The lpdoc Documentation Generator

An Automatic Documentation Generator for (C)LP Systems
REFERENCE MANUAL
The Ciao Documentation Series
`https://ciao-lang.org/`
Generated/Printed on: 22 July 2023
Technical Report CLIP 5/97.1-3.5
Version 3.5 (2022/3/2, 20:34:30 CET)

Edited by:
Manuel Hermenegildo
Jose F. Morales

Table of Contents

1 Summary

`lpdoc` is an *automatic program documentation generator* for (C)LP systems.

`lpdoc` generates a reference manual automatically from one or more source files for a logic program (including ISO-Prolog, Ciao, and other CLP systems). It is particularly useful for documenting library modules, for which it automatically generates a description of the module interface. However, `lpdoc` can also be used quite successfully to document full applications and to generate nicely formatted plain ascii "readme" files. A fundamental advantage of using `lpdoc` to document programs is that it is much easier to maintain a true correspondence between the program and its documentation, and to identify precisely to what version of the program a given printed manual corresponds.

The quality of the documentation generated can be greatly enhanced by including within the program text:

- *assertions* (types, modes, etc. ...) for the predicates in the program, and
- *machine-readable comments* (in the "literate programming" style).

The assertions and comments included in the source file need to be written using the Ciao system *assertion language*. A simple compatibility library is available to make traditional (constraint) logic programming systems ignore these assertions and comments allowing normal treatment of programs documented in this way.

The documentation is currently generated directly in `HTML` or `texinfo` format. From the `texinfo` output, `lpdoc` also generates printed and on-line manuals in several other formats automatically (`pdf`, `info`, `ascii`, etc.), by calling publicly available tools. `lpdoc` can also generate 'man' pages (Unix man page format) as well as brief descriptions in html or emacs info formats suitable for inclusion in an on-line index of applications. In particular, `lpdoc` can create and maintain fully automatically WWW and info sites containing on-line versions of the documents it produces.

The `lpdoc` manual (and the Ciao system manuals) are generated by `lpdoc`.

`lpdoc` is distributed under the GNU general public license.

This documentation corresponds to version 3.5 (2022/3/2, 20:34:30 CET).

2 Introduction

`lpdoc` is an *automatic program documentation generator* for (C)LP systems.

`lpdoc` generates a reference manual automatically from one or more source files for a logic program (including ISO- `Prolog` [DEDC96], `Ciao` [Bue95], and other `CLP` [JM94] systems). It is particularly useful for documenting library modules, for which it automatically generates a description of the module interface. However, `lpdoc` can also be used quite successfully to document full applications and to generate nicely formatted plain ASCII "readme" files. A fundamental advantage of using `lpdoc` to document programs is that it is much easier to maintain a true correspondence between the program and its documentation, and to identify precisely to what version of the program a given printed manual corresponds.

2.1 Overview of this document

This first part of the document provides usage information on how to generate and access manuals (Chapter 3 [Generating and accessing manuals], page 13), as well as detailed instructions required to write proper documentation manuals (Chapter 4 [Writing documentation], page 17). Examples are given using the files in the `examples` directory provided with the `lpdoc` distribution.

Other parts of this document provide:

- Documentation on the syntax and meaning of the *assertions* that `lpdoc` uses (those defined in the `Ciao assertions` library [PBH97, PBH98, Bue98]). These include *comment* assertions (containing basically documentation text), formal assertions (containing properties), and combined assertions.
- Documentation on a basic set of properties, types, etc. which are predefined in the `Ciao basic_props`, `regtypes`, `native_props`, and `meta_props` libraries. These properties, and any others defined by the user or in other `Ciao` libraries, can be used in program assertions.
- Documentation on the formatting commands that can be embedded in *comments*.

A separate internals manual provides information on how the different internal parts of `lpdoc` are connected, which can be useful if new capabilities need to be added to the system or its libraries are used for other purposes.

All of the above have been generated automatically from the assertions in the corresponding sources and can also be seen as examples of the use of `lpdoc`.

Some additional information on `lpdoc` can be found in [Her00].

2.2 Version/Change Log

Version 3.5 (2022/3/2, 20:34:30 CET)
Highlights of this release:

- ADDED: Dark color-scheme CSS.
- IMPROVED: Selected test cases can be documented as examples (`example/1` prop).
- IMPROVED: UTF-8 as default encoding in texinfo documentation.
- IMPROVED: Added support for more accented characters in HTML backend.
- IMPROVED: Simpler CSS (vars).

Version 3.4 (2021/3/18, 19:33:30 CET)
Highlights of this release:

- FIXED: Missing command declarations for markdown syntax.

- NEW: `-op` for alternative code source suffix (useful for flycheck).
- NEW: A `nil` backend for lpdoc syntax checking only (no actual documentation is generated).
- FIXED: Avoid redefining `SETTINGS` module errors.
- FIXED: Improve support for `bibtex` accents.

Version 3.3 (2020/3/20, 14:48:45 CET)

Highlights of this release:

- Improved tolerance to missing dependencies, reporting disabled backends and causing dependency. Markdown translation turned on by default.

Detailed list of new features and changes:

- ENHANCED: `is_operational` checks for backends (disable them dynamically if some 3rd-party dependencies, e.g., tex, makeinfo, etc. are missing). Informative messages about what is missing.
- ENHANCED: Improved support for website generation
- CHANGED: `Up` navigation arrow links to `/` URL (i.e., website or doc index). This is handy when using `M-x ciao-serve` or navigating the ciao-lang website.
- CHANGED: Do not abort if `bibtex` is missing/failing (just emit error messages).
- FIXED: Turn on markdown translation by default in default `SETTINGS.pl` (used, e.g., by `C-c D B` in the emacs mode).
- FIXED: Check that arg 1 of fact pointed by `@`includefact is a string.
- FIXED: Dependency to `texindex` for dvi,ps,pdf formats.
- FIXED: Do not abort doc generation on `convert` errors.

Version 3.2 (2018/12/6, 11:25:8 CEST)

- Major improvements in HTML backend (which now is also mobile-friendly).
- Built-in search in HTML manuals.
- Do not generate `.ps` or `.dvi` by default.
- (experimental) markdown-style documentation and documentation comments `"%! "`.

Version 3.1 (2016/12/31, 11:36:37 CEST)

- Simplified manual setting files.
- LPdoc can be used as a library (simplified interface `lpdoc(docmaker)`, etc.).
- Adding built-in toplevel (in addition to the command-line interface).
- The markup commands `@uref{URL}` and `@uref{Text}{URL}` have been deprecated. Use `@href{URL}` and `@href{URL}{Text}` instead.

Version 3.0 (2011/7/7, 16:33:15 CEST)

- Major redesign of the documentation generator:
  - LPdoc redesigned to work internally with a 'doctree' representation (a-la Pillow). (Jose Morales)
  - A native HTML backend (not generated from texi). (Jose Morales)
  - Allow custom website generation from LPdoc documents. (Jose Morales)
  - Two passes for document generation, allowing resolution of bibliographical references in all backends (including HTML). (Jose Morales)
  - `doc_structure/1` in `SETTINGS` allows structure in LPdoc documents (sections can really be nested inside parts). (Jose Morales)

  - `:- doc(_,_)` is the recommended syntax for documentation comments now.
  - Replacing `:- comment` by `:- doc` in LPdoc code, updated documentation. (Jose Morales)
- General improvements and bug fixes:
  - Designed a logo for LPdoc. (Jose Morales)
  - LPdoc comments can now be written using `%!` style comment syntax. (Manuel Hermenegildo)
  - Now commas etc. are allowed in section names (so that they can be used in other formats). They are eliminated automatically in texi and info. This avoids wrong section names –and thus dangling pointers– in generated texinfo files. (Manuel Hermenegildo)
  - Eliminated superfluous copy of summary in info mode. (Manuel Hermenegildo)
  - Eliminated unsupported chars that broke texi manual cross-referencing. (Manuel Hermenegildo)
  - Improved treatment of accents (dotless i and dotless j, o, etc.). (Manuel Hermenegildo)
  - Initial size passed to `xdvi` more appropriate for current `xdvis`. (Manuel Hermenegildo)
  - Accents in bibliography fixed. (Manuel Hermenegildo)
  - Now repeated sections are disambiguated. (Manuel Hermenegildo)
  - Eliminated unnecessary escaping (especially for &). (Manuel Hermenegildo)
  - Better detection of when version is not available. (Manuel Hermenegildo)
  - Added new `doc(address, _)` comment, which is the right place to put address/contact information in manuals (Jose Morales)
  - Added new `@version{}` command (expands to the version of the software to be documented). (Jose Morales)
  - Shorter `SETTINGS.pl` files (with some rudimentary, assertion-based checking of options) (Jose Morales)
  - Bug fix: '`@@ include`' and '`@@ includeverbatim`' are no longer a problem (space can be omitted) (Jose Morales)
  - Added and documented a new `documentation` filetype (for some parts of the manual that contains only documentation). That avoids the old trick of declaring a fake `main/0` predicate. (Jose Morales)
  - Style for subtitle added automatically (in texinfo, it is *emph*; in HTML it is normal text with smaller font). The entries in `subtitle_extra` are free-form. (Jose Morales)
  - Bugs and changelog appear now in the global links in the HTML backend. (Jose Morales)
  - Merged code that documented `.pl` and `.lpdoc` files. (Jose Morales)
  - No copyright section if no copyright comment. (Jose Morales)
  - Auxiliary documentation files ending in '`_doc`' displayed incorrect names for the module (ending in '`_doc`'). E.g., `use_package(foo_doc)` was displayed instead of `use_package(foo_doc)`. Fixed. (Jose Morales)
  - In `verbatim` enviroments, new-line characters are removed from the beginning. (Jose Morales)

- Fix wrong use of `erase/1` for clauses (which resulted in segmentation fault when documentation generation failed) (Jose Morales)
- Fixed image generation (now uses `.png` files for HTML) (Jose Morales)
- New code for text escape fixed some problems, like '`@/1`' operator not being displayed correctly in Info. (Jose Morales)
- Colors for Prolog variables (in HTML). (Jose Morales)
- Added `@begin{alert}` environment for alert messages (like cartouche, but in red). (Jose Morales)
- Supporting '`@"`' command for umlaut, in addition to '`@..`' (Jose Morales)
- Double quotes correctly translated to HTML (Jose Morales)
- `@author` command to reference authors (changed command referring to people by `@author`, in all the documentation) (Jose Morales)
- Simplification of documentation setting files (see the documentation for further details) (Jose Morales)
- Using `open` for `lpdoc htmlview` command in MacOS X (Jose Morales)
- Adding `html` and `pdf` formats as options for emacs customization of LPdoc (`html` is the default one now) (Jose Morales)
- Improved detection of external tools for image conversion. (Manuel Hermenegildo)
- Added section name syntax auto-correction. This avoids wrong section names –and thus dangling pointers– in generated texinfo files. (Manuel Hermenegildo)
- Document size more appropriate for current xdvi versions. (Manuel Hermenegildo)
- Lpdoc no longer adds .info filename suffix to .infoindex entries since it breaks Debian's install-info –remove and goes against standard practice anyway. (Jose Luis Gonzalez)
- Added option –cv, –comment-version, that tells lpdoc if the file has version comment. Formatting of lpdoc version comments completed. (Edison Mera)
- Improved handling of option values. Added -d option to lpdoc, that allows defining additional values in the argument. Added options -l and -m that are similar to the corresponding lpmake options. (Edison Mera)

- Support for in-code sections (experimental):
  - Latex-like font-lock highlight of sectioning documentation comments (`:- doc(C, "...")`, with `C` one of `title`, `section`, and `subsection`).

    Currently the `section` and `subsection` comments are still ignored by LPdoc. (Jose Morales)
- Support for mathematical notation (experimental):
  - new `@math{...}` and `@begin{displaymath}...@end{displaymath}` enviroments are supported (see the documentation for more details) (Jose Morales)
  - In documentation strings, single `\` must be escaped (e.g. '`@math{\\lambda}`') (Jose Morales)
  - Supported in both the texinfo and HTML (using MathJax) backends. (Jose Morales)

- Added `@defmathcmd{Cmd}{N}{Def}` and `@defmathcmd{Cmd}{Def}`, both for texinfo and HTML backends. Those LPdoc commands define new mathematical environments (equivalent to `\newcommand`). (Jose Morales)

Version 2.1 (2004/10/28, 16:38:17 CEST)
Last version before moving to subversion. 1.9 and 2.0 were merged. 1.9 (based on makefiles) is deprecated.

- New functionality:
  - Use of `:- doc` declarations (as a shorthand for `comment`) now allowed. (Manuel Hermenegildo)
  - Made xdvi viewer, ps viewer, and xdvi zoom size be parameters (the latter since new versions of xdvi display sizes differently than old ones). (Manuel Hermenegildo)
  - Processing options can now be set for each file independently. (Manuel Hermenegildo)
  - Proper pdf generation now achieved in most cases, thanks to newer versions of `dvips`. (Manuel Hermenegildo)
  - Added option -c Target in lpdoc, that treats Target as a separate component. (Edison Mera)
  - Added option -f ConfigFile in lpdoc, that uses the file ConfigFile instead the default LPSETTINGS.pl. (Edison Mera)
  - Added option ascii that generates documentation in ascii plain format. (Edison Mera)
  - Added –help option. Is equal to -h. (Edison Mera)
  - Added option testsettings to check that the settings file is correctly specified. (Edison Mera)
  - Changed `generate_html_pointer/5` by `generate_html_pointer/6` to let it work with any given directory, and not only the working directory. (Edison Mera)

Version 2.0 (1999/8/17, 17:28:52 CEST)
Major change to eliminate need for Makefiles: lpdoc is now a standalone command (Manuel Hermenegildo). Proceeds in parallel with further development of 1.9. Merge pending. Previous changes incorporated since 1.8:

- New functionality:
  - A new parameter `PAPERTYPE` can be set in the `SETTINGS` file which controls the format of printed output. (Manuel Hermenegildo)
  - Default pdf viewer is now `ghostview`, sicne recent versions handle `pdf` well. (Manuel Hermenegildo)
  - Changed default style sheet in order to show `<PRE>` lines with a monospaced font. (Daniel Cabeza Gras)
  - Mode definitions now documented in a separate section. The way they are documented has been improved. (Manuel Hermenegildo)
  - References in files now updated only if `.refs` file is not empty. (Manuel Hermenegildo)
  - A *copy* of the html style sheet is now included in *distributions*. Also *Copies* of the html and info index head and tail files. (Manuel Hermenegildo)
  - Made pointers relative in library html templates. (Manuel Hermenegildo)
- Bug fixes and other minor improvements:

- Declarations now documented properly even if they have the same name and arity as a predicate. (Manuel Hermenegildo)
- Accented i's now translate correctly in html. (Manuel Hermenegildo)
- Fixed a funny installation quirk: while we want to install LPdoc in the Ciao group, the manuals produced by LPdoc should be installed in the LPdoc group. (Manuel Hermenegildo)
- Now using `lpdoclib` path alias. (Manuel Hermenegildo)
- Fixed bug in ordering of html indices in recent Linux versions, related to varying file listing order depending on locale. (Manuel Hermenegildo)

Version 1.9 (1999/7/8, 18:19:43 MEST)

In this release the name of the application has changed to `lpdoc`.

- New commands:
  - `@`begin`{`cartouche`}` and `@`end`{`cartouche`}` commands now supported.
  - `@`foonote command now supported.
  - New `gmake htmlview` command (makes a running `netscape` visit the generated html manual). Suggested by Per Cederberg.
  - New `gmake distclean` command, intended for software distributions. Leaves the generated documents and eliminates *all* intermediate files (including `.texic`/`.texi` files).
  - Adobe `pdf` format now supported as a valid target. Unfortunately, embedded `.eps` figures are not supported at this time in pdf output.
  - The second argument of `:- comment(hide,...).` and `:- comment(doinclude,...).` declarations can now be a list of predicate names.
  - A `-u` *File* option is now supported so that a file including, e.g., path alias definitions can be included (this has the same functionality as the `-u` option in `ciaoc`).
  - Now typing just `gmake` does nothing. In order to do something at least one target should be specified. This was necessary so that recursive invocations with empty arguments did nothing.
  - Added a new filetype: `part`. This allows splitting large documents into parts, each of which groups a series of chapters.
- Other new functionality:
  - A style sheet can now be specified which allows modifying many characteristics of the html output (fonts, colors, background, ...) (thanks to Per Cederberg).
  - Added limited support for changing page numbering (in `SETTINGS` file).
  - The concept indexing commands (`@`index, `@`cindex, and `@`concept) now work somewhat differently, to make them consistent with other indexing commands.
  - The old *usage* index is now called, more appropriately, *global* index. Correspondingly, changed things so that now every definition goes to the global index in addition to its definitional index.
  - Imported files from module `user` are now documented separately.
  - Now a warning is issued if characters unsupported by info are used in section names.
  - Navigation in html docs was improved.

- The table of contents in printed manuals now contains entries for the individual descriptions of predicates, props, regtypes, declarations, etc. This can be shut off with the `-shorttoc` option.
- Made more silent in normal conditions: file inclusion is muted now unless `-v` option is selected.
- A single `.texi` file is now constructed (by grouping the `.texic` files generated for all components) in which the references and menus are resolved. This has the advantage that the process of resolving references and menus has now been sped up very significantly. Also, `texi` is now a valid target (perhaps useful for distributions). The generated files now have `texic` (*texinfo component*).
- Now, declarations are always documented as long as there is a `decl` assertion. Also, they are now documented in a separate section.

- Bug fixes and other minor improvements:
  - The directory containing html manual is now called *BASENAME*`_html` instead of just *BASENAME*, which was confusing.
  - Now requesting building a .ps only does not leave a .dvi behind (useful for distributions).
  - File names can now include the symbol `_` even if they contain figures.
  - `TeX`-related intermediate files are now cleaned up after each run in order to avoid clutter.
  - Fixed `-modes`, which was broken since going to the new normalizer (was normalizer problem). Fixed problem with no documentation when only modes given.
  - Fixed duplication of documentation for internal predicates when also exported.
  - Minor formatting problem when no documentation nor definition found for a regtype fixed.
  - Determining exports, imports, etc. now done solely by calls to `c_itf` library (and, thus, synchronized with `ciaoc` compiler).

(Manuel Hermenegildo)

Version 1.8 (1999/3/24, 21:15:33 MET)

This version completes the port to using the ciao 0.8 modular assertion processing library. In addition, it includes the following improvements:

- Now, if the name of a file being documented ends in `_doc`, the `_doc` part is left out when referring to the file in the documentation (useful if one would like to place the documentation declarations in different file).
- It is now possible to declare (via a `comment/2` declaration) the intended use of a file which is not a module (i.e. a package, user, or include file), which results in correct documentation of operator definitions, new declarations, etc. The declaration is only needed for 'user' files (i.e., files to be loaded with `ensure_loaded/1`).
- Separated generation of the manuals from their installation. I.e., `gmake install` now does not force a `gmake all`, which has to be done by hand. This was necessary to ensure correct installation of distributed manuals, even if modification dates are changed during installation. Previously, in some cases generation was triggered unnecessarily.
- New `-v` option allows using quieter by default operation when not debugging.

- New option `-propmods` makes the name of the module in which a property is defined appear in front of the property in the places where the property is used.
- New option `-noisoline` makes the textual explanation of the `iso/1` property not appear in the description of the usage (but the ⟨• ISO •⟩symbol does appear)
- Two new options, `-nosysmods` and `-noengmods`, selectively avoid listing the system or engine libraries used.
- If there is no declaration for a predicate, now a line is output with the name and arity and a simple comment saying that there is no further documentation available (this has the great advantage that then it goes in the index, and, for example in ciao, they get added to completion commands!).
- Now, if a property or regtype declaration has no textual comment, the actual definition is given (first level only) in the place where it is documented, and a simple generic message where it is used.
- Added `@`noindent and `@`iso commands.
- Nicer spacing now when printing predicate names which are operators, as well as modes, etc.
- Reporting of versions in libraries has been improved: now both the global version and the last version in which the library itself was changed are reported.
- Exported new declarations also documented now for include-type files.
- A module is now documented even if exports nothing at all.
- Engine modules used now documented even if no other modules used (was a reported bug).
- Fixed indexing of names containing `@` etc. for newer versions of texinfo.
- Tabs in verbatim modes now converted to a number of spaces (8). Not perfect, but produces better output than leaving the tabs in.
- Tex is now run in 'nonstopmode' which means it will typically not stop if there are minor errors (but some errors may go unnoticed...).
- The full path of the version maintenance directory is now computed (correctly) using the directory of the `.pl` file being documented as base.
- Notices for missing subtitle, copyright, and summary now only given from main file and not for components.
- Added special handling of regtype and generalized it to handle some props specially if there is a certain comp property present.

(Manuel Hermenegildo)

Version 1.7 (1998/12/2, 17:43:50 MET)
Major port to use the ciao 0.8 modular assertion processing library. (Manuel Hermenegildo)

Version 1.6 (1998/9/8, 12:49:26 MEST)
Added support for inserting images (.eps files) in text via `@`image command, email addresses via `@`email command, and url references via `@`uref command.

Unix 'man' output much improved. Also, it now includes a usage section. The correspoding text must be given in a string contained in the first argument of a fact of the `usage_message/1` predicate which appears in the program. Also, formatting of 'man' pages has been greatly improved.

A new 'ascii' format is now supported: a simple minded ascii manual (basically, an info file without pointers).

(Manuel Hermenegildo)

Version 1.5 (1998/8/23, 20:30:32 EST)
Now supporting a `@`cite command (YES!). It automatically accesses the bib entries in `.bib` files (using `bibtex`) and produces a 'References' appendix. `@`cite can be used in the text strings exactly as \cite in LaTeX. The set of bib files to be used is given in the `SETTINGS` file.

Defining the type of version maintenance that should be performed by the `emacs` ciao.el mode (i.e., whether version numbers are in a given directory or in the file itself) is controlled now via a standard `commment/2` declaration. You should now write a declaration such as:

```
:- comment(version_maintenance,dir('../version')).
```

to state that control info is kept in directory `../version`. This has the advantage that it is shorter than the previous solution and that lpdoc can read this info easily. Using this guarantees that the version numbers of the manuals always concide with those of the software.

Generation of indices of manuals (.htmlbullet files): if several manuals are installed in the same directory, an index to them is now generated at the beginning of the html cover page describing the directory.

(Manuel Hermenegildo)

Version 1.4 (1998/8/4, 19:10:35 MET DST)
The set of paths defined in `SETTINGS` for finding the source files are now also used to find 'included' files. As a result, full path is not needed any more in, e.g, `@`include command.

New `@`ref command which can be used to refer to chapeter, sections, subsections, etc..

Support for recent minor changes in assertion format, including '#' as comment separator.

Used modules are now separated in documentation (in the interface description) by type (user, system, engine...).

Supports new 'hide' option in comments, to prevent an exported predicate from being documented. This is useful for example for avoiding mentioning in the documentation multifile predicates which are not intended to be modified by the user.

(Manuel Hermenegildo)

Version 1.3 (1998/7/10, 16:35:2 MET DST)
Exports are now listed in the chapter header separated by kind (pred, types, properties, ...).

The list of other modules used by a module is now separated in the chapter header into User and System modules (controlled by two sets of paths in `SETTINGS`).

New *hide* option of comment/2 decl prevents an exported predicate from being included in the documentation: `:- comment(hide,p/3)`.

(Manuel Hermenegildo)

Version 1.2 (1998/6/4, 9:12:19 MET DST)
Major overall improvements... (Manuel Hermenegildo)

Version 1.1 (1998/3/31)
Incorporated autodoc and autodoformats library to source in order to make distribution standalone. Improvements to installation and documentation. `Makefile`s now also install documentation in public areas and produce global indices. Several documents can cohexist in the same installation directory. (Manuel Hermenegildo)

Version 1.0 (1998/2/24)
First Ciao-native distribution, with installation. (Manuel Hermenegildo)

Version 0.9 (1998/2/24)
Intermediate version, preparing for first major release. Modified `Makefile` and `SETTINGS` to handle installation of manuals. (Manuel Hermenegildo)

Version 0.6 (1998/2/10)
Added new indices and options, as well as more orthogonal handling of files. (Manuel Hermenegildo)

Version 0.4 (1998/2/24)
Added support for nroff -m formatting (e.g., for man pages). Added support for optional selection of indices to be generated. Added support for reexported predicates. Added (low level) ascii format. Added option handling (-nobugs -noauthors -noversion -nochangelog -nopatches -modes and -headprops ...). -literalprops. Fixed presentation when there are multiple kinds of assertions. Better error checking for includefact/includedef. (Manuel Hermenegildo)

Version 0.3 (1998/2/10)
Changed file reader to use Ciao native builtins. As a result, syntax files and full Ciao syntax now supported. Major reorganization of the code to make formatting more orthogonal. Now applications and libraries can be components or main files, standalone or with components interchangeably. `@`includefact, new predicate types, used libraries now precisely detected, `docinclude` option. (Manuel Hermenegildo)

Version 0.2 (1997/12/16)
Ported to native ciao. Version handling, selection of indices, `@`include. Added generation of an html brief description for a global index. Added unix manual page generation. Added support for specifying library paths. -l option for htmlindex and man. Installation improved: now all files for one application in the same directory. (Manuel Hermenegildo)

Version 0.1 (1997/7/30)
First official version (major rewrite from several previous prototypes, autodocumented!). (Manuel Hermenegildo)

Version 0.0 (1996/10/10)
First prototype.

3 Generating and accessing manuals

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

This section describes how to generate a manual (semi-)automatically from a set of source files using `lpdoc` and how to access it. See Chapter 4 [Writing documentation], page 17 for details about writing proper `lpdoc` documentation.

3.1 Basic usage

`lpdoc` can be used directly from the command line, the `Emacs` editor, or from bundle manifest files (see `bundles` in the Ciao reference manual).

The following provides the basic command-line usage and the main command line options available when invoking `lpdoc`. The basic usage is:

```
lpdoc [Options] Input
```

where `Input` is the input file (a module or a *documentation configuration* file), and (optional) options specifying the selected target format (`-t Format`, as described in `docformat/1`), generation options `--Name=Value` (see `doccfg` for a complete list), as well as options to view or clean the generated documentation files.

Use `lpdoc --help` to see a complete list of the available command line options.

3.1.1 Generating manuals

Manuals can be generated from single module or from a collection of modules specified in a *documentation configuration* file (as described in Chapter 4 [Writing documentation], page 17) which defines how the documentation is structured, as well as options for its generation). For example, executing from the command line:

```
lpdoc file.pl
```

generates the documentation for `file.pl` in the default format (`HTML` for single modules or the default formats specified in `docformat/1` for complex manuals), while the command:

```
lpdoc -t pdf file.pl
```

generates a PDF manual. The manuals generated will generally be written in the same directory as the input file, and they will have the same name but with the format as extension (i.e., in the example above it would be `file.pdf`). See `output_dir/1` and `output_name/1` options to change the location or name of the output).

To enable incremental documentation generation, `lpdoc` maintains intermediate files in a directory named `file.cachedoc/`. See ⟨undefined⟩ [Cleaning up], page ⟨undefined⟩ for cleanup commands to remove both the intermediate and target files.

If you use the `Emacs` editor (highly recommended in all circumstances), then the simplest way to quickly generate a manual is by doing it from the Ciao Emacs mode (this mode comes with the Ciao distribution and is automatically installed with Ciao). The Ciao Emacs mode provides menu- and keyboard-binding driven facilities for generating a standalone document with the documentation corresponding to the file in the buffer being visited by Emacs. This is specially useful while modifying the source of a file, in order to check the output that will be produced when incorporating this file into a larger document.

3.1.2 Cleaning up documentation

`lpdoc` can also take care of tidying up the output of the documentation generation and the intermediate documenation generation files, using the following options:

- `--clean`: deletes all intermediate files, but leaves the targets (i.e., the `.pdf`, `.ascii`, `.html`, etc. files), as well as all the generated `.texic` files.
- `--distclean`: deletes all intermediate files and the generated `.texic` files, leaving only the targets (i.e., the `.pdf`, `.ascii`, `.html`, etc. files). This is the option normally used when building software distributions in which the manuals come ready made in the distribution itself and will not need to be generated during installation.
- `--docsclean`: deletes all intermediate files and the generated targets, but leaves the `.texic` files. This option can be used in software distributions in which the manuals in the different formats will be generated during installation. This is generally more compact, but requires the presence of several tools, such as `tex`, `Emacs`, etc., in order to generate the manuals in the target formats during installation.
- `--realclean`: performs a complete cleanup, deleting also the .texic files, i.e., it typically leaves only the original source files. This is is the most compact, but requires the presence of the tools mentioned above, the source files from which the manuals are generated and `lpdoc` in order to regenerate the manuals in the target formats during installation.

3.2 Accessing manuals

Once generated, the documentation can be viewed by opening the target output with an appriopriate viewer (e.g., Web browser for `file.html/index.html`, PDF viewer for `file.pdf`, etc.). For convenience `lpdoc` provides a generic view command:

```
lpdoc -t Format --view file.pl
```

which will open a default viewer application for the specified format and file.

For HTML documentation (specially when it is part of bundles) we encourage the use of the `ciao-serve` command, which starts a local HTTP server and provides access to dynamic documentation parts (such as search, and advanced index options).

3.2.1 Accessing info manuals

Generated `.info` files are meant to be viewed by the `Emacs` editor or by the standalone `info` application, both publicly available from the GNU project sites. To view the a generated `info` file from `Emacs` manually (i.e., before it is installed in a common area), type `C-u M-x info`. This will prompt for an info file name. Input the name of the info file generated by `lpdoc` (`main.info`) and `Emacs` will open the manual in info mode.

Automatic, direct on-line access to the information contained in the info file (e.g., going automatically to predicate descriptions by clicking on predicate names in programs in an `Emacs` buffer) can be easily implemented via existing `.el` packages such as `info-look`, written by Ralph Schleicher (`rs@ralph-schleicher.de`). Support for this package can be found in `info-look-ciao.el`

3.2.2 Accessing man manuals

The `Unix man` format manuals generated by `lpdoc` can be viewed using the `Unix man` command. In order for `man` to be able to locate the manuals, they should be copied to one of the subdirectories (e.g., `/usr/local/man/manl`) of one of the main man directories (in the previous

case the main directory would be `/usr/local/man`). As usual, any directory can be used as as a man main directory, provided it is included in the environment variable `MANPATH`.

3.2.3 Accessing Active Logic Documents

The `.html` pages generated by `lpdoc` can be conveniently accessed by users in their web browsers without requiring any installation. This facilitates easy sharing of `ALD`s.

4 Writing documentation

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

This section includes some details for writing proper documentation in `lpdoc`, improving the layout of manuals, usage tips, and troubleshooting advice.

4.1 Documenting source files

While `lpdoc` can produce useful documentation from the interface of the source file, the quality of the documentation generated can be greatly enhanced by including within the program text:

- *assertions*, and
- *machine-readable comments*.

Assertions are declarations which are included in the source program and provide the compiler with information regarding properties of the code. Typical assertions include type declarations, modes, computational properties (such as nonfailure or determinacy), many compiler directives (such as `dynamic/1`, `op/3`, `meta_predicate/1`...), etc. When documenting a module, `lpdoc` will use the assertions associated with the module interface to construct a textual description of this interface. In principle, only the exported predicates are documented, although any predicate can be included in the documentation by explicitly requesting it (see the documentation for the `doc/2` declaration). Judicious use of these assertions allows at the same time documenting the program code, documenting the external use of the module, and greatly improving program debugging and allowing program verification. The latter is possible because the assertions provide the compiler with information on the intended meaning or behaviour of the program (i.e., the specification) which can be checked at compile-time (by a suitable preprocessor/static analyzer) and/or at run-time (via checks inserted by a preprocessor). See Chapter 10 [The Ciao assertion language], page 53 for more details.

Machine-readable comments are also declarations included in the source program which contain additional information intended to be read by humans (i.e., this is an instantiation of the *literate programming* style of Knuth [Knu84]). Typical such comments include title, author(s), summary, bugs, changelog, etc. Judicious use of these comments allows enhancing at the same time the documentation of the program text and the manuals generated from it. See Chapter 7 [Documentation mark-up language and doc declarations], page 31 for more details. These declarations can also be writtten in wiki (*mark-down*) style – see `doccomments`.

`lpdoc` requires these **assertions** and **machine-readable** comments to be written using the `Ciao` *assertion language*. While `Ciao` has core support for this language, it is however quite straightforward in most Prolog and (C)LP systems to define a library with dummy declarations so that the assertions and comments meant for `lpdoc` are simply ignored by the compiler, making it possible to compile programs documented using assertions and comments in such systems and at the same time generate documentation using `lpdoc` (as well as making other uses of the assertions such as checking tehm statically and/or dynamically).

4.2 More complex manuals

Writing and generating more complex manuals involves writing a *documentation configuration* (`doccfg`) file (e.g, `SETTINGS.pl`) with this basic structure:

```
:- module(_, [], [doccfg]).

doc_structure :=
  '<MAIN MODULE>'-[
    '<COMP1>',
    ...
    '<COMPn>'
  ].
```

The first line indicates that this is a module implementing a `doccfg`. The second line defines through `doc_structure/1` the *document structure*, specifying in a tree the main file and (optional) component files. When documenting complex manuals with more than one source files, the main file is the one that will provide the general title, author, date, summary, and the introduction. The component files will appear as separate chapters.

Any option not directly specified will use the default values indicated in `doccfg`. You may

5 Documentation configuration options

These are the predicates that define the options that can be used in documentation configuration files. Their admissible values are given in their corresponding types in Chapter 6 [Admissible values for the documentation configuration options], page 25. See Chapter 4 [Writing documentation], page 17 for an introduction.

5.1 Usage and interface

- **Library usage:**
  Use with the `:- module(_, [], [doccfg])` directive in the documentation configuration file and provide the definition for the required entries.
- **Exports:**
  - *Predicates:*
    `doc_structure/1`, `filepath/1`, `output_name/1`, `output_dir/1`, `doc_mainopts/1`, `doc_compopts/1`, `docformat/1`, `index/1`, `bibfile/1`, `startpage/1`, `papertype/1`, `libtexinfo/1`, `comment_version/1`, `allow_markdown/1`, `allow_runnable/1`, `syntax_highlight/1`, `verbosity/1`, `warning_level/1`, `autogen_warning/1`, `html_layout/1`, `htmlurl/1`, `html_asset/1`, `tmplpath/1`, `load_doc_module/1`, `end_of_file/0`.
- **New operators defined:**
  `function/1` [1150,fx], `fun_eval/1` [1150,fx], `fun_return/1` [1150,fx], `:=/2` [1130,xfx], `~/1` [50,fx], `^^/1` [910,fx], `^/1` [25,fy], `|/2` [1105,xfy], `?/2` [1050,xfy].
- **Implicit imports:**
  - *Application modules:*
    `doccfg_props`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `stream_basic`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `fsyntax`, `assertions`, `assertions/assertions_basic`, `regtypes`.

5.2 Documentation on exports

doc_structure/1: PREDICATE

Usage: `doc_structure(Term)`

It defines the document structure as a tree.

The tree is defined as a root node with optional children. Nodes can be atoms or pairs (`N-Cs`), where `Cs` is a list of nodes. The root of the tree is the main file of the manual, i.e., the file which determines the manual's cover page, and first chapter. The children files are used as components, i.e., they will constitute the subsequent chapters of the manual.

If the main or any component file name ends with `_doc`, then it is treated as documenting the same file name without the `_doc` suffix. This is useful for separating the documentation from the source file (see ⟨undefined⟩ [Separating the documentation from the source file], page ⟨undefined⟩).

filepath/1: PREDICATE

Usage: `filepath(Path)`

It defines the directories where the `.pl` files to be documented can be found.

You also need to specify all the paths containing files which are used by the files being documented. For example, the paths to files included by an `@include` command or to figures.

– *The following properties should hold upon exit:*

`Path` is a full path to a directory. (`dirpath/1`)

output_name/1: PREDICATE

Usage: `output_name(Base)`

It determines the base file name of the main documents generated by `lpdoc`.

By default it is equal to the main file name (root) specified in `doc_structure/1`.

If the main file name ends with `_doc`, then it is equal to the name without the `_doc` suffix (see `doc_structure/1`).

If the `versioned_output` option is specified in `doc_mainopts/1`, the bundle version number is appended to the output name.

– *The following properties should hold upon exit:*

`Base` is a source name. (`sourcename/1`)

output_dir/1: PREDICATE

Usage: `output_dir(Base)`

Output directory. If undefined, uses directory of input

– *The following properties should hold upon exit:*

`Base` is a source name. (`sourcename/1`)

doc_mainopts/1: PREDICATE

Usage: `doc_mainopts(Option)`

`Option` is a processing option which should be activated when processing the main file.

It can be set to a series of options which allow more detailed control of what is included in the documentation for the main file and how (i.e., including bug information , versions and patches or only patches , authors , changelog , explanation of modes, *one-sided printing* (*two-sided* is the default), etc.).

See `option_comment/2` for a description of these options.

The default values are: `no_bugs`, `no_patches`.

– *Call and exit should be compatible with:*

Documentation generation options. (`supported_option/1`)

doc_compopts/1: PREDICATE

Usage: `doc_compopts(Option)`

`Option` is a processing option which should be activated when processing the secondary files (all except the main file).

Currently these options are common to all component files but they can be different from `doc_mainopts/1`.

See `option_comment/2` for a description of these options.

The default values are: `no_bugs`, `no_patches`.

– *Call and exit should be compatible with:*
Documentation generation options. (`supported_option/1`)

docformat/1: PREDICATE

Usage: `docformat(Format)`

Defines the documentation formats to be generated by default.

If the main file is an **application**, and the `manl` option is selected, then `lpdoc` looks for a `usage_message/1` fact, which should contain a string as argument, and will use that string to document the *usage of the application* (i.e., it will be used to fill in the *synopsis section of the man page*).

See `supported_format/1` for the available formats.

The default values are: `pdf`, `manl`, `info`, `html`.

– *The following properties should hold upon exit:*
Available formats (`supported_format/1`)

index/1: PREDICATE

Usage: `index(Idx)`

Defines the index sections to be generated by default.

Selecting `all` generates all the supported indices. However, note that this (as well as selecting many indices explicitely) exceeds the capacity of most texinfo installations.

See `index_comment/1` for a description of the indices.

The default values are: `concept`, `lib`, `pred`, `prop`, `regtype`, `decl`, `author`, `global`.

– *The following properties should hold upon exit:*
Supported indexes (`supported_index/1`)

bibfile/1: PREDICATE

Usage: `bibfile(F)`

It determines a list of *.bib files* (one file per path), i.e., files containing *bibliographic entries* in `bibtex` format.

This is only relevant if you are using citations in the text (using the `@`cite command). In that case those will be the files in which the citations will be searched for. All the references will appear together in a *References* appendix at the end of the manual.

If you are not using citations, then select the `no_biblio` option on the main file, which will prevent an empty 'References' appendix from appearing in the manual.

– *The following properties should hold upon exit:*
`F` is an atom describing the name of a file. (`filename/1`)

startpage/1: PREDICATE

Usage: `startpage(PageNumber)`

Page number of the first page of the manual.

Setting this to a different value allows changing the page number of the first page of the manual. This can be useful if the manual is to be included in a larger document or set of manuals. Typically, this should be an *odd* number.

The default value is 1.

– *The following properties should hold upon exit:*

`PageNumber` is an integer. (`int/1`)

papertype/1: PREDICATE

Usage: `papertype(PageNumber)`

Type of paper/format for printable outputs (e.g., `pdf`).

The currently supported outputs (most of them inherited from `texinfo`) are:

`afourpaper`
: The default, usable for printing on *A4 paper*. Rather busy, but saves trees.

`afourwide`
: This one crams even more stuff than `afourpaper` on an A4 page. Useful for generating manuals in the least amount of space. It saves more trees.

`afourlatex`
: This one is a little less compressed than `afourpaper`.

`smallbook`
: Small pages, like in a handbook.

`letterpaper`
: For printing on American *letter size paper*.

`afourthesis`
: A *thesis-like style* (i.e., double spaced, wide margins etc.). Useful – for inserting `lpdoc` output as appendices of a thesis or similar document. It does not save trees.

See also the `onesided` option for the main file.

The default value is: `afourpaper`.

– *The following properties should hold upon exit:*

Possible papertypes (`supported_papertype/1`)

libtexinfo/1: PREDICATE

Usage:

If set to yes the `texinfo.tex` file that comes with the lpdoc distribution will be used when generating manuals in formats such as `dvi` and `ps`. Otherwise, the texinfo file that comes with your `tex` installation will be used. It is recommended that you leave this set to `'yes'`.

– *The following properties should hold upon exit:*

Enumerated type for yes/no (`yesno/1`)

comment_version/1: PREDICATE

Usage:

The source files contain version information. If not specified lpdoc will assume the opposite

– *The following properties should hold upon exit:*

Enumerated type for yes/no (`yesno/1`)

allow_markdown/1: PREDICATE

Usage:

Allow LPdoc-flavored markdown in docstrings

– *The following properties should hold upon exit:*

Enumerated type for yes/no (`yesno/1`)

allow_runnable/1: PREDICATE

Usage:

Allow runnable code blocks (Ciao Playground)

– *The following properties should hold upon exit:*

Enumerated type for yes/no (`yesno/1`)

syntax_highlight/1: PREDICATE

Usage:

Syntax highlight code blocks (only for HTML backend)

– *The following properties should hold upon exit:*

Enumerated type for yes/no (`yesno/1`)

verbosity/1: PREDICATE

Usage:

Level of verbosity of (normally progress) messages. `quiet` means no messages printed, `normal` means standard messages, and `all` means more detailed messages. See `autodoc_message_t/1`.

– *The following properties should hold upon exit:*

`Arg1` is a verbosity level. (`verbosity_t/1`)

warning_level/1: PREDICATE

Usage:

Warning reporting level. `none` means only error messages are printed, `normal` means errors and warnings, `all` means also notes. See `autodoc_message_t/1`.

– *The following properties should hold upon exit:*

`Arg1` is a warning level. (`warning_level_t/1`)

autogen_warning/1: PREDICATE

Usage:

Include an *automatically generated* notice inside the output text.

– *The following properties should hold upon exit:*

Enumerated type for yes/no (`yesno/1`)

html_layout/1: PREDICATE

Usage:

Layout for html output.

– *The following properties should hold upon exit:*

`Arg1` is any term. (`term/1`)

htmlurl/1: PREDICATE

Usage:

Deploy URL for html output

– *The following properties should hold upon exit:*

`Arg1` is any term. (`term/1`)

html_asset/1: PREDICATE

Usage:

Directory with additional files for HTML backend

– *The following properties should hold upon exit:*

`Arg1` is a full path to a directory. (`dirpath/1`)

tmplpath/1: PREDICATE

Usage:

Directory for HTML templates

– *The following properties should hold upon exit:*

`Arg1` is a full path to a directory. (`dirpath/1`)

load_doc_module/1: PREDICATE

Usage: `load_doc_module(F)`

Documentation module (`doc_module`) for extensions (see `doc_module`).

– *The following properties should hold upon exit:*

`F` is any term. (`term/1`)

end_of_file/0: PREDICATE

No further documentation available for this predicate.

6 Admissible values for the documentation configuration options

This module defines the regular types that define the admissible values for the documentation configuration options.

6.1 Usage and interface

- **Library usage:**
  `:- use_module(library(doccfg/doccfg_props)).`
- **Exports:**
  - *Predicates:*
    `option_comment/2`, `index_comment/2`.
  - *Regular Types:*
    `dirpath/1`, `filename/1`, `supported_option/1`, `supported_format/1`, `supported_index/1`, `supported_papertype/1`, `yesno/1`, `verbosity_t/1`, `warning_level_t/1`.

6.2 Documentation on exports

dirpath/1: REGTYPE

An atom describing a path to a directory. Should be a full, explicit path (i.e., not containing environment variables).

Usage: `dirpath(P)`

`P` is a full path to a directory.

filename/1: REGTYPE

Usage: `filename(P)`

`P` is a full path to a file.

supported_option/1: REGTYPE

In general, selecting none of these options generates the most verbose manuals, i.e., each option generally supresses the production of a certain kind of output (on the other hand, `'verbose'` selects verbose output from `ciaoc` when processing the file).

Possible options:

```
supported_option(verbose).
supported_option(no_bugs).
supported_option(no_authors).
supported_option(no_stability).
supported_option(no_version).
supported_option(versioned_output).
supported_option(no_lpdocack).
supported_option(no_changelog).
supported_option(no_patches).
supported_option(autogen_warning).
```

```
supported_option(modes).
supported_option(head_props).
supported_option(literal_props).
supported_option(no_propnames).
supported_option(no_undefined).
supported_option(no_propsepln).
supported_option(no_biblio).
supported_option(no_sysmods).
supported_option(no_engmods).
supported_option(no_packages).
supported_option(no_isoline).
supported_option(propmods).
supported_option(no_propuses).
supported_option(shorttoc).
supported_option(regtype_props).
supported_option(onesided).
supported_option(no_math).
supported_option(tests).
supported_option(no_examples).
supported_option(status).
```

Usage:

Documentation generation options.

option_comment/2: PREDICATE

```
option_comment(Option,Text)
```

The currently supported options are:

```
option_comment(verbose,"Verbose output (good for debugging).").
option_comment(no_bugs,"Do not include information on bugs.").
option_comment(no_authors,"Do not include author names.").
option_comment(no_stability,"Do not include stability comment.").
option_comment(no_version,"Do not include version information.").
option_comment(versioned_output,"Include version in the output name.").
option_comment(no_lpdocack,"Do not include an ack of LPdoc in output.").
option_comment(no_changelog,"Do not include change log.").
option_comment(no_patches,"Do not include comments for patches.").
option_comment(autogen_warning,"Include autogenerated warning (only ascii backe
option_comment(modes,"Do not translate modes and their arguments (except for pr
option_comment(head_props,"Do not move head properties to body.").
option_comment(literal_props,"Do not use text to document properties.").
option_comment(no_propnames,"Do not include property names in prop text.").
option_comment(no_undefined,"Do not signal undefined properties in text.").
option_comment(no_propsepln,"Do not put each property in a separate line.").
option_comment(no_biblio,"Do not include a bibliographical 'References' appendi
option_comment(no_sysmods,"Do not include system modules in the import list.").
option_comment(no_engmods,"Do not include system engine modules in the import l
option_comment(no_packages,"Do not include packages in the import list.").
option_comment(no_isoline,"Do not include *textual* description that a given us
option_comment(propmods,"Include module name to which props belong.").
option_comment(no_propuses,"Do not include property uses (from assertions) in i
```

```
option_comment(shorttoc,"Produce shorter table of contents (no entries for indiv
option_comment(regtype_props,"Include in the doc for regtypes the global prop st
option_comment(onesided,"For printing on one side (default is two).").
option_comment(no_math,"Disable mathematical environments.").
option_comment(tests,"Include test assertions in doc for preds (default: only te
option_comment(no_examples,"Document no test assertions, i.e., also not those th
option_comment(status,"Document also the status of assertions (default is no sta
```

Usage: `option_comment(Option,Text)`

`Option` is a documentation option which is supported. These options can be applied to the main file or to the components. `Text` describes the effect of selecting that option.

– *The following properties should hold upon exit:*

Documentation generation options. (`supported_option/1`)

`Text` is a string (a list of character codes). (`string/1`)

supported_format/1: REGTYPE

The available formats are:

```
supported_format(texi).
supported_format(dvi).
supported_format(ps).
supported_format(pdf).
supported_format(ascii).
supported_format(manl).
supported_format(info).
supported_format(html).
```

Usage:

Available formats

supported_index/1: REGTYPE

Supported indexes:

```
supported_index(concept).
supported_index(lib).
supported_index(apl).
supported_index(pred).
supported_index(prop).
supported_index(regtype).
supported_index(decl).
supported_index(op).
supported_index(modedef).
supported_index(file).
supported_index(global).
supported_index(all).
```

Usage:

Supported indexes

index_comment/2: PREDICATE

`index_comment(Index,Text)`

The currently supported indices are:

```
index_comment(lib,"Library/module index").
index_comment(apl,"Application index").
index_comment(pred,"Predicate/method index").
index_comment(prop,"Property index").
index_comment(regtype,"Regular type index").
index_comment(decl,"Declaration index").
index_comment(op,"Operator index").
index_comment(modedef,"Mode index").
index_comment(file,"File/directory index").
index_comment(concept,"Concept index").
index_comment(author,"Author index").
index_comment(global,"Global index").
index_comment(all,"All supported indices").
```

Usage: `index_comment(Index,Text)`

`Type` is a type of index which is supported. `Text` describes the index contents.

– *The following properties should hold upon exit:*

`Index` is currently instantiated to an atom. (`atom/1`)

`Text` is a string (a list of character codes). (`string/1`)

supported_papertype/1: REGTYPE

Possible papertypes:

```
supported_papertype(letterpaper).
supported_papertype(smallbook).
supported_papertype(afourpaper).
supported_papertype(afourlatex).
supported_papertype(afourwide).
supported_papertype(afourthesis).
```

Usage:

Possible papertypes

yesno/1: REGTYPE

Enumerated type:

```
yesno(yes).
yesno(no).
```

Usage:

Enumerated type for yes/no

verbosity_t/1: REGTYPE

Defines the levels of verbosity for the messages produced by the message priting predicates.

```
verbosity_t(quiet).
verbosity_t(progress).
verbosity_t(full).
```

Usage: `verbosity_t(L)`

`L` is a verbosity level.

warning_level_t/1: REGTYPE

Defines which levels of warning will be printed by the message priting predicates.

```
warning_level_t(none).
warning_level_t(normal).
warning_level_t(all).
```

Usage: `warning_level_t(L)`

`L` is a warning level.

6.3 Documentation on imports

This module has the following direct dependencies:

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `regtypes`, `assertions`, `assertions/assertions_basic`, `fsyntax`.

7 Documentation mark-up language and doc declarations

Author(s): Manuel Hermenegildo.

Two documentation aids are defined herein:

- `doc/2` declarations and their many uses. These declarations are one of the two main means of enhancing the documentation of programs (the other being `assertions`). `doc/2` declarations provide *textual comments* which are *complementary* to the formal statements present in *assertions*. See the documentation for `doc/2` below, which also contains examples of use.
- The formatting commands of the *mark-up language* used in the text strings inside both `doc/2` declarations and in the comment field of assertions. See `docstring/1`, `stringcommand/1`, etc. below.

`doc/2` declarations can be used to document predicates and other parts of the program source in place of (or in combination with) the normal comments that are typically inserted in the code (and they are ignored by the compiler in the same way as classical comments). However, they are much preferred to classical comments because they are more structured and they are machine-readable, and can thus also be used to generate printed or on-line documentation automatically, using the `lpdoc` automatic documentation generator.

Note that in addition to the more structured `doc/2` declarations and mark-up language described here, these declarations can also be written in a *wiki* (*markdown*) style – see `doccomments` for details.

7.1 Usage and interface

- **Library usage:**
  It is not necessary to use this library in user programs. The recommended procedure in order to make use of the `doc/2` declarations that this library defines is to include instead the `assertions` package, which provides support for all assertion- and comment-related declarations, using one of the following declarations, as appropriate:

```
:- module(...,...,[assertions]).
:- use_package(assertions).
```

- **Exports:**
  - *Properties:*
    `docstring/1`, `stringcommand/1`.
  - *Regular Types:*
    `version_descriptor/1`, `filetype/1`, `stability_level/1`.

7.2 Documentation on exports

docstring/1: PROPERTY

Defines the format of the character strings which can be used in machine readable comments (`doc/2` declarations) and assertions. These character strings can include certain *formatting commands*.

- All printable characters are admissible in documentation strings except "`@`", "`\`", "`{`," and "`}`". To produce these characters the following *escape sequences* should be used, respectively: `@@`, `\\`, `@{`, and `@}`.

Note that in strings, each \ character must be escaped (with \, \\), so to get one backslash four backslashes (\\\\) have to be used, i.e., in a string \\\\ is used to represent \\, and that is the command that will output a \ character.

- In order to allow better formatting of on-line and printed manuals, in addition to normal text, certain formatting commands can be used within these strings. The syntax of all these commands is:

  `@`*command*

  (followed by either a space or `{}`), or

  `@`*command*`{`*body*`}`

  where *command* is the command name and *body* is the (possibly empty) command body. Alternatively, \ can be used instead of `{}` (escaped if appears in strings).

  The set of commands currently admitted can be found in the documentation for the predicate `stringcommand/1`.

Usage: `docstring(Text)`

`Text` is a *documentation string*.

stringcommand/1: PROPERTY

Defines the set of structures which can result from parsing a formatting command admissible in comment strings inside assertions.

In order to make it possible to produce documentation in a wide variety of formats, the command set is kept small. The names of the commands are intended to be reminiscent of the commands used in the LaTeX text formatting system. Both "`@`" and "\" are allowed in command names. Note that "\" needs to be escaped in strings, which makes the source less readable. In such case it is recommended using "`@`" (and, in any case, many ideas in LaTeX were taken from scribe, where the escape character was indeed `@`!).

The following are the currently admissible commands.

- **Formatting commands:**

  The following commands are used to format certain words or sentences in a special font, build itemized lists, introduce sections, include examples, etc.

  `@comment{`*text*`}`
  : *text* will be treated as a *comment* and will be ignored.

  `@begin{itemize}`
  : marks the beginning of an *itemized list*. Each item should be in a separate paragraph and preceded by an `@item` command.

  `@item`
  : marks the beginning of a new *item in an itemized list*.

  `@end{itemize}`
  : marks the end of an itemized list.

  `@begin{enumerate}`
  : marks the beginning of an *enumerated list*. Each item should be in a separate paragraph and preceded by an `@item` command.

  `@end{enumerate}`
  : marks the end of an enumerated list.

  `@begin{description}`
  : marks the beginning of a *description list*, i.e., a list of items and their description (this list describing the different allowable commads is in fact a description list). Each item should be in a separate paragraph and contained in an `@item{`*itemtext*`}` command.

`@item{`*itemtext*`}`
: marks the beginning of a *new item in description list*, and contains the header for the item.

`@end{description}`
: marks the end of a description list.

`@begin{verbatim}`
: marks the beginning of *fixed format text*, such as a program example. A fixed-width, typewriter-like font is used.

`@end{verbatim}`
: marks the end of formatted text.

`@begin{cartouche}`
: marks the beginning of a section of text in a *framed box*, with round corners.

`@end{cartouche}`
: marks the end of a section of text in a framed box.

`@begin{note}`
: marks the beginning of a section of text in a *framed box*, for note messages.

`@end{note}`
: marks the end of the note message.

`@begin{alert}`
: marks the beginning of a section of text in a *framed box*, for alert messages.

`@end{alert}`
: marks the end of the alert message.

`@section{`*text*`}`
: starts a *section* whose title is *text*. Due to a limitation of the `info` format, do not use `:` or `-` or `,` in section, subsection, title (chapter), etc. headings.

`@subsection{`*text*`}`
: starts a *subsection* whose title is *text*.

`@subsubsection{`*text*`}`
: starts a *subsubsection* whose title is *text*.

`@footnote{`*text*`}`
: places *text* in a *footnote*.

`@hfill` introduces horizontal filling space (may be ignored in certain formats).

`@bf{`*text*`}` *text* will be formatted in *bold face* or any other *strong face*.

`@em{`*text*`}` *text* will be formatted in *italics face* or any other *emphasis face*.

`@tt{`*text*`}` *text* will be formatted in a *fixed-width font* (i.e., *typewriter-like font*).

`@key{`*key*`}`
: *key* is the identifier of a *keyboard key* (i.e., a letter such as `a`, or a special key identifier such as `RET` or `DEL`) and will be formatted as ⟨LFD⟩ or in a fixed-width, typewriter-like font.

`@sp{`*N*`}` generates *N blank lines* of space. Forces also a paragraph break.

`@p` forces a *paragraph break*, in the same way as leaving one or more blank lines.

`@noindent`
: used at the beginning of a paragraph, states that the first line of the paragraph should not be indented. Useful, for example, for *avoiding indentation* on paragraphs that are continuations of other paragraphs, such as after a verbatim.

- **Indexing commands:**

  The following commands are used to mark certain words or sentences in the text as concepts, names of predicates, libraries, files, etc. The use of these commands is highly recommended, since it results in very useful indices with little effort.

  `@index{`*text*`}`
  : *text* will be printed in an emphasized font and will be included in the concept definition index (and also in the usage index). This command should be used for the first or *definitional* appearance(s) of a concept. The idea is that the concept definition index can be used to find the definition(s) of a concept.

  `@cindex{`*text*`}`
  : *text* will be included in the concept index (and also in the usage index), but it is not printed. This is used in the same way as above, but allows sending to the index a different text than the one that is printed in the text.

  `@concept{`*text*`}`
  : *text* will be printed (in a normal font). This command is used to mark that some text is a defined concept. In on-line manuals, a direct access to the corresponding concept definition may also be generated. A pointer to the place in which the `@`concept command occurs will appear only in the usage index.

  `@pred{`*predname*`}`
  : *predname* (which should be in functor/arity form) is the name of a predicate and will be printed in fixed-width, typewriter-like font. This command should be used when referring to a predicate (or a property or type) in a documentation string. A reference will be included in the usage index. In on-line manuals, a direct access to the corresponding predicate definition may also be generated.

  `@op{`*operatorname*`}`
  : *operatorname* (which should be in functor/arity form) is the name of an operator and will be printed in fixed-width, typewriter-like font. This command should be used when referring to an operator in a documentation string. A reference will be included in the usage index. In on-line manuals, a direct access to the corresponding operator definition may also be generated.

  `@decl{`*declname*`}`
  : *declname* (which should be in functor/arity form) is the name of a declaration and will be printed in fixed-width, typewriter-like font. This command should be used when referring to a declaration in a documentation string. A reference will be included in the usage index. In on-line manuals, a direct access to the corresponding declaration definition may also be generated.

  `@lib{`*libname*`}`
  : *libname* is the name of a library and will be printed in fixed-width, typewriter-like font. This command should be used when referring to

a module or library in a documentation string. A reference will be included in the usage index. In on-line manuals, a direct access to the corresponding module definition may also be generated.

`@apl{`*aplname*`}`
: *aplname* is the name of an application and will be printed in fixed-width, typewriter-like font. This command should be used when referring to an application in a documentation string. A reference will be included in the usage index.

`@file{`*filename*`}`
: *filename* is the name of a file and will be printed in fixed-width, typewriter-like font. This command should be used when referring to a file in a documentation string. A reference will be included in the usage index.

`@var{`*varname*`}`
: *varname* is the name of a variable and will be formatted in an emphasized font. Note that when referring to variable names in a `pred/1` declaration, such names should be enclosed in `@var` commands for the automatic documentation system to work correctly.

- **Referencing commands:**

  The following commands are used to introduce *bibliographic citations* and *references* to *sections*, *urls*, *email addresses*, etc.

  `@cite{`*keyword*`}`
  : *keyword* is the identifier of a *bibliographic entry*. Such entry is assumed to reside in one of a number of `bibtex` files (*.bib files*) . A reference in brackets ([ ]) is inserted in the text an the full reference is included at the end, with all other references, in an appendix. For example, `@cite{iso-prolog}` will introduce a citation to a bibliographic entry whose keyword is `iso-prolog`. The list of bibliography files which will be searched for a match is determined by `bibfile/1` fact of the `doccfg` file.

  `@ref{`*section title*`}`
  : introduces at point a reference to the section or node *section title*, where *section title* must be the exact *text* of the section title.

  `@href{`*URL*`}`
  : introduces at point a reference to the *Universal Resource Locator* (i.e., a *WWW address ' URL'*.

  `@href{`*URL*`}{`*text*`}`
  : introduces at point a reference to the Universal Resource Locator URL, associated to the text *text*.

  `@email{`*address*`}`
  : introduces at point a reference to *email address address*.

  `@email{`*text*`}{`*address*`}`
  : introduces at point a reference to the email address address, associated to the text *text*.

  `@author{`*text*`}`
  : *text* will be printed (in a normal font). This command is used to reference the name of an author (not necessarily establishing the module authorship).

- **Date and Version:**

`@today` prints the current *date*.

`@version` prints the *version* of the current manual.

- **Mathematics:**

  The following commands are used to format text in mathematical .

  `@math{`*text*`}`
  : in-line typeset the *text* formula.

  `@begin{displaymath}`
  : marks the beginning of a formula (useful for long formulas).

  `@end{displaymath}`
  : marks the end of the (long) formula.

  `@defmathcmd{`*cmd*`}{`*n*`}{`*def*`}`
  : defines the math command *cmd*, taking *n* arguments, which is expanded as *def*. Arguments are denotated as `#1`, ..., `#n` inside *def*.

  `@defmathcmd{`*cmd*`}{`*def*`}`
  : defines the math command *cmd*, which is expanded as *def* (with no arguments).

- **Inclusion commands:**

  The following commands are used to include code or strings of text as part of documentation. The latter may reside in external files or in the file being documented. The former must be part of the module being documented. There are also commands for inserting and scaling images.

  `@include{`*filename*`}`
  : the contents of *filename* will be included in-line, as if they were part of the string. This is useful for common pieces of documentation or storing in a separate file long explanations if they are perceived to clutter the source file.

  `@includecode{`*filename*`}`
  : the contents of *filename* will be included in-line and represented in verbatim mode (without `@begin{verbatim}` and `@end{verbatim}`). Commands within the file are not interpreted. This is useful for including code fragments in documentation.

  `@includecode{`*lang*`}{`*filename*`}`
  : same as above, but the contents of *filename* will be included in-line and represented in language *lang*.

  `@includeverbatim{`*filename*`}`
  : (DEPRECATED) as above, but the contents of the file are not necessarily represented in verbatim mode. Note that this only means that the file will be included as is. If you want the string to be represented in verbatim mode in the output, you must surround the `@includeverbatim{`*filename*`}` with `@begin{verbatim}` and `@end{verbatim}`.

  `@includefact{`*factname*`}`
  : it is assumed that the file being documented contains a fact of the predicate *factname*`/1`, whose argument is a character string. The contents of that character string will be included in-line, as if they were part of the documentation string. This is useful for *sharing pieces of text* between the documentation and the running code. An example is the text which explains the *usage of a command* (options, etc.).

`@includedef{`*predname*`}`
: it is assumed that the file being documented contains a definition for the predicate *predname*. The clauses defining this predicate will be included in-line, in verbatim mode, as if they were part of the documentation string.

`@image{`*epsfile*`}`
: *including an image* at point, contained in file *epsfile*. The *image file* should be in *encapsulated postscript* format.

  `@image{`*epsfile*`}{`*width*`}{`*height*`}` same as above, but *width* and *height* should be integers which provide a size (in points) to which the image will be scaled.

- **Accents and special characters:**

  The following commands can be used to insert *accents* and *special characters*.

`@‘{o}` ⇒ ò

`@’{o}` ⇒ ó

`@^{o}` ⇒ ô

`@..{o}` ⇒ ö

`@"{o}` ⇒ ö

`@~{o}` ⇒ õ

`@={o}` ⇒ ō

`@.{o}` ⇒ ȯ

`@u{o}` ⇒ ŏ

`@v{o}` ⇒ ǒ

`@H{o}` ⇒ ő

`@t{oo}` ⇒ o͡o

`@c{o}` ⇒ o̧

`@d{o}` ⇒ ọ

`@b{o}` ⇒ o̲

`@oe` ⇒ œ

`@OE` ⇒ Œ

`@ae` ⇒ æ

`@AE` ⇒ Æ

`@aa` ⇒ å

`@AA` ⇒ Å

`@o` ⇒ ø

`@O` ⇒ Ø

`@l` ⇒ ł

`@L` ⇒ Ł

`@ss` ⇒ ß

`@?` ⇒ ¿

`@!` ⇒ ¡

`@i` ⇒ ı

`@j` ⇒ ȷ

`@copyright` ⇒ ©

`@iso` ⇒ ⟨• ISO •⟩

`@bullet` ⇒ •

`@result` ⇒ ⇒

Usage: `stringcommand(CO)`

`CO` is a structure denoting a command that is admissible in strings inside assertions.

version_descriptor/1: REGTYPE

A structure denoting a complete version description:

```
version_descriptor([]).
version_descriptor(version(Version,Date)) :-
    version_number(Version),
    ymd_date(Date).
version_descriptor(version(Version,Date,Time)) :-
    version_number(Version),
    ymd_date(Date),
    time_struct(Time).
```

Usage: `version_descriptor(Descriptor)`

`Descriptor` is a complete version descriptor.

filetype/1: REGTYPE

Intended uses of a file:

```
filetype(module).
filetype(user).
filetype(include).
filetype(package).
filetype(application).
filetype(part).
filetype(documentation).
```

Usage: `filetype(Type)`

`Type` describes the intended use of a file.

stability_level/1: REGTYPE

The defined stability levels that can be attached to files:

```
stability_level(devel).
stability_level(alpha).
stability_level(beta).
stability_level(prod).
stability_level(devel(L)) :-
    docstring(L).
stability_level(alpha(L)) :-
    docstring(L).
stability_level(beta(L)) :-
    docstring(L).
stability_level(prod(L)) :-
    docstring(L).
```

Usage: `stability_level(Level)`

`Level` describes a level of stability.

7.3 Documentation on internals

doc/2: DECLARATION

This declaration provides one of the main means for adding *machine readable comments* to programs (the other one is adding *documentation strings* to assertions).

Usage 1: :- `doc(CommentType,TitleText)`.

Provides a *title* for the module, library, or application. When generating documentation automatically, the text in `TitleText` will be used appropriately (e.g., in the cover page as document title or as chapter title if part of a larger document). This will also be used as a brief description of the manual in on-line indices. There should be at most one of these declarations per module.

- *Example:*

```
:- doc(title,"Documentation-oriented assertions").
```

– *The following properties should hold upon exit:*

`title=CommentType` (`= /2`)

`TitleText` is a *documentation string*. (`docstring/1`)

Usage 2: :- `doc(CommentType,SubtitleText)`.

Provides a *subtitle*, an explanatory or alternate *title*. The subtitle will be displayed under the proper title.

- *Example:*

```
:- doc(title,"Dr. Strangelove").
:- doc(subtitle,"How I Learned to Stop Worrying and Love the Bomb").
```

– *The following properties should hold upon exit:*

`subtitle=CommentType` (`= /2`)

`SubtitleText` is a *documentation string*. (`docstring/1`)

Usage 3: :- `doc(CommentType,SubtitleText)`.

Provides additional *subtitle* lines. This can be, e.g., an explanation of the application to add to the title, the address of the author(s) of the application, etc. When generating documentation automatically, the text in `SubtitleText` will be used accordingly. Several of these declarations can appear per module, which is useful for, e.g., multiple line addresses.

- *Example:*

```
:- doc(subtitle_extra,"A Reference Manual").
:- doc(subtitle_extra,"Technical Report 1/1.0").
```

– *The following properties should hold upon exit:*

`subtitle_extra=CommentType` (`= /2`)

`SubtitleText` is a *documentation string.* (`docstring/1`)

Usage 4: :- `doc(CommentType,LogoName)`.

The name of the logo image for the manual.

– *The following properties should hold upon exit:*

`logo=CommentType` (`= /2`)

`LogoName` is any term. (`term/1`)

Usage 5: :- `doc(CommentType,AuthorText)`.

Provides the *author*(s) of the module or application. If present, when generating documentation for the module automatically, the text in `AuthorText` will be placed in the corresponding chapter or front page. There can be more than one of these declarations per module. In order for author indexing to work properly, please use one author declaration per author. If more explanation is needed (who did what when, etc.) use an acknowledgements comment.

- *Example:*

```
:- doc(author,"Alan Robinson").
```

– *The following properties should hold upon exit:*

`author=CommentType` (`= /2`)

`AuthorText` is a *documentation string.* (`docstring/1`)

Usage 6: :- `doc(CommentType,Text)`.

Provides the physical and electronic *address*, or any other contact information for the authors of the module or application.

- *Example:*

```
:- doc(address,"Syracuse University").
```

– *The following properties should hold upon exit:*

`address=CommentType` (`= /2`)

`Text` is a *documentation string.* (`docstring/1`)

Usage 7: :- `doc(CommentType,AckText)`.

Provides *acknowledgements* for the module. If present, when generating documentation for the module automatically, the text in `AckText` will be placed in the corresponding chapter or section. There can be only one of these declarations per module.

- *Example:*

```
:- doc(ack,"Module was written by Alan, but others helped.").
```

– *The following properties should hold upon exit:*

`ack=CommentType` (`= /2`)

`AckText` is a *documentation string.* (`docstring/1`)

Usage 8: :- `doc(CommentType,StabilityText)`.

Provides a *stability* level for the module. If present, when generating documentation for the module automatically, the level of stability of the module will be documented. There can be only one of these declarations per module. The second argument should be an atom, out of a number of predefined stability levels. The text included is predefined for each level. Alternatively, one of the admissible atoms can be used as the functor with a single argument, which contains the text to be included.

- *Examples:*

```
:- doc(stability,devel).
:- doc(stability,alpha).
:- doc(stability,beta).
:- doc(stability,prod).
:- doc(stability,alpha("My own comment on stability.")).
```

– *The following properties should hold upon exit:*

`stability=CommentType` (`=/2`)

`StabilityText` describes a level of stability. (`stability_level/1`)

Usage 9: :- `doc(CommentType,CopyrightText)`.

Provides a *copyright* text. This normally appears somewhere towards the beginning of a printed manual. There should be at most one of these declarations per module.

- *Example:*

```
:- doc(copyright,"Copyright @copyright{} 2001 FSF.").
```

– *The following properties should hold upon exit:*

`copyright=CommentType` (`=/2`)

`CopyrightText` is a *documentation string.* (`docstring/1`)

Usage 10: :- `doc(CommentType,SummaryText)`.

Provides a brief global explanation of the application or library. The text in `SummaryText` will be used as the *abstract* for the whole manual. There should be at most one of these declarations per module.

- *Example:*

```
:- doc(summary,"This is a @@bf@{very@} important library.").
```

– *The following properties should hold upon exit:*

`summary=CommentType` (`=/2`)

`SummaryText` is a *documentation string.* (`docstring/1`)

Usage 11: :- `doc(CommentType,CommentText)`.

Provides the main comment text for the module or application. When generating documentation automatically, the text in `CommentText` will be used as the *introduction* or *main body* of the corresponding chapter or manual. There should be at most one of these declarations per module. `CommentText` may use **sections** if substructure is needed.

- *Example:*

```
:- doc(module,"This module is the @@lib@{comments@} library.").
```

– *The following properties should hold upon exit:*

`module=CommentType` (`=/2`)

`CommentText` is a *documentation string.* (`docstring/1`)

Usage 12: :- `doc(CommentType,CommentText)`.

Provides additional comments text for a module or application. When generating documentation automatically, the text in `CommentText` will be used in one of the last sections or appendices of the corresponding chapter or manual. There should be at most one of these declarations per module. `CommentText` may use **subsections** if substructure is needed.

- *Example:*

```
:- doc(appendix,"Other module functionality...").
```

– *The following properties should hold upon exit:*

`appendix=CommentType` (`= /2`)

`CommentText` is a *documentation string.* (`docstring/1`)

Usage 13: :- `doc(CommentType,CommentText)`.

Provides a description of how the library should be loaded. Normally, this information is gathered automatically when generating documentation automatically. This declaration is meant for use when the module needs to be treated in some special way. There should be at most one of these declarations per module.

- *Example:*

```
:- doc(usage,"Do not use: still in development!").
```

– *The following properties should hold upon exit:*

`usage=CommentType` (`= /2`)

`CommentText` is a *documentation string.* (`docstring/1`)

Usage 14: :- `doc(CommentType,Section)`.

Insert a *program section* with name `Section`. Sectioning commands allow a structured separation of the program into parts. The division is only for documentation purposes, so visibility and scope of definitions is not affected by sectioning commands.

- *Example:*

```
:- doc(section,"Main Steps of the Algorithm").
```

– *The following properties should hold upon exit:*

`section=CommentType` (`= /2`)

`Section` is a *documentation string.* (`docstring/1`)

Usage 15: :- `doc(CommentType,SubSection)`.

Insert a *program subsection* with name `SubSection` (see *program section* command for more details).

- *Example:*

```
:- doc(subsection,"Auxiliary Definitions").
```

– *The following properties should hold upon exit:*

`subsection=CommentType` (`= /2`)

`SubSection` is a *documentation string.* (`docstring/1`)

Usage 16: :- `doc(CommentType,SubSubSection)`.

Insert a *program subsubsection* with name `SubSubSection` (see *program section* command for more details).

- *Example:*

```
:- doc(subsubsection,"Auxiliary Definitions").
```

– *The following properties should hold upon exit:*

`subsubsection=CommentType` (`= /2`)

`SubSubSection` is a *documentation string.* (`docstring/1`)

Usage 17: :- `doc(PredName,CommentText)`.

Provides an introductory comment for a given predicate, function, property, type, etc., denoted by `PredName`. When generating documentation for the module automatically, the text in `Text` will be used as the introduction of the corresponding predicate/function/... description. There should be at most one of these declarations per predicate, function, property, or type.

- *Example:*

```
:- doc(doc/2,"This declaration provides one of the main
   means for adding @@concept@{machine readable comments@} to
   programs.").
```

– *The following properties should hold upon exit:*

`PredName` is a predicate name. (`predname/1`)

`CommentText` is a *documentation string.* (`docstring/1`)

Usage 18: :- `doc(CommentType,CommentText)`.

Documents a known *bug* or *planned improvement* in the module or application. Several of these declarations can appear per module. When generating documentation automatically, the text in the `Text` fields will be used as items in an itemized list of module or application bugs.

- *Example:*

```
:- doc(bug,"Comment text still has to be written by user.").
```

– *The following properties should hold upon exit:*

`bug=CommentType` (`= /2`)

`CommentText` is a *documentation string.* (`docstring/1`)

Usage 19: :- `doc(Version,CommentText)`.

Provides a means for keeping a *log of changes*. `Version` contains the *version number* and date corresponding to the change and `CommentText` an explanation of the change. Several of these declarations can appear per module. When generating documentation automatically, the texts in the different `CommentText` fields typically appear as items in an itemized list of changes. The emacs Ciao mode helps tracking version numbers by prompting for version comments when files are saved. This mode requires version comments to appear in reverse chronological order (i.e., the topmost comment should be the most recent one).

- *Example:*

```
:- doc(version(1*1+21,1998/04/18,15:05*01+'EST'), "Added some
   missing comments.  (Manuel Hermenegildo)").
```

– *The following properties should hold upon exit:*

`Version` is a complete version descriptor. (`version_descriptor/1`)

`CommentText` is a *documentation string.* (`docstring/1`)

Usage 20: :- `doc(CommentType,VersionMaintenanceType)`.

Defines the type of version maintenance that should be performed by the `emacs` Ciao mode.

- *Example:*

```
:- doc(version_maintenance,dir('../version')).
Version control info is kept in directory @tt{../version}. See
the definition of @pred{version_maintenance_type/1} for more
information on the different version maintenance modes. See the
documentation on the @index{emacs Ciao mode} in the Ciao manual
for information on how to automatically insert version control
@decl{doc/2} declarations in files.

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 @concept{emacs Ciao mode} and to turn version
maintenance off.  Setting the version maintenance mode in this
way has the disadvantage that @apl{lpdoc} will not be aware of
the type of version maintenance being performed (the lines above
are comments for Prolog). However, this can be useful in fact
for setting the @index{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 properties should hold upon exit:*

`version_maintenance=CommentType` (`= /2`)

`VersionMaintenanceType` is a type of version maintenance for a file. (`version_maintenance_type/1`)

Usage 21: :- `doc(CommentType,PredName)`.

This is a special case that is used to control which predicates are included in the documentation. Normally, only exported predicates are documented. A declaration `:- doc(doinclude,PredName).` forces documentation for predicate (or type, property, function, ...) `PredName` to be included even if `PredName` is not exported. Also, if `PredName` is reexported from another module, a declaration `:- doc(doinclude,PredName).` will force the documenation for `PredName` to appear directly in this module.

- *Example:*

```
:- doc(doinclude,p/3).
```

– *The following properties should hold upon exit:*

`doinclude=CommentType` (`= /2`)

`PredName` is a predicate name. (`predname/1`)

Usage 22: :- `doc(CommentType,PredName)`.

A different usage which allows the second argument of `:- doc(doinclude,...)` to be a list of predicate names.

– *The following properties should hold upon exit:*

`doinclude=CommentType` (`= /2`)

`PredName` is a list of `prednames`. (`list/2`)

Usage 23: :- `doc(CommentType,PredName)`.

This is similar to the previous usage but has the opposite effect: it signals that an exported predicate should *not* be included in the documentation.

- *Example:*

```
:- doc(hide,p/3).
```

– *The following properties should hold upon exit:*

`hide=CommentType` (`= /2`)

`PredName` is a predicate name. (`predname/1`)

Usage 24: :- `doc(CommentType,PredName)`.

A different usage which allows the second argument of `:- doc(hide,...)` to be a list of predicate names.

– *The following properties should hold upon exit:*

`hide=CommentType` (`= /2`)

`PredName` is a list of `prednames`. (`list/2`)

Usage 25: :- `doc(CommentType,FileType)`.

Provides a way of defining the intended use of the file. This use is normally easily inferred from the contents of the file itself, and therefore such a declaration is in general not needed. The exception is the special case of include files and Ciao packages, which are typically indistiguishable from normal *user* files (i.e., files which are not modules), but are however quite different in their form of use (they are loaded via `include/1` or `use_package/1` declarations instead of `ensure_loaded/1`) and effect (since they are included, they 'export' operators, declarations, etc.). Typically, it is assumed by default that files which are not modules will be used as include files or packages. Thus, a `doc/2` declaration of this kind strictly only needs to be added to user-type files.

Example:

```
:- doc(filetype,user).

There is another special case: the value @tt{part}. This
@em{filetype} is used to flag files which serve as introductions
to boundaries between major @index{parts in large
documents}. See @ref{Splitting large documents into parts} for
details.
```

– *The following properties should hold upon exit:*

`filetype=CommentType` (`= /2`)

`FileType` describes the intended use of a file. (`filetype/1`)

Usage 26: :- `doc(CommentType,FileName)`.

Do not document anything that comes from a file whose name (after taking away the path and the suffix) is `FileName`. This is used for example when documenting packages

to avoid the documenter from including documentation of certain other packages which the package being documented uses.

- *Example:*

```
:- doc(nodoc,regtypes).
```

– *The following properties should hold upon exit:*

`nodoc=CommentType` (`= /2`)

`FileName` is an atom. (`atm/1`)

version_number/1: REGTYPE

`Version` is a structure denoting a complete version number (major version, minor version, and patch number):

```
version_number(Major*Minor+Patch) :-
    int(Major),
    int(Minor),
    int(Patch).
```

Usage: `version_number(Version)`

`Version` is a complete version number

ymd_date/1: REGTYPE

A Year/Month/Day structure denoting a date:

```
ymd_date(Y/M/D) :-
    int(Y),
    int(M),
    int(D).
```

.

Usage: `ymd_date(Date)`

`Date` is a Year/Month/Day structure denoting a date.

time_struct/1: REGTYPE

A struture containing time information:

```
time_struct(Hours:Minutes*Seconds+TimeZone) :-
    int(Hours),
    int(Minutes),
    int(Seconds),
    atm(TimeZone).
```

Usage: `time_struct(Time)`

`Time` contains time information.

version_maintenance_type/1: REGTYPE

Possible kinds of version maintenance for a file:

```
version_maintenance_type(on).
version_maintenance_type(off).
version_maintenance_type(dir(Path)) :-
    atm(Path).
```

- `on`: version numbering is maintained locally in the file in which the declaration occurs, i.e., an independent version number is kept for this file and the current version is given by the most recent `doc(version(...),...)` declaration.
- `off`: no version numbering maintained.
- `dir(Path)`: version numbering is maintained (globally) in directory `Path`. This is useful for maintaining a common global version for an application which involves several files.

The automatic maintenance of version numbers is typically done by the Ciao `emacs` mode.

Usage: `version_maintenance_type(Type)`

`Type` is a type of version maintenance for a file.

7.4 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`strings`.

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `dcg`, `assertions`, `assertions/assertions_basic`, `regtypes`, `fsyntax`.

8 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`).

8.1 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:

```
@tt{%!} @em{Comment}      (or)    @tt{/*!} @em{Comment} @tt{*/}
@tt{%}  @em{...}
```

which is used to write arbitrary chunks of documentation (usually referring to the code after them).

```
@tt{%<} @em{Comment}      (or)    @tt{/*<} @em{Comment} @tt{*/}
@tt{%}  @em{...}
```

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.

8.2 Relation with comment assertions

This package allows using an alternative syntax for machine-readable comments. Essentially, most comments of the form:

```
:- doc(@em{CommentType},@em{Body}).
```

can be written as:

```
@tt{%!} @@@em{CommentType} @em{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.

8.3 Usage and interface

- **Library usage:**
  `:- use_package(doccomments).`
  or
  `:- module(...,...,[doccomments]).`
- **Implicit imports:**
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

9 Documentation markdown language

Author(s): The Ciao Development Team.

This section describes the *markdown*-flavor lightweight syntax for writing documentation. `LPdoc` supports combining both markup and markdown syntax in the same document. Internally, all markdown syntax is translated to the equivalent markup before processing.

9.1 Syntax

9.1.1 Sections

Sections are marked with one or more `#` symbols at the beginning of a line:

```
# Sections and subsections

Text for the section.

## Subsection

Text for the subsection.

### Subsubsection

Text for the subsubsection.
```

9.1.2 Styling text

Text surrounded by `_` (or `*`), `__` (or `**`) produces emphasized or bold text:

```
Text can be _emphasized_ (*emphasized*) or __bold__ (**bold**).
```

produces the following results:

Text can be *emphasized* (*emphasized*) or **bold** (**bold**).

9.1.3 Paragraphs

Paragraphs are break by blank lines, as follows:

```
Begin of
a paragraph.
This paragraph ends here.

Begin of a new paragraph.
This paragraph ends here.

The last new paragraph. This paragraph ends here.
```

which produce the following results:

Begin of a paragraph. This paragraph ends here.

Begin of a new paragraph. This paragraph ends here.

The last new paragraph. This paragraph ends here.

9.1.4 Lists

Lists are composed of lines beginning with `-` items, which can be nested:

```
- item 1
  - subitem 1-1
  - subitem 1-2
    - subitem 1-2-1
  - subitem 1-3
- item 2
```

10 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 11 [Types and properties related to assertions], page 63 module.

10.1 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.

10.2 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.

10.3 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`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions/assertions_basic`.

10.4 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 @var{L}.".
:- pred length(L,N) : var * integer => list * integer
    # "Outputs @var{L} of length @var{N}.".
:- pred length(L,N) : list * integer => list * integer
    # "Checks that @var{L} is of length @var{N}.".
```

Usage: :- `pred(AssertionBody)`.

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

`AssertionBody` is an assertion body. (`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. (`assrt_status/1`)

`AssertionBody` is an assertion body. (`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. (`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. (`assrt_status/1`)

`AssertionBody` is a call assertion body. (`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. (`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. (`assrt_status/1`)

`AssertionBody` is a predicate assertion body. (`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. (`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. (`assrt_status/1`)

`AssertionBody` is a comp assertion body. (`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 12 [Declaring regular types], page 69 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. (`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. (`assrt_status/1`)

`AssertionBody` is an assertion body. (`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. (`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. (`assrt_status/1`)

`AssertionBody` is a predicate assertion body. (`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. (`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. (`assrt_status/1`)

`AssertionBody` is a call assertion body. (`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. (`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. (`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. (`assrt_status/1`)

`AssertionBody` is a predicate assertion body. (`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) # "@var{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)
   # "@var{A} is bound upon predicate entry.".
```

Usage: :- `modedef(AssertionBody)`.

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

`AssertionBody` is an assertion body. (`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. (`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. (`assrt_status/1`)

`AssertionBody` is an assertion body. (`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. (`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. (`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. (`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. (`docstring/1`)

– *The following properties should hold globally:*

DEPRECATED. (`deprecated/1`)

10.5 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. (`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. (`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. (`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. (`property_conjunction/1`)

11 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.

11.1 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`.

11.2 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.

11.3 Documentation on imports

This module has the following direct dependencies:

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

12 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) # "@var{X} is a tree.".
```

instead of the more cumbersome:

```
:- prop tree(X) + regtype # "@var{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.

12.1 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.

12.2 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`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.
  - *Packages:*
    `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`.

12.3 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 13 [Basic data types and properties], page 75.

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. (`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. (`assrt_status/1`)

`AssertionBody` is an assertion body. (`assrt_body/1`)

13 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.

13.1 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`.

13.2 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. (`native/1`)

Other properties:

`term(X)`

- *The following properties hold globally:*
  `term(X)` is side-effect `free`. (`sideff/2`)

`term(X)`

– *The following properties hold globally:*
`term(X)` is evaluable at compile-time. (`eval/1`)

`term(X)`
– *The following properties hold globally:*
`term(X)` is equivalent to `true`. (`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. (`native/1`)

Other properties:

`int(T)`
– *The following properties hold globally:*
`int(T)` is side-effect `free`. (`sideff/2`)

`int(T)`
– *If the following properties hold at call time:*
`T` is currently a term which is not a free variable. (`nonvar/1`)
then the following properties hold globally:
`int(T)` is evaluable at compile-time. (`eval/1`)
All calls of the form `int(T)` are deterministic. (`is_det/1`)

`int(T)`
– *The following properties hold upon exit:*
`T` is an integer. (`int/1`)
– *The following properties hold globally:*
Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`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. (`native/1`)

Other properties:

`nnegint(T)`
– *The following properties hold globally:*
`nnegint(T)` is side-effect `free`. (`sideff/2`)

`nnegint(T)`

– *If the following properties hold at call time:*

`T` is currently a term which is not a free variable. (`nonvar/1`)

then the following properties hold globally:

`nnegint(T)` is evaluable at compile-time. (`eval/1`)

`nnegint(T)`

– *The following properties hold upon exit:*

`T` is a non-negative integer. (`nnegint/1`)

– *The following properties hold globally:*

Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`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. (`native/1`)

Other properties:

`flt(T)`

– *The following properties hold globally:*

`flt(T)` is side-effect `free`. (`sideff/2`)

`flt(T)`

– *If the following properties hold at call time:*

`T` is currently a term which is not a free variable. (`nonvar/1`)

then the following properties hold globally:

`flt(T)` is evaluable at compile-time. (`eval/1`)

All calls of the form `flt(T)` are deterministic. (`is_det/1`)

`flt(T)`

– *The following properties hold upon exit:*

`T` is a float. (`flt/1`)

– *The following properties hold globally:*

Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`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. (`native/1`)

Other properties:
`num(T)`
– *The following properties hold globally:*
`num(T)` is side-effect `free`. (`sideff/2`)
`num(T)` is binding insensitive. (`bind_ins/1`)

`num(T)`
– *If the following properties hold at call time:*
`T` is currently a term which is not a free variable. (`nonvar/1`)
then the following properties hold globally:
`num(T)` is evaluable at compile-time. (`eval/1`)
All calls of the form `num(T)` are deterministic. (`is_det/1`)

`num(T)`
– *The following properties hold upon exit:*
`T` is a number. (`num/1`)
– *The following properties hold globally:*
Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`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. (`native/1`)

Other properties:
`atm(T)`
– *The following properties hold globally:*
`atm(T)` is side-effect `free`. (`sideff/2`)

`atm(T)`
– *If the following properties hold at call time:*
`T` is currently a term which is not a free variable. (`nonvar/1`)
then the following properties hold globally:
`atm(T)` is evaluable at compile-time. (`eval/1`)
All calls of the form `atm(T)` are deterministic. (`is_det/1`)

`atm(T)`
– *The following properties hold upon exit:*
`T` is an atom. (`atm/1`)
– *The following properties hold globally:*
Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`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. (`native/1`)

Other properties:

`struct(T)`

– *The following properties hold globally:*

`struct(T)` is side-effect `free`. (`sideff/2`)

`struct(T)`

– *If the following properties hold at call time:*

`T` is currently a term which is not a free variable. (`nonvar/1`)

then the following properties hold globally:

`struct(T)` is evaluable at compile-time. (`eval/1`)

`struct(T)`

– *The following properties hold upon exit:*

`T` is a compound term. (`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. (`native/1`)

Other properties:

`gnd(T)`

– *The following properties hold globally:*

`gnd(T)` is side-effect `free`. (`sideff/2`)

`gnd(T)`

– *If the following properties hold at call time:*

`T` is currently ground (it contains no variables). (`ground/1`)

then the following properties hold globally:

`gnd(T)` is evaluable at compile-time. (`eval/1`)

All calls of the form `gnd(T)` are deterministic. (`is_det/1`)

`gnd(T)`

– *The following properties hold upon exit:*

`T` is ground. (`gnd/1`)

– *The following properties hold globally:*

Indicates the type of test that a predicate performs. Required by the nonfailure analyisis. (`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. (`native/1`)

Other properties:

`gndstr(T)`

– *The following properties hold globally:*
`gndstr(T)` is side-effect `free`. (`sideff/2`)

`gndstr(T)`

– *If the following properties hold at call time:*
`T` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold globally:
`gndstr(T)` is evaluable at compile-time. (`eval/1`)
All calls of the form `gndstr(T)` are deterministic. (`is_det/1`)

`gndstr(T)`

– *The following properties hold upon exit:*
`T` is a ground compound term. (`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`. (`sideff/2`)

`constant(T)`

– *If the following properties hold at call time:*
`T` is currently a term which is not a free variable. (`nonvar/1`)
then the following properties hold globally:
`constant(T)` is evaluable at compile-time. (`eval/1`)
All calls of the form `constant(T)` are deterministic. (`is_det/1`)

`constant(T)`

– *The following properties hold upon exit:*
`T` is an atomic term (an atom or a number). (`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`. (`sideff/2`)

`cgoal(T)`
– *If the following properties hold at call time:*
`T` is currently a term which is not a free variable. (`nonvar/1`)
then the following properties hold globally:
`cgoal(T)` is evaluable at compile-time. (`eval/1`)
All calls of the form `cgoal(T)` are deterministic. (`is_det/1`)

`cgoal(T)`
– *The following properties hold upon exit:*
`T` is currently a term which is not a free variable. (`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`. (`sideff/2`)

`callable(T)`
– *If the following properties hold at call time:*
`T` is currently a term which is not a free variable. (`nonvar/1`)
then the following properties hold globally:
`callable(T)` is evaluable at compile-time. (`eval/1`)
All calls of the form `callable(T)` are deterministic. (`is_det/1`)

`callable(T)`
– *The following properties hold upon exit:*
`T` is currently a term which is not a free variable. (`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`. (`sideff/2`)

`operator_specifier(X)`

– *If the following properties hold at call time:*

`X` is currently a term which is not a free variable. (`nonvar/1`)

then the following properties hold globally:

`operator_specifier(X)` is evaluable at compile-time. (`eval/1`)

All calls of the form `operator_specifier(X)` are deterministic. (`is_det/1`)

Goal `operator_specifier(X)` produces `7` solutions. (`relations/2`)

`operator_specifier(T)`

– *The following properties hold upon exit:*

`T` specifies the type and associativity of an operator. (`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`. (`sideff/2`)

`list(L)`

– *If the following properties hold at call time:*
`L` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold globally:
`list(L)` is evaluable at compile-time. (`eval/1`)
All calls of the form `list(L)` are deterministic. (`is_det/1`)

`list(T)`
– *The following properties hold upon exit:*
`T` is a list. (`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`. (`sideff/2`)

`list(T,L)`
– *If the following properties hold at call time:*
`T` is currently ground (it contains no variables). (`ground/1`)
`L` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold globally:
`list(T,L)` is evaluable at compile-time. (`eval/1`)

`list(T,X)`
– *The following properties hold upon exit:*
`X` is a list. (`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`. (`sideff/2`)

`nlist(T,L)`

– *If the following properties hold at call time:*
`T` is currently ground (it contains no variables). (`ground/1`)
`L` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold globally:
`nlist(T,L)` is evaluable at compile-time. (`eval/1`)

`nlist(T,X)`
– *The following properties hold upon exit:*
`X` is any term. (`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`. (`sideff/2`)
`member(X,L)` is binding insensitive. (`bind_ins/1`)

`member(X,L)`
– *If the following properties hold at call time:*
`L` is a list. (`list/1`)
then the following properties hold globally:
`member(X,L)` is evaluable at compile-time. (`eval/1`)

`member(_X,L)`
– *The following properties hold upon exit:*
`L` is a list. (`list/1`)

`member(X,L)`
– *If the following properties hold at call time:*
`L` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold upon exit:
`X` is currently ground (it contains no variables). (`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 `T`s.

Meta-predicate with arguments: `sequence(pred(1),?)`.

Other properties:

`sequence(T,S)`
– *The following properties hold globally:*
`sequence(T,S)` is side-effect `free`. (`sideff/2`)

`sequence(T,S)`

– *If the following properties hold at call time:*
`T` is currently ground (it contains no variables). (`ground/1`)
`S` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold globally:
`sequence(T,S)` is evaluable at compile-time. (`eval/1`)

`sequence(T,E)`

– *The following properties hold upon exit:*
`T` is currently ground (it contains no variables). (`ground/1`)
`E` is currently a term which is not a free variable. (`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`. (`sideff/2`)

`sequence_or_list(T,S)`

– *If the following properties hold at call time:*
`T` is currently ground (it contains no variables). (`ground/1`)
`S` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold globally:
`sequence_or_list(T,S)` is evaluable at compile-time. (`eval/1`)

`sequence_or_list(T,E)`

– *The following properties hold upon exit:*
`T` is currently ground (it contains no variables). (`ground/1`)
`E` is currently a term which is not a free variable. (`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`. (`sideff/2`)

`character_code(T)`

– *If the following properties hold at call time:*
`T` is currently a term which is not a free variable. (`nonvar/1`)
then the following properties hold globally:
`character_code(T)` is evaluable at compile-time. (`eval/1`)

`character_code(I)`

– *The following properties hold upon exit:*
`I` is an integer which is a character code. (`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`. (`sideff/2`)

`string(T)`

– *If the following properties hold at call time:*
`T` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold globally:
`string(T)` is evaluable at compile-time. (`eval/1`)

`string(T)`

– *The following properties hold upon exit:*
`T` is a string (a list of character codes). (`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`. (`sideff/2`)

`bytelist(T)`

– *If the following properties hold at call time:*
`T` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold globally:
`bytelist(T)` is evaluable at compile-time. (`eval/1`)

`bytelist(T)`

– *The following properties hold upon exit:*
`T` is list of bytes. (`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`. (`sideff/2`)

`predname(P)`

– *If the following properties hold at call time:*

`P` is currently ground (it contains no variables). (`ground/1`)

then the following properties hold globally:

`predname(P)` is evaluable at compile-time. (`eval/1`)

`predname(P)`

– *The following properties hold upon exit:*

`P` is a predicate name. (`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`. (`sideff/2`)

`atm_or_atm_list(T)`

– *If the following properties hold at call time:*

`T` is currently ground (it contains no variables). (`ground/1`)

then the following properties hold globally:

`atm_or_atm_list(T)` is evaluable at compile-time. (`eval/1`)

`atm_or_atm_list(T)`

– *The following properties hold upon exit:*

`T` is an atom or a list of atoms. (`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). (`ground/1`)
`Prop` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold globally:
`compat(Term,Prop)` is evaluable at compile-time. (`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`. (`sideff/2`)

`inst(Term,Prop)`

– *If the following properties hold at call time:*
`Term` is currently ground (it contains no variables). (`ground/1`)
`Prop` is currently ground (it contains no variables). (`ground/1`)
then the following properties hold globally:
`inst(Term,Prop)` is evaluable at compile-time. (`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`. (`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`. (`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`. (`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`. (`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`. (`sideff/2`)

`not_further_inst(G,V)` is not checked during run-time checking. (`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. (`cgoal/1`)

`X` is an element of `[free,soft,hard]`. (`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. (`native/1`)

`sideff(G,X)` is side-effect `free`. (`sideff/2`)

`sideff(G,X)` is not checked during run-time checking. (`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`. (`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`. (`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`. (`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. (`cgoal/1`)

Meta-predicate with arguments: `rtcheck(goal)`.

Other properties:

`rtcheck(G)`

– *The following properties hold globally:*

`rtcheck(G)` is side-effect `free`. (`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. (`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).

(`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`. (`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. (`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`. (`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.

13.3 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`native_props`, `terms_check`.

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `hiord_rt`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `nortchecks`, `nativeprops`.

14 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.

15 Properties related to sharing/aliasing, groundness

These properties are part of the `native_props` library.

15.1 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`.

15.2 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)`. (`native/2`)
  `mshare(X)` is not checked during run-time checking. (`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)`. (`native/2`)
  `mshare(Xs,Xss)` is not checked during run-time checking. (`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]])`. (`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)`. (`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. (`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. (`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)`. (`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)`. (`native/2`)

`clique(X)` is not checked during run-time checking. (`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)`. (`native/2`)

`clique_1(X)` is not checked during run-time checking. (`no_rtcheck/1`)

15.3 Documentation on imports

This module has the following direct dependencies:

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

16 Properties related to determinacy, failure, choice-points

These properties are part of the `native_props` library.

16.1 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`.

16.2 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. (`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. (`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`. (`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`. (`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. (`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`. (`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`. (`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. (`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. (`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. (`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. (`no_rtcheck/1`)

Meta-predicate with arguments: `possibly_fails(goal)`.

16.3 Documentation on imports

This module has the following direct dependencies:

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

17 Properties related to cardinality and exact solutions

These properties are part of the `native_props` library.

17.1 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`.

17.2 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. (`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. (`cgoal/1`)
  `N` is an integer. (`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. (`cgoal/1`)
  `Check` is a term which represents a goal, i.e., an atom or a structure. (`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. (`cgoal/1`)
`N` is an integer. (`int/1`)
then the following properties should hold globally:
The runtime check of this property is `unimplemented`. (`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. (`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. (`cgoal/1`)
`Sols` is a list. (`list/1`)

17.3 Documentation on imports

This module has the following direct dependencies:

– *Internal (engine) modules:*
`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*
`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

18 Properties related to exceptions and signals

These properties are part of the `native_props` library.

18.1 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`.

18.2 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. (`list/1`)
  then the following properties should hold globally:
  The runtime check of this property is `unimplemented`. (`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`. (`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,?)`.

18.3 Documentation on imports

This module has the following direct dependencies:

– *Internal (engine) modules:*
`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*
`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

19 Properties related to side effects

These properties are part of the `native_props` library.

19.1 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`.

19.2 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. (`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. (`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. (`no_rtcheck/1`)

Meta-predicate with arguments: `sideff_hard(goal)`.

19.3 Documentation on imports

This module has the following direct dependencies:

– *Internal (engine) modules:*
`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*
`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

20 Properties related to polyhedral constraints

These properties are part of the `native_props` library.

20.1 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`.

20.2 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. (`native/1`)

20.3 Documentation on imports

This module has the following direct dependencies:

- *Internal (engine) modules:*
  `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

21 Properties related to data sizes, cost, termination

These properties are part of the `native_props` library.

21.1 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`.

21.2 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. (`term/1`)

`Y` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`size(X,Y)` is not checked during run-time checking. (`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. (`approx/1`)

`X` is any term. (`term/1`)

`Y` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`size(A,X,Y)` is not checked during run-time checking. (`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. (`approx/1`)

`M` is a term size metric. (`measure_t/1`)

`X` is any term. (`term/1`)

`Y` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`size(A,M,X,Y)` is not checked during run-time checking. (`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. (`term/1`)

`Y` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`size_lb(X,Y)` is not checked during run-time checking. (`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. (`term/1`)

`Y` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`size_ub(X,Y)` is not checked during run-time checking. (`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. (`term/1`)

`Y` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`size_o(X,Y)` is not checked during run-time checking. (`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. (`cgoal/1`)

`Var` is any term. (`term/1`)

`Metric` is a term size metric. (`measure_t/1`)

then the following properties should hold globally:

`size_metric(Head,Var,Metric)` is not checked during run-time checking. (`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. (`cgoal/1`)

`Approx` represents an approximation. (`approx/1`)

`Var` is any term. (`term/1`)

`Metric` is a term size metric. (`measure_t/1`)

then the following properties should hold globally:

`size_metric(Head,Approx,Var,Metric)` is not checked during run-time checking. (`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. (`cgoal/1`)

`Y` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`steps_lb(X,Y)` is not checked during run-time checking. (`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. (`cgoal/1`)

`Y` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`steps_ub(X,Y)` is not checked during run-time checking. (`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. (`cgoal/1`)

`Y` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`steps(X,Y)` is not checked during run-time checking. (`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. (`cgoal/1`)

`Y` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`steps_o(X,Y)` is not checked during run-time checking. (`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. (`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. (`cgoal/1`)

`Resource` is the name/identifier of a resource, either an atom or a compound term. (`resource_id/1`)

`Lower` is a cost expression. (`cost_expression/1`)

`Upper` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`costb(Goal,Resource,Lower,Upper)` is not checked during run-time checking. (`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. (`cgoal/1`)

`Approx` represents an approximation. (`approx/1`)

`Resource` is the name/identifier of a resource, either an atom or a compound term. (`resource_id/1`)

`Expr` is a cost expression. (`cost_expression/1`)

then the following properties should hold globally:

`cost(Goal,Approx,Resource,Expr)` is not checked during run-time checking. (`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. (`cgoal/1`)

then the following properties should hold globally:

`terminates(X)` is not checked during run-time checking. (`no_rtcheck/1`)

Meta-predicate with arguments: `terminates(goal)`.

21.3 Documentation on imports

This module has the following direct dependencies:

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `regtypes`.

22 Meta-properties

Author(s): Francisco Bueno.

This library allows the use of some meta-constructs which provide for specifying properties of terms which are unknown at the time of the specification, or expressed with a shorthand for the property definition, i.e., without really defining it.

An example of such use is an assertion which specifies that any property holding upon call will also hold upon exit:

```
:- pred p(X) : Prop(X) => Prop(X).
```

Another example is using shorthands for properties when documenting:

```
:- pred p(X) : regtype(X,(^(list;list);list)).
```

(See below for an explanation of such a regular type.)

22.1 Usage and interface

- **Library usage:**
  `:- use_module(library(assertions/meta_props))`
  or also as a package `:- use_package(metaprops)`.
  Note the different names of the library and the package.
- **Exports:**
  - *Properties:*
    `call/2`, `prop/2`, `regtype/2`.
  - *Multifiles:*
    `callme/2`.

22.2 Documentation on exports

call/2: PROPERTY

`call(P,A)`

`A` has property `P` (provided that `P` is a property). Equivalent to `P(A)`.

Usage: `call(P,A)`

`A` has property `P`.

- *If the following properties should hold at call time:*
  `P` is a term which represents a goal, i.e., an atom or a structure. (`cgoal/1`)

prop/2: PROPERTY

Usage: `prop(A,P)`

`A` has property `P`.

- *If the following properties should hold at call time:*
  `P` has property `^((cgoal;prop_abs))`. (`prop/2`)

regtype/2: PROPERTY

Usage: `regtype(A,T)`

`A` is of type `T`.

– *If the following properties should hold at call time:*

`T` has property `^((regtype;prop_abs))`. (`prop/2`)

22.3 Documentation on multifiles

callme/2: PREDICATE

(User defined.) A hook predicate you have to define as `callme(P,X):- P(X), !.` in the program that uses this library. This is done automatically if the package is used instead of the library module (but then you *should not* define `callme/2` in your program).

Usage: `callme(A,B)`

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

`A` is a term which represents a goal, i.e., an atom or a structure. (`cgoal/1`)

The predicate is *multifile*.

22.4 Documentation on internals

prop_abs/1: PROPERTY

`prop_abs(Prop)`

`Prop` is a *property abstraction*, i.e., a *parametric property*, or a term formed of property abstractions, where the functors used in the term are escaped by `^`.

One particular case of property abstractions are *parametric regular type abstractions*, i.e., a parametric type functor or a `^`-escaped term formed of regular type abstractions.

Such abstractions are a short-hand for a corresponding regular type (correspondingly, property). For example, the following abstraction:

```
^(list;list);list
```

denotes terms of the form `(X;Y)` where `list(X)` and `list(Y)` hold and also terms `T` such that `list(T)` holds. It is equivalent to the regular type:

```
abstract_type((X;Y)):- list(X), list(Y).
abstract_type(T):- list(T).
```

Usage: `prop_abs(Prop)`

`Prop` is a property abstraction.

22.5 Documentation on imports

This module has the following direct dependencies:

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `hiord_rt`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `hiord`.

23 An example - Documenting a library module

Author(s): Manuel Hermenegildo.

An example of the use of `lpdoc` is this manual, which can be built in the `doc` directory of the `lpdoc` distribution. Other examples of manuals generated using `lpdoc` can be found in the `Ciao` system and preprocessor `doc` directories (i.e., most of the `Ciao` manuals are generated using `lpdoc`) and in the various `Ciao` bundles (typically each has an `lpdoc` manual). Some simpler examples can be found in the `examples` directory of the `lpdoc` distribution. In particular, the chapter following this one contains the documentation generated automatically for the module defined by file `examples/example_module.pl` (which for simplicity contains only assertions, i.e., no actual code) and which is included in source form below. Comparing this code with the output in the following chapter illustrates the use and some of the capabilities of `lpdoc`:

```
%% The module headers produce documentation on the module interface
%% Exported predicates (+ properties and types) are documented by default
:- module(example_module,
      [bar/1,baz/1,aorb/1,tree_of/2,list_or_aorb/2,q/2,r/1, p/1, p/5, u/3,
       long/1, w/1, mytype/1, t/5, s/1, q/1],
      [dynamic,assertions,basicmodes,fsyntax,regtypes,hiord,nativeprops]).

%% We import two types: list/1 and list/2 (now in basic_props, which is
%% exported by default from assertions).

%% We reexport list/1
:- reexport(engine(basic_props), [list/1]).

:- use_module(library(lists), [length/2]).
%:- use_module(bar).
:- ensure_loaded(foo).

%% "doc" declarations provide additional information
:- doc(title,"Auto documenter output for the example module").

:- doc(author,"Anonymous Author 1").
:- doc(author,"Anonymous Author 2").

%%% These use a predefined message:
% :- doc(stability,devel).
:- doc(stability,alpha).
% :- doc(stability,beta).
% :- doc(stability,prod).
%%% These use a custom message:
% :- doc(stability,devel("This is a custom message...")).
% :- doc(stability,alpha("This is a @bf{custom message}...")).
% :- doc(stability,beta("This is a custom message...")).
% :- doc(stability,prod("This is a custom message...")).
%%% These produce an error:
% :- doc(stability,"").
% :- doc(stability,foo).
% :- doc(stability,foo("This is a custom message...")).

:- doc(summary,"This is a brief summary description of the module
   or file. In this case the file is a library.").
```

```
:- doc(module,"This is where general comments on the file go. In
   this case the file is a library which contains some assertion examples
   for testing the @em{automatic documentation system}. ").

%% An example of a comment documenting a bug
:- doc(bug,"Library is hard to execute: no actual code!").

%% Standard declarations are documented with the corresponding predicate
:- data r/1.
:- dynamic q/2.
:- multifile p/3.
:- dynamic p/3.
:- meta_predicate p(?,:,?).

%% Uncommenting this would make these not appear in the documentation
%% :- doc(hide,[bar/1,baz/1]).

%% This is a type definition in Prolog syntax: declaration and code
:- regtype bar(X) # "@var{X} is an acceptable kind of bar.".

bar(night).
bar(day).

%% This is another type definition in Prolog syntax, with no comment.
:- regtype baz/1.

baz(a).
baz(b).

%% Two type definitions in 'typedef' syntax (will be expanded to code as above)
%% :- typedef aorb ::= ^a;^b.
%% :- typedef listof_or_aorb(X) ::= list(X);aorb.

%% Using functional notation:
:- regtype aorb/1.

aorb := a.
aorb := b.

:- regtype tree_of/2.
:- meta_predicate tree_of(pred(1),?).
tree_of(_) := void.
tree_of(T) := tree(~call(T),~tree_of(T),~tree_of(T)).

:- regtype list_or_aorb/2.
:- meta_predicate list_or_aorb(pred(1),?).

list_or_aorb(T)  := ~list(T).
list_or_aorb(_T) := ~aorb.
```

```
%% This is a property definition
%% This comment appears only in the place where the property itself
%% is documented.
:- doc(long/1,"This is a property, describing a list that is longish.
   The definition is:

   @includedef{long/1}

   ").

%% The comment here will be used to document any predicate which has an
%% assertion which uses the property
:- prop long(L) # "@var{L} is rather long.".

long(L) :-
    length(L,N),
    N>100.

%% Now, a series of assertions:
%%
%% This declares the entry mode of this exported predicate (i.e.,
%% how it is called from outside).
:- entry p/3 : gnd * var * var.

%% This describes all the calls
:- calls p/3 : foo * bar * baz.

:- prop foo/1.
foo(_).

%% This describes the successes (for a given type of calls)
:- success p/3 : int * int * var => int * int * gnd.

%% This describes a global property (for a given type of calls)
:- comp p/3 : int * int * var + not_fails.

:- doc(p/3,"A @bf{general comment} on the predicate." ).
%% Documenting some typical usages of the predicate
:- pred p/3
    : int * int * var
       => int * int * list
    + (iso,not_fails)
    # "This mode is nice.".
:- pred p(Preds,Value,Assoc)
    : var * var * list
       => int * int * list
    + not_fails # "This mode is also nice.".
:- pred p/3
       => list * int * list
    + (not_fails,not_fails)
    # "Just playing around.".
```

```
:- pred q(A)
    : list(A)
       => (list(A),gnd(A))
    + not_fails
    # "Foo".
:- pred q(A)
    # "Not a bad use at all.".

:- pred q/2
    : var * {gnd,int}
       => {gnd,int} * int.
:- pred q/2
    :: int * list
    # "Non-moded types are best used this way.".

q(_).

:- pred p/1 : var => list.

p(_).

:- pred r(A)
    : list(A)
       => (list(int,A),gnd(A))
    + not_fails
    # "This uses parametric types".

:- doc(doinclude,s/1). %% Forces documentation even if not exported
:- pred s(A)
    : list(A)
       => (list(A),gnd(A))
    + not_fails.

s(_).

:- doc(doinclude,[list/2,list/1]). %% Forces (local) documentation even if
                                   %% not exported

:- modedef og(A)
       => gnd(A)
    # "This is a @em{mode} definition: the output is ground.".

:- doc(doinclude,og/2).

:- modedef og(A,T)
    :: T(A)
    => gnd(A)
    #  "This is a @em{parametric mode definition}.".

:- pred t(+A,-B,?C,@D,og(E))
```

```
    :: list * list * int * int * list
    :  long(B)
    => (gnd(C),gnd(A))
    +  not_fails
    #  "This predicate uses @em{modes} extensively.".

t(_, _, _, _, _).

%% Some other miscellaneous assertions:

%% Check is default assertion status anyway...
:- check pred u(+,-,og).
:- check pred u(int,list(mytype),int).

u(_, _, _).

%% ``true'' status is normally compiler output
:- true pred w(+list(mytype)).

mytype(_).

w(_).

:- doc(doinclude,is/2).

:- trust pred is(Num,Expr) : arithexpression(Expr) => num(Num)
   # "Typical way to describe/document an external predicate (e.g.,
      written in C).".

:- doc(doinclude,p/5).
:- pred p(og(int),in,@list(int),-,+A) + steps_lb(1+length(A)).

p(_, _, _, _, _) :- _ is 1.

%% Version information. The ciao.el emacs mode allows automatic maintenance
```

24 Including Editable and Runnable Examples

Author(s): The Ciao Development Team.

This section describes how to include editable and runnable code blocks in `LPdoc` documents, to create interactive *Active Logic Documents* (`ALD`s). These code blocks can either run embedded in the document or be loaded into a playground,

24.1 Runnable Code Blocks

A *runnable* code block is a special documentation code block where the language is marked as `ciao_runnable`, as follows:

```
‘‘‘ciao_runnable

‘‘‘
```

These code fragments are automatically rendered as editable cells that can be run, either in place, embedded in the document, for formats/backends that support this (typically `html`), or loaded into a separate playground. Additional commands can be included within the code which allow marking code areas specially to provide different behaviors and functionality. Below, we provide an overview of the available commands along with a brief explanation for each:

- **Focus**: The `@begin{focus}` and `@end{focus}` directives are used to selectively show certain sections of the code, while hiding the rest (typically module declarations, imports, or auxiliary code, tests, etc.).

  To reload or reset the code in this cell, simply **click on the green tick mark** located in the top left corner of the pane. This functionality enables you to reload the code at any time.

  Example:

  ```
  ‘‘‘ciao_runnable
  :- module(_, [qsort/2], []).
  %! \begin{focus}
  qsort(Data, Out) :-
      qsort_(Data, Out, []).

  qsort_([], R, R).
  qsort_([X|L], R, R0) :-
      partition(L, X, L1, L2),
      qsort_(L2, R1, R0),
      qsort_(L1, R, [X|R1]).
  %! \end{focus}

  partition([],_,[],[]).
  partition([X|L],Y,[X|L1],L2) :-
      X =< Y, !,
      partition(L,Y,L1,L2).
  partition([X|L],Y,L1,[X|L2]) :-
      partition(L,Y,L1,L2).
  ‘‘‘
  ```

  Which produces the following cell:

  ```
  :- module(_, [qsort/2], []).
  %! \begin{focus}
  qsort(Data, Out) :-
      qsort_(Data, Out, []).

  qsort_([], R, R).
  qsort_([X|L], R, R0) :-
      partition(L, X, L1, L2),
      qsort_(L2, R1, R0),
      qsort_(L1, R, [X|R1]).
  %! \end{focus}

  partition([],_,[],[]).
  partition([X|L],Y,[X|L1],L2) :-
  ```

25 Source in markdown for the 'factorial using ISO-Prolog arithmetic' example

```
\title Exercise: factorial using ISO-Prolog arithmetic

Consider again the factorial example, using Peano arithmetic:
```ciao_runnable
:- module(_, _, [assertions,sr/bfall]).
%! \begin{focus}
factorial(0,s(0)).
factorial(s(N),F) :-
    factorial(N,F1),
    times(s(N),F1,F).
%! \end{focus}

nat_num(0).
nat_num(s(X)) :- nat_num(X).

times(0,Y,0) :- nat_num(Y).
times(s(X),Y,Z) :- plus(W,Y,Z), times(X,Y,W).

plus(0,Y,Y) :- nat_num(Y).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

Some facts to note about this version:
  - It is fully reversible!
```ciao_runnable
?- factorial(X,s(s(s(s(s(s(0)))))).
```
  - But also inefficient...
```ciao_runnable
?- factorial(s(s(s(s(0)))),Y).
```

We can also code it using ISO-Prolog arithmetic, i.e., `is/2`:
```ciao
 ... Z is X * Y ...
```
Note that this type of arithmetic has limitations: it only works
in one direction, i.e., `X` and `Y` must be bound to arithmetic terms.

But it provides a (large!) performance gain. Also, meta-logical
tests (see later) allow using it in more modes.

Try to encode the factorial program using `is/2`:
```ciao_runnable
:- module(_, _, [assertions]).

:- test factorial(A, B) : (A = 0) => (B = 1) + (not_fails, is_det).
:- test factorial(A, B) : (A = 1) => (B = 1) + (not_fails, is_det).
:- test factorial(A, B) : (A = 2) => (B = 2) + (not_fails, is_det).
:- test factorial(A, B) : (A = 3) => (B = 6) + (not_fails, is_det).
```
```

```
:- test factorial(A, B) : (A = 4) => (B = 24) + (not_fails, is_det).
:- test factorial(A, B) : (A = 5) => (B = 120) + (not_fails, is_det).
:- test factorial(A, B) : (A = 0, B = 0) + (fails, is_det).
:- test factorial(A, B) : (A = 5, B = 125) + (fails, is_det).
:- test factorial(A, B) : (A = -1) + (fails, is_det).

%! \begin{hint}
% TASK 1 - Rewrite with Prolog arithmetic
factorial(0,s(0)).    % TODO: Replace s(0) by 1
factorial(M,F) :-     % TODO: Make sure that M > 0
    M = s(N),         % TODO: Compute N from M using is/2 (note that N is
    factorial(N,F1),  %       unbound, so you need to compute N from M!)
    times(M,F1,F).    % TODO: Replace times/3 by a call to is/2 (using *)
% When you are done, press the circle ("Run tests") or the arrow
% ("Load into playground").
%! \end{hint}
%! \begin{solution}
factorial(0,1).
factorial(N,F) :-
    N > 0,
    N1 is N-1,
    factorial(N1,F1),
    F is F1*N.
%! \end{solution}
```

Note that wrong goal order can raise an error (e.g., moving the last call to ‘is/2‘ before the call to factorial).

Next: Let’s try using constraints instead!

26 Exercise: factorial using ISO-Prolog arithmetic

Consider again the factorial example, using Peano arithmetic:

```
:- module(_, _, [assertions,sr/bfall]).
%! \begin{focus}
factorial(0,s(0)).
factorial(s(N),F) :-
    factorial(N,F1),
    times(s(N),F1,F).
%! \end{focus}

nat_num(0).
nat_num(s(X)) :- nat_num(X).

times(0,Y,0) :- nat_num(Y).
times(s(X),Y,Z) :- plus(W,Y,Z), times(X,Y,W).

plus(0,Y,Y) :- nat_num(Y).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

Some facts to note about this version:

- It is fully reversible!

```
?- factorial(X,s(s(s(s(s(s(0)))))).
```

- But also inefficient...

```
?- factorial(s(s(s(s(0)))),Y).
```

We can also code it using ISO-Prolog arithmetic, i.e., `is/2`:

```
... Z is X * Y ...
```

Note that this type of arithmetic has limitations: it only works in one direction, i.e., `X` and `Y` must be bound to arithmetic terms.

But it provides a (large!) performance gain. Also, meta-logical tests (see later) allow using it in more modes.

Try to encode the factorial program using `is/2`:

```
:- module(_, _, [assertions]).

:- test factorial(A, B) : (A = 0) => (B = 1) + (not_fails, is_det).
:- test factorial(A, B) : (A = 1) => (B = 1) + (not_fails, is_det).
:- test factorial(A, B) : (A = 2) => (B = 2) + (not_fails, is_det).
:- test factorial(A, B) : (A = 3) => (B = 6) + (not_fails, is_det).
:- test factorial(A, B) : (A = 4) => (B = 24) + (not_fails, is_det).
:- test factorial(A, B) : (A = 5) => (B = 120) + (not_fails, is_det).
:- test factorial(A, B) : (A = 0, B = 0) + (fails, is_det).
:- test factorial(A, B) : (A = 5, B = 125) + (fails, is_det).
:- test factorial(A, B) : (A = -1) + (fails, is_det).

%! \begin{hint}
% TASK 1 - Rewrite with Prolog arithmetic
factorial(0,s(0)).    % TODO: Replace s(0) by 1
factorial(M,F) :-     % TODO: Make sure that M > 0
    M = s(N),         % TODO: Compute N from M using is/2 (note that N is
    factorial(N,F1),  %       unbound, so you need to compute N from M!)
```

```
    times(M,F1,F).    % TODO: Replace times/3 by a call to is/2 (using *)
% When you are done, press the circle ("Run tests") or the arrow
% ("Load into playground").
%! \end{hint}
%! \begin{solution}
factorial(0,1).
factorial(N,F) :-
    N > 0,
    N1 is N-1,
    factorial(N1,F1),
    F is F1*N.
%! \end{solution}
```

Note that wrong goal order can raise an error (e.g., moving the last call to `is/2` before the call to factorial).

Next: Let's try using constraints instead!

References

[**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.

[**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.

[**Bue95**] F. Bueno.
The CIAO Multiparadigm Compiler: A User's Manual.
Technical Report CLIP8/95.0, Facultad de Informática, UPM, June 1995.

[**Bue98**] F. Bueno.
Using Assertions for Static Debugging of CLP: A Manual.
Technical Report CLIP1/98.0, DISCIPL Project/CLIP Group, UPM, June 1998.

[**DEDC96**] P. Deransart, A. Ed-Dbali, and L. Cervoni.
Prolog: The Standard.
Springer-Verlag, 1996.

[**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.

[**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.

[**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.

[**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.

[**JL88**] D. Jacobs and A. Langen.
Compilation of Logic Programs for Restricted And-Parallelism.
In *European Symposium on Programming*, pages 284–297, 1988.

[**JM94**] J. Jaffar and M.J. Maher.
Constraint Logic Programming: A Survey.
Journal of Logic Programming, 19/20:503–581, 1994.

[**Knu84**] D. Knuth.
Literate programming.
Computer Journal, 27:97–111, 1984.

[**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.

[**PBH97**] G. Puebla, F. Bueno, and M. V. Hermenegildo.
An Assertion Language for Debugging of Constraint Logic Programs.
Technical Report CLIP2/97.1, Facultad de Informática, UPM, July 1997.

[**PBH98**] G. Puebla, F. Bueno, and M. V. Hermenegildo.
A Framework for Assertion-based Debugging in Constraint Logic Programming.
In *Proceedings of the JICSLP'98 Workshop on Types for CLP*, pages 3–15, Manchester, UK, June 1998.

[**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.

Library/Module Index

Predicate Index

Property Index

B

C

D

E

F

H

I

L

M

N

P

R

S

T

Regular Type Index

A

B

C

D

F

G

I

L

M

N

O

P

S

T

V

W

Y

Declaration Index

Concept Index

K

L

M

N

O

P

R

S

T

U

V

W

Author Index

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

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y