File to Perms.

- *The following properties should hold at call time:*

  Perms is a term providing modes for User, Group, and Others. (system_extra:perms_term/1)

  File is an atom. (basic_props:atm/1)

perms_term/1: REGTYPE

`perms_term(Perms)`

Perms is a term providing valid permissions (‘‘modes’’) for User, Group, and Others. These are all valid_mode/1s. Defined as follows:

```
perms_term(perms(U,G,O)) :-
    valid_mode(U),
    valid_mode(G),
    valid_mode(O).
```

Usage: `perms_term(Perms)`

Perms is a term providing modes for User, Group, and Others.

valid_mode/1: REGTYPE

`valid_mode(Mode)`

Mode is an atom that provides a valid set of file permissions (a valid ‘‘mode’’). Defined as follows:

```
valid_mode('').
valid_mode('X').
valid_mode(x).
valid_mode(w).
valid_mode(wX).
valid_mode(wx).
valid_mode(r).
valid_mode(rX).
valid_mode(rx).
valid_mode(rw).
valid_mode(rwX).
valid_mode(rwx).
```

Usage: `valid_mode(Mode)`

`Mode` is a file permissions mode.

set_exec_perms/2: PREDICATE

Usage: `set_exec_perms(File,Perms)`

Set file permissions, but treat regular files as directories w.r.t. 'X' flag

mkpath/1: PREDICATE

Usage: `mkpath(Path)`

Creates the directories necessary to access the given `Path` (which can be absolute or relative).

– *The following properties should hold at call time:*

  `Path` is a source name. (stream_basic:sourcename/1)

mkpath/2: PREDICATE

Usage: `mkpath(Path,Perms)`

Like `mkpath`, but sets permissions of new directories as `Perms`.

– *The following properties should hold at call time:*

  `Path` is a source name. (stream_basic:sourcename/1)

  `Perms` is any term. (basic_props:term/1)

mkpath/3: PREDICATE

Usage: `mkpath(Path,Perms,Owner)`

Like `mkpath`, but sets permissions and owner of new directories to `Perms` and `Owner`.

– *The following properties should hold at call time:*

  `Path` is a source name. (stream_basic:sourcename/1)

  `Perms` is any term. (basic_props:term/1)

  `Owner` is any term. (basic_props:term/1)

mkpath_mode/3: PREDICATE

No further documentation available for this predicate.

mkpath_mode/2: PREDICATE

Usage:

Equivalent to `mkpath_mode(Path,Mode,_)`.

– *The following properties should hold at call time:*

  `Arg1` is a source name. (stream_basic:sourcename/1)

  `Arg2` is an integer. (basic_props:int/1)

mkpath_mode/1: PREDICATE

Usage:

Equivalent to `mkpath_mode(Path,0o777,_)`.

– *The following properties should hold at call time:*

  `Arg1` is a source name. (stream_basic:sourcename/1)

mktempdir_in_tmp/2: PREDICATE

Usage: `mktempdir_in_tmp(Template,Path)`

Create a directory in the temporary directory using Template (see mktemp_in_tmp/2). An empty file CREATED_WITH_MKTEMPDIR is created inside Path as a safety check for rmtempdir/1.

rmtempdir/1: PREDICATE

Usage: `rmtempdir(Path)`

Remove the temporary directory Path (recursively) created with mktempdir_in_tmp/2. As a safety check, this predicate throws an exception if the CREATED_WITH_MKTEMPDIR file is not in Path.

create_rel_link/2: PREDICATE

Usage: `create_rel_link(From,To)`

Create a *relocatable* symlink (computing relative paths) (e.g., /a/b/c (symlink) -> /a/d/e becomes /a/b/c (symlink) -> ../d/e

create_link/2: PREDICATE

Usage: `create_link(From,To)`

Create a symlink from From to To. On platforms where symlinks are not supported (Windows) the file is copied instead. The file To is removed if it existed before.

relpath/3: PREDICATE

Usage: `relpath(A,B,C)`

C is a path to B relative to A (using .. if needed) (e.g., /a/b/c -> /a/d/e becomes /a/b/c -> ../../d/e. Assume both are absolute, otherwise just return B.

istty/1: PREDICATE

Usage: `istty(FD)`

Check if the file descriptor is associated with a terminal.

datime_atom/1: PREDICATE

No further documentation available for this predicate.

datime_atom/2: PREDICATE

No further documentation available for this predicate.

datime_string/1: PREDICATE

No further documentation available for this predicate.

datime_string/2: PREDICATE

No further documentation available for this predicate.

datime_to_string/2: PREDICATE

No further documentation available for this predicate.

replace_strings/3: PREDICATE

No further documentation available for this predicate.

210.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  regexp_code, system, pathnames, messages, terms, lists, stream_utils, process, port_reify, source_tree.
- *Packages:*

  prelude, initial, condcomp, assertions, assertions/assertions_basic, regtypes, isomodes, hiord, regexp.

211 Shell-style argument parsing

Author(s): Jose F. Morales.

This module implements shell-style argument parsing. This is useful for parsing argument strings from other tools (e.g., those returned by pkg-config) and using them in calls to process_call/3 (without using a full shell interpreter).

It offers a parse_shell_args/2 predicate that splits a single input into a list of arguments, following these rules:

- arguments are separated by blanks (whitespaces or tabulators)
- arguments may contain double and single quoted substrings (quotes are removed)
- blanks are not separators when they appear in quoted strings
- escape character removes special meaning from blank or quote characters

Note that this library is not intended to do full shell parsing (command or variable substitutions, etc.).

211.1 Usage and interface

- **Library usage:**
  `:- use_module(library(parse_shell_args)).`
- **Exports:**
  - *Predicates:*
    parse_shell_args/2.

211.2 Documentation on exports

parse_shell_args/2: PREDICATE

Usage: parse_shell_args(Atm,Args)

Parse shell-style arguments from atom Atm into Args

- *Call and exit should be compatible with:*
  Atm is an atom. (basic_props:atm/1)
  Args is a list of atms. (basic_props:list/2)
- *The following properties should hold at call time:*
  Atm is currently a term which is not a free variable. (term_typing:nonvar/1)
  Args is a free variable. (term_typing:var/1)

211.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  prelude, initial, condcomp, assertions, assertions/assertions_basic, dcg, regtypes, isomodes.

212 File archiver

Author(s): The Ciao Development Team, Jose F. Morales.

This module offers predicates to generate file archives in some common formats. An archive file is a file that is composed of one or more computer files along with metadata, with can be compressed. See archive_files/4 for the supported formats.

212.1 Usage and interface

- **Library usage:**
  `:- use_module(library(archive_files)).`
- **Exports:**
  - *Predicates:*
    archive_files/4.

212.2 Documentation on exports

archive_files/4: PREDICATE

Usage: archive_files(SourceDir,Files,TopDir,Archive)

Create an archive Archive of the given files at Files, relative to SourceDir. If TopDir is not '', files are placed inside the archive under TopDir.

The format is automatically detected from the extension of Archive (where valid formats are: .tar.gz, .tgz, .tar.bz2, .tbz, and .zip).

- *Call and exit should be compatible with:*
  SourceDir is an atom. (basic_props:atm/1)
  Files is a list of atms. (basic_props:list/2)
  TopDir is an atom. (basic_props:atm/1)
  Archive is an atom. (basic_props:atm/1)

212.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  native_props, terms, process, pathnames, system, system_extra, port_reify, bundle_flags, stream_utils.
- *Packages:*
  prelude, initial, condcomp, assertions, assertions/assertions_basic, basicmodes, nativeprops, regtypes, fsyntax.

213 File locks

Author(s): José Manuel Gómez Pérez, Daniel Cabeza, Manuel Carro.

This module implements file locks: the ability to lock a file so that other processes cannot access it until the file is unlocked.

NOT IMPLEMENTED. Consider locking in open/3.

213.1 Usage and interface

- **Library usage:**
  `:- use_module(library(file_locks)).`
- **Exports:**
  - *Predicates:*
    lock_file/3, unlock_file/2.

213.2 Documentation on exports

lock_file/3: PREDICATE

Usage: lock_file(File,LockType,Result)

Tries to lock File with LockType and returns the result (either true or false) in Result.

- *Call and exit should be compatible with:*
  File is an atom. (basic_props:atm/1)
  LockType is an atom. (basic_props:atm/1)
  Result is an atom. (basic_props:atm/1)

unlock_file/2: PREDICATE

Usage: unlock_file(File,Result)

Tries to unlock File the result (either true or false) in Result.

- *Call and exit should be compatible with:*
  File is an atom. (basic_props:atm/1)
  Result is an atom. (basic_props:atm/1)

213.3 Documentation on imports

This module has the following direct dependencies:

- *Packages:*
  prelude, initial, condcomp, assertions, assertions/assertions_basic.

214 Symbolic filenames

Author(s): Francisco Bueno.

This module provides a predicate for file opening which can use any term as an alias for the filename (i.e., symbolic filenames) instead of the usual constants which are file system path names of the actual files.

The correspondence between an alias and the actual file path is done dynamically, without having to recompile the program. It is possible to define the correspondence via facts for file_alias/2 in a file declared with multifile:alias_file/1 in the program: those facts will be dynamically loaded when running the program. Alternatively, the correspondence can be defined via shell environment variables, by defining the value of a variable by the (symbolic) name of the file to be the path of the actual file.

214.1 Usage and interface

- **Library usage:**
  :- use_module(library(symfnames)).
- **Exports:**
  - *Predicates:*
    open/3.
  - *Multifiles:*
    alias_file/1, file_alias/2.

214.2 Documentation on exports

open/3: PREDICATE

open(File,Mode,Stream)

Open File with mode Mode and return in Stream the stream associated with the file. It is like stream_basic:open/3, but File is considered a symbolic name: either defined by user:file_alias/2 or as an environment variable. Predicate user:file_alias/2 is inspected before the environment variables.

Usage:

- *Call and exit should be compatible with:*
  Stream is an open stream. (stream_basic:stream/1)
- *The following properties should hold at call time:*
  File is any term. (basic_props:term/1)
  Mode is an opening mode ('read', 'write' or 'append'). (stream_basic:io_mode/1)
- *The following properties should hold upon exit:*
  Stream is an open stream. (stream_basic:stream/1)

214.3 Documentation on multifiles

alias_file/1: PREDICATE

`alias_file(File)`

Declares File to be a file defining symbolic names via file_alias/2. Anything else in File is simply ignored. The predicate is *multifile*.

file_alias/2: PREDICATE

`file_alias(Alias,File)`

Declares Alias as a symbolic name for File, the real name of an actual file (or directory). The predicate is *multifile*.

The predicate is of type *data*.

214.4 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  datafacts_rt, read, system.
- *Packages:*
  prelude, initial, condcomp, assertions, assertions/assertions_basic, isomodes, datafacts.

214.5 Other information

The example discussed here is included in the distribution files. There is a main application file which uses module mm. This module reads a line from a file; the main predicate in the main file then prints this line. The important thing is that the file read is named by a symbolic name "file". The main application file declares another file where the symbolic names are assigned actual file names:

```
:- use_module(mm).

:- multifile alias_file/1.
alias_file(myfiles).

main :- p(X), display(X), nl.
```

Now, the file myfiles.pl can be used to change the file you want to read from without having to recompile the application. The current assignment is:

```
%:- use_package([]).
file_alias(file,'mm.pl').
```

so the execution of the application will show the first line of mm.pl. However, you can change to:

```
file_alias(file,'main.pl').
```

and then execution of the same executable will show the first line of main.pl.

215 Open a document with an external application

Author(s): The Ciao Development Team.

This module provides predicates to open documents with the user's preferred external applications. It is based on external commands typically provided by each operating system (e.g., xdg-open (Linux), cygstart (Windows), open (macOS)).

215.1 Usage and interface

- **Library usage:**
  `:- use_module(library(opendoc)).`
- **Exports:**
  - *Predicates:*
    `opendoc/1`, `opendoc/2`.

215.2 Documentation on exports

opendoc/1: PREDICATE

Usage: `opendoc(Target)`

Opens Target (a file or URL) with the user's preferred application

- *Call and exit should be compatible with:*
  Target is any term. (basic_props:term/1)
- *The following properties should hold at call time:*
  Target is currently a term which is not a free variable. (term_typing:nonvar/1)

opendoc/2: PREDICATE

Usage: `opendoc(Target,Opts)`

Opens Target (a file or URL) with the user's preferred application. Opts is a subset of [generic, in_emacs] (generic is the default)

- *Call and exit should be compatible with:*
  Target is any term. (basic_props:term/1)
  Opts is a list. (basic_props:list/1)
- *The following properties should hold at call time:*
  Target is currently a term which is not a free variable. (term_typing:nonvar/1)
  Opts is currently a term which is not a free variable. (term_typing:nonvar/1)

215.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  pathnames, process, terms, lists.
- *Packages:*

  prelude, initial, condcomp, assertions, assertions/assertions_basic, regtypes, dcg, basicmodes, fsyntax.

216 Calling emacs from Prolog

Author(s): The Ciao Development Team.

This library provides a *prolog-emacs interface*. This interface is complementary to (and independent from) the emacs mode, which is used to develop programs from within the emacs editor/environment. Instead, this library allows calling emacs from a running Prolog program. This facilitates the use of emacs as a ‘‘user interface’’ for a Prolog program. Emacs can be made to:

- Visit a file, which can then be edited.
- Execute arbitrary *emacs lisp* code, sent from Prolog.

In order for this library to work correctly, the following is needed:

- You should be running the emacs editor on the same machine where the executable calling this library is executing.
- This emacs should be running the *emacs server*. This can be done by including the following line in your .emacs file:

  ```
  ;; Start a server that emacsclient can connect to.
  (server-start)
  ```

  Or typing M-x server-start within emacs.

Examples:

Assuming that a .pl file loads this library, then:

..., emacs_edit('foo'), ...
    Opens file foo for editing in emacs.

..., emacs_eval_nowait("(run-ciao-toplevel)"), ...
    Starts execution of a Ciao top-level within emacs.

216.1 Usage and interface

- **Library usage:**
  :- use_module(library(emacs)).
- **Exports:**
  - *Predicates:*
    emacs_edit/1, emacs_edit_nowait/1, emacs_eval/1, emacs_eval_nowait/1.
  - *Regular Types:*
    elisp_string/1.

216.2 Documentation on exports

emacs_edit/1: PREDICATE

Usage:

Opens the given file for editing in emacs. Waits for editing to finish before continuing.

- *The following properties should hold at call time:*
  Arg1 is an atom which is the name of a file. (emacs:filename/1)

emacs_edit_nowait/1: PREDICATE

Usage:

Opens the given file for editing in emacs and continues without waiting for editing to finish.

- *The following properties should hold at call time:*

  Arg1 is an atom which is the name of a file. (emacs:filename/1)

emacs_eval/1: PREDICATE

Usage:

Executes in emacs the lisp code given as argument. Waits for the command to finish before continuing.

- *The following properties should hold at call time:*

  Arg1 is a string containing emacs lisp code. (emacs:elisp_string/1)

emacs_eval_nowait/1: PREDICATE

Usage:

Executes in emacs the lisp code given as argument and continues without waiting for it to finish.

- *The following properties should hold at call time:*

  Arg1 is a string containing emacs lisp code. (emacs:elisp_string/1)

elisp_string/1: REGTYPE

Usage: elisp_string(L)

L is a string containing emacs lisp code.

216.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*

  terms_check, lists, system, process.

- *Packages:*

  prelude, initial, condcomp, assertions, assertions/assertions_basic, regtypes, isomodes, fsyntax, hiord.

References

[AAF91] J. Almgren, S. Andersson, L. Flood, C. Frisk, H. Nilsson, and J. Sundberg.
Sicstus Prolog Library Manual.
Po Box 1263, S-16313 Spanga, Sweden, October 1991.

[AC19a] Joaquín Arias and Manuel Carro.
Description, Implementation, and Evaluation of a Generic Design for Tabled CLP.
Theory and Practice of Logic Programming, 19(3):412--448, 2019.

[AC19b] Joaquín Arias and Manuel Carro.
Evaluation of the Implementation of an Abstract Interpretation Algorithm using Tabled CLP.
Theory and Practice of Logic Programming, 19(5-6):1107--1123, September 2019.

[AC19c] Joaquín Arias and Manuel Carro.
Incremental evaluation of lattice-based aggregates in logic programming using modular TCLP.
In *21st Intl. Symposium on Practical Aspects of Declarative Languages*, pages 98--114, January 2019.

[AKNL86] Hassan Ait-Kaci, Roger Nasr, and Pat Lincoln.
E An Overview.
Technical Report AI-420-86-P, Microelectronics and Computer Technology Corporation, 9430 Research Boulevard, Austin, TX 78759, December 1986.

[AKPS92] H. Aït-Kaci, A. Podelski, and G. Smolka.
A feature-based constraint system for logic programming with entailment.
In *Proc. Fifth Generation Computer Systems 1992*, pages 1012--1021, 1992.

[Apt97] K. Apt, editor.
From Logic Programming to Prolog.
Prentice-Hall, Hemel Hempstead, Hertfordshire, England, 1997.

[BA82] M. Ben-Ari.
Principles of Concurrent Programming.
Prentice Hall International, 1982.

[BBP81] D.L. Bowen, L. Byrd, L.M. Pereira, F.C.N. Pereira, and D.H.D. Warren.
Decsystem-10 prolog user's manual.
Technical report, Department of Artificial Intelligence, University of Edinburgh, October 1981.

[BCHP96] F. Bueno, D. Cabeza, M. V. Hermenegildo, and G. Puebla.
Global Analysis of Standard Prolog Programs.
In *European Symposium on Programming*, number 1058 in LNCS, pages 108--124, Sweden, April 1996. Springer-Verlag.

[BGH99] F. Bueno, M. García de la Banda, and M. V. Hermenegildo.
Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming.
ACM Transactions on Programming Languages and Systems, 21(2):189--238, March 1999.

[**BLGH04**] F. Bueno, P. López-García, and M. V. Hermenegildo.
Multivariant Non-Failure Analysis via Standard Abstract Interpretation.
In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100--116, Heidelberg, Germany, April 2004. Springer-Verlag.

[**BLGPH04**]
F. Bueno, P. López-García, G. Puebla, and M. V. Hermenegildo.
The Ciao Prolog Preprocessor.
Technical Report CLIP1/04, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2004.

[**Bue95**] F. Bueno.
The CIAO Multiparadigm Compiler: A User's Manual.
Technical Report CLIP8/95.0, Facultad de Informática, UPM, June 1995.

[**Byr80**] L. Byrd.
Understanding the Control Flow of Prolog Programs.
In S.-A. Tärnlund, editor, *Proceedings of the 1980 Logic Programming Workshop*, pages 127--138, Debrecen, Hungary, July 1980.

[**Cab04**] D. Cabeza.
An Extensible, Global Analysis Friendly Logic Programming System.
PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.

[**Car87**] M. Carlsson.
Freeze, Indexing, and Other Implementation Issues in the Wam.
In *Fourth International Conference on Logic Programming*, pages 40--58. University of Melbourne, MIT Press, May 1987.

[**Car88**] M. Carlsson.
Sicstus Prolog User's Manual.
Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[**CCH06**] A. Casas, D. Cabeza, and M. V. Hermenegildo.
A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems.
In *The 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, pages 142--162, Fuji Susono (Japan), April 2006.

[**CD96**] Philippe Codognet and Daniel Diaz.
Compiling constraints in clp(fd).
J. Log. Program., 27(3):185--226, 1996.

[**CGH93**] M. Carro, L. Gómez, and M. Hermenegildo.
Some Paradigms for Visualizing Parallel Execution of Logic Programs.
In *1993 International Conference on Logic Programming*, pages 184--201. MIT Press, June 1993.

[**CH95**] D. Cabeza and M. V. Hermenegildo.
Distributed Concurrent Constraint Execution in the CIAO System.
In *Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid.
Available from http://www.cliplab.org/.

[**CH99a**] D. Cabeza and M. V. Hermenegildo.
Higher-order Logic Programming in Ciao.
Technical Report CLIP7/99.0, Facultad de Informática, UPM, September 1999.

[**CH99b**] D. Cabeza and M. V. Hermenegildo.
The Ciao Modular Compiler and Its Generic Program Processing Library.
In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*, pages 147--164. N.M. State U., December 1999.

[**CH00a**] D. Cabeza and M. V. Hermenegildo.
A New Module System for Prolog.
In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131--148. Springer-Verlag, July 2000.

[**CH00b**] D. Cabeza and M. V. Hermenegildo.
The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library.
In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[**CH00c**] M. Carro and M. V. Hermenegildo.
Tools for Constraint Visualization: The VIFID/TRIFID Tool.
In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 253--272. Springer-Verlag, September 2000.

[**CH00d**] M. Carro and M. V. Hermenegildo.
Tools for Search Tree Visualization: The APT Tool.
In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 237--252. Springer-Verlag, September 2000.

[**CH01**] D. Cabeza and M. Hermenegildo.
Distributed WWW Programming using (Ciao) Prolog and the PiLLoW Library.
Theory and Practice of Logic Programming, 1(3):251--282, May 2001.

[**CHGT98**] D. Cabeza, M. V. Hermenegildo, S. Genaim, and C. Taboch.
Design of a Generic, Homogeneous Interface to Relational Databases.
Technical Report D3.1.M1-A1, CLIP7/98.0, RADIOWEB Project, September 1998.

[**CHL04**] D. Cabeza, M. V. Hermenegildo, and J. Lipton.
Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction.
In *Ninth Asian Computing Science Conference (ASIAN'04)*, number 3321 in LNCS, pages 93--108. Springer-Verlag, December 2004.

[**CHV96**] D. Cabeza, M. V. Hermenegildo, and S. Varma.
The PiLLoW/Ciao Library for INTERNET/WWW Programming using Computational Logic Systems.
In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, pages 72--90, JICSLP'96, Bonn, September 1996.
Text and code available from
http://www.cliplab.org/miscdocs/pillow/pillow.html
(http://www.cliplab.org/miscdocs/pillow/pillow.html).

[CM81] W.F. Clocksin and C.S. Mellish.
Programming in Prolog.
Springer-Verlag, 1981.

[Col78] A. Colmerauer.
Metamorphosis grammars.
In *Natural language communication with computers*, pages 133--189.
Springer LNCS 63, 1978.

[Col82] A. Colmerauer et al.
Prolog II: Reference Manual and Theoretical Model.
Groupe D'intelligence Artificielle, Faculté Des Sciences De Luminy, Marseille, 1982.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni.
Prolog: The Standard.
Springer-Verlag, 1996.

[dG12] P. Chico de Guzmán.
Advanced Evaluation Strategies for Tabling and Parallelism in Logic Programs.
PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, November 2012.

[dGCHS12]
P. Chico de Guzmán, M. Carro, M. V. Hermenegildo, and P. Stuckey.
A general implementation framework for tabled clp.
In Tom Schrijvers and Peter Thiemann, editors, *15th Symposium on Functional and Logic Programming*, volume 7294 of *LNCS*, pages 104--119, May 2012.

[Dij65] E.W. Dijkstra.
Co-operating sequential processes.
In F. Genuys, editor, *Programming Languages*. Academic Press, London, 1965.

[DL93] S. K. Debray and N. W. Lin.
Cost Analysis of Logic Programs.
ACM Transactions on Programming Languages and Systems, 15(5):826--875, November 1993.

[DLGH97] S.K. Debray, P. López-García, and M. V. Hermenegildo.
Non-Failure Analysis for Logic Programs.
In *1997 International Conference on Logic Programming*, pages 48--62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[DLGHL97]
S. K. Debray, P. López-García, M. V. Hermenegildo, and N.-W. Lin.
Lower Bound Cost Estimation for Logic Programs.
In *1997 International Logic Programming Symposium*, pages 291--305. MIT Press, Cambridge, MA, October 1997.

[DS99] Bart Demoen and K. Sagonas.
CHAT is $).
In D. Mc. Allester H. Ganzinger and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning*, volume 1705 of *Lectures Notes in Computer Science*, pages 337--357. Springer, September 1999.

[GdW94] J.P. Gallagher and D.A. de Waal.
Fast and Precise Regular Approximations of Logic Programs.
In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming (ICLP'94)*, pages 599--613. MIT Press, 1994.

[HBC96] M. V. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla.
The CIAO Multi-Dialect Compiler and System: A Demo and Status Report.
In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation Technology*. Computer Science Department, Technical University of Madrid, September 1996.
Available from
http://www.cliplab.org/Projects/COMPULOG/meeting96/papers/PS/clip.ps.gz
(http://www.cliplab.org/Projects/COMPULOG/meeting96/papers/PS/clip.ps.gz).

[HBC12] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla.
An Overview of Ciao and its Design Philosophy.
Theory and Practice of Logic Programming, 12(1--2):219--252, January 2012.
http://arxiv.org/abs/1102.5497.

[HBdlBP95]
M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla.
The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems.
In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995.
Available from http://www.cliplab.org/ (http://www.cliplab.org/).

[HBPLG99]
M. V. Hermenegildo, F. Bueno, G. Puebla, and P. López-García.
Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor.
In *1999 Int'l. Conference on Logic Programming*, pages 52--66, Cambridge, MA, November 1999. MIT Press.

[HC93] M. Hermenegildo and The CLIP Group.
Towards CIAO-Prolog -- A Parallel Concurrent Constraint System.
In *Proc. of the Compulog Net Area Workshop on Parallelism and Implementation Technologies*. FIM/UPM, Madrid, Spain, June 1993.

[HC94] M. Hermenegildo and The CLIP Group.
Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System.
In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123--133. Springer-Verlag, May 1994.

[HC97] M. V. Hermenegildo and The CLIP Group.
An Automatic Documentation Generator for (C)LP -- Reference Manual.
The Ciao System Documentation Series--TR CLIP5/97.3, Facultad de Informática, UPM, August 1997.
Online at http://ciao-lang.org.

[HCC95] M. Hermenegildo, D. Cabeza, and M. Carro.
Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems.

In *Proc. of the Twelfth International Conference on Logic Programming*, pages 631--645. MIT Press, June 1995.

[Her86] M. Hermenegildo.
An Abstract Machine for Restricted AND-parallel Execution of Logic Programs.
In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25--40. Imperial College, Springer-Verlag, July 1986.

[Her96] M. Hermenegildo.
Writing ‘‘Shell Scripts’’ in SICStus Prolog, April 1996.
Posting in comp.lang.prolog. Available from http://www.cliplab.org/ (http://www.cliplab.org/).

[Her99] M. V. Hermenegildo.
A Documentation Generator for Logic Programming Systems.
Technical Report CLIP10/99.0, Facultad de Informática, UPM, September 1999.

[Her00] M. V. Hermenegildo.
A Documentation Generator for (C)LP Systems.
In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345--1361. Springer-Verlag, July 2000.

[HG91] M. Hermenegildo and K. Greene.
The &-Prolog System: Exploiting Independent And-Parallelism.
New Generation Computing, 9(3,4):233--257, 1991.

[Hog84] C. J. Hogger.
Introduction to Logic Programming.
Academic Press, London, 1984.

[Hol90] C. Holzbaur.
Specification of Constraint Based Inference Mechanisms through Extended Unification.
PhD thesis, University of Vienna, 1990.

[Hol92] C. Holzbaur.
Metastructures vs. Attributed Variables in the Context of Extensible Unification.
In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260--268. LNCS 631, Springer Verlag, August 1992.

[Hol94] C. Holzbaur.
SICStus 2.1/DMCAI Clp 2.1.1 User's Manual.
University of Vienna, 1994.

[HPB99] M. V. Hermenegildo, G. Puebla, and F. Bueno.
Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging.
In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25--Year Perspective*, pages 161--192. Springer-Verlag, July 1999.

[HPBLG05]
M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia.
Integrated Program Debugging, Verification, and Optimization Using

Abstract Interpretation (and The Ciao System Preprocessor).
Science of Computer Programming, 58(1--2):115--140, October 2005.

[JL87] Joxan Jaffar and Jean-Louis Lassez.
Constraint LP.
In *POPL*, pages 111--119. ACM, 1987.

[JL88] D. Jacobs and A. Langen.
Compilation of Logic Programs for Restricted And-Parallelism.
In *European Symposium on Programming*, pages 284--297, 1988.

[Knu84] D. Knuth.
Literate programming.
Computer Journal, 27:97--111, 1984.

[Kor85] R. Korf.
Depth-first iterative deepening: an optimal admissible tree search.
Artificial Intelligence, 27:97--109, 1985.

[LGBH05] P. López-García, F. Bueno, and M. V. Hermenegildo.
Determinacy Analysis for Logic Programs Using Mode and Type Information.
In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19--35. Springer-Verlag, August 2005.

[LGBH10] P. López-García, F. Bueno, and M. V. Hermenegildo.
Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.
New Generation Computing, 28(2):117--206, 2010.

[LGHD96] P. López-García, M. V. Hermenegildo, and S. K. Debray.
A Methodology for Granularity Based Control of Parallelism in Logic Programs.
Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation, 21(4--6):715--734, 1996.

[MH89] K. Muthukumar and M. Hermenegildo.
Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation.
In *1989 North American Conference on Logic Programming*, pages 166--189. MIT Press, October 1989.

[Mye86] Eugene W Myers.
An O(ND) difference algorithm and its variations.
Algorithmica, 1(1-4):251--266, 1986.

[Nai85] L. Naish.
The MU-Prolog 3.2 Reference Manual.
TR 85/11, Dept. of Computer Science, U. of Melbourne, October 1985.

[Nai91] Lee Naish.
Adding equations to NU-Prolog.
In *Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming (PLILP'91)*, number 528 in Lecture Notes in Computer Science, pages 15--26, Passau, Germany, August 1991. Springer-Verlag.

[Par97] The RADIOWEB Project Partners.
RADIOWEB EP25562: Automatic Generation of Web Sites for the Radio

Brodcasting Industry -- Project Description / Technical Annex. Technical Report, RADIOWEB Project, July 1997.

[PBH97] G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997. Available from ftp://cliplab.org/pub/papers/assert_lang_tr_discipldeliv.ps.gz (ftp://cliplab.org/pub/papers/assert_lang_tr_discipldeliv.ps.gz) as technical report CLIP2/97.1.

[PBH00] G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23--61. Springer-Verlag, September 2000.

[PH99] G. Puebla and M. V. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *ICLP'99 Workshop on Optimization and Implementation of Declarative Languages*, pages 45--61. U. of Southampton, U.K, November 1999.

[PW80] F.C.N. Pereira and D.H.D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231--278, 1980.

[RRR96] Prasad Rao, C. R. Ramakrishnan, and I.V. Ramakrishnan. A Thread in Time Saves Tabling Time. In *JICSLP*. MIT Press, 1996.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[Swe95] Swedish Institute of Computer Science, P.O. Box 1263, S-16313 Spanga, Sweden. *Sicstus Prolog V3.0 User's Manual*, 1995.

[TS86] H. Tamaki and M. Sato. OLD Resol. with Tabulation. In *ICLP'86*, pages 84--98. LNCS, 1986.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.

[War88] D.H.D. Warren. The Andorra Model. Presented at Gigalips Project workshop. U. of Manchester, March 1988.

[War92] D. S. Warren. Memoing for Logic Programs. *CACM*, 35(3):93--111, 1992.

Library/Module Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Predicate Index

A

B

C

D

E

F

I

J

K

L

O

P

Q

R

S

Property Index

Regular Type Index

A

B

C

D

E

F

R

S

T

U

V

W

Declaration Index

T

U

Concept Index

&

.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Author Index

T

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, authors, etc., are referred to in the text of the document.

.

/

:

;

=

?

@

[

|

~

+

>

^

\

<

A

B

C

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z