The Ciao Prolog Playground

Jose F. Morales
Guillermo García Pradales (playground interface)
Manuel Hermenegildo
The Ciao Development Team

Table of Contents

1 Introduction

The Ciao Prolog Playground (`/playground`) offers several main functionalities:

- A very easy way to run and share Prolog code (`/ciao/build/doc/ciao_playground.html/ciao_playground_using.html`), directly from any modern browser.

  The main advantage over other ways of using Ciao is that the playground does not require any installation or interaction with a server since *everything runs within the browser*.
- An easy way to embed runnable code examples (`/ciao/build/doc/ciao_playground.html/ciao_playground_embedding.html`) in tutorials, manuals, slides, exercises, etc., and in general any kind of document.

  These documents can be developed with many tools, such as Google Docs, Jupyter notebooks, Word, Powerpoint, LaTeX, Pages, Keynote, web site generators, etc., etc.

  The examples are stored in the documents themselves and do not need to be uploaded to (or edited in) any server.

  Here is an example of an exercise generated with the LPdoc tool (`/ciao/build/doc/ciao_playground.html/factorial_peano_iso.html`).
- An easy way to create interactive verification tutorials (`/ciao/build/doc/ciao_playground.html/exfilter_documents.html`) for CiaoPP.
- An easy way to create and distribute applications.

  The Playground can be specialized to create standalone web-based applications, with editor and top level, which also do not need installation, running fully within the user's browser. An example is the s(CASP) playground (`/playground/scasp.html`).

The Ciao Playground is based on a web-based editor component and the `wasm` build grade of Ciao (using WebAssembly and compiled with Emscripten).

2 Using the Ciao Prolog Playground

The Ciao Prolog Playground (`/playground`) includes two main areas, which are both editable (note that the arrangement of the areas is configurable):

- The left area (or top area, if the window is narrow) is an editor for the program. Users can add their code into this editor and modify it there. The program can then be loaded into the system's top level, in the right area, where it can be run by entering queries at the prompt.
- The right area (or bottom area, if the window is narrow) hosts the top-level, where most communication with the system takes place. The top-level first shows `Loading Ciao...` while the system is loading its dependencies. When the top-level is completely ready to use, it will show the prompt `?-` . Now programs can be loaded and queries issued.

There is also a preview area for graphical program output.

2.1 Buttons

⟨New⟩ Erases all previous content from the editor.

⟨Open⟩ Lets you upload your own Prolog code from your local filesytem. This action also erases all previous content present in the editor.

⟨Save⟩ Allows downloading the code written in the playground directly to your filesystem.

⟨Load⟩ Loads the code in the editor into the top-level. It compiles it and, in case there are any errors, they are printed in the right (bottom) editor and highlighted in the left (top) one. Once you click this button, you can ask queries about your code in the top-level.

⟨Run tests⟩ Loads the unit test library (`unittest`) into the top-level and runs any unit tests (test assertions) that may appear in the code.

⟨Share!⟩ Is an alternative form of save: it copies into the clipboard a link that allows opening the playground with the current state of the program loaded in the editor area.

⟨Abort query⟩

This button will appear below the top-level when running a query that is taking too long. It terminates the query currently running.

> This button fully terminates the Ciao worker executing the Ciao process, so it may take a some time to reload dependencies after aborting. The code in the editor area will be reloaded into the top-level.

⟨Debug⟩ Debug (or stop debugging) source and enable tracing the control flow of the program for the next queries (see debugger documentation for more information).

⟨Preview documentation⟩

Generate and preview the documentation for the current module (using LPdoc).

⟨Analyze and check assertions⟩

Analyze the program and perform compile-time checking of the assertions (types, modes, determinacy, ...) in the current module (using CiaoPP).

2.2 Key bindings

The playground supports a number of key bindings (including some of the most used in `Emacs`). They are listed below, classified by functionality.

2.2.1 Editing key bindings

These commands are useful to edit and move around the editor areas:

- ⟨Ctrl⟩-⟨A⟩ - move cursor to the beginning of the line.
- ⟨Ctrl⟩-⟨E⟩ - move cursor the end of the line.
- ⟨Ctrl⟩-⟨F⟩ - move cursor forward.
- ⟨Ctrl⟩-⟨B⟩ - move cursor backward.
- ⟨Ctrl⟩-⟨P⟩ - previous line.
- ⟨Ctrl⟩-⟨N⟩ - next line.
- ⟨Ctrl⟩-⟨S⟩ - search forward in the editor.
- ⟨Ctrl⟩-⟨R⟩ - search backward in the editor.
- ⟨Ctrl⟩-⟨D⟩ - delete character to the right.
- ⟨Ctrl⟩-⟨H⟩ - delete character to the left.
- ⟨Ctrl⟩-⟨K⟩ - kill the line after the cursor.
- ⟨Ctrl⟩-⟨M⟩ - insert line below.
- ⟨Ctrl⟩-⟨O⟩ - insert line after the current position.
- ⟨Ctrl⟩-⟨Z⟩ - undo.
- ⟨Ctrl⟩-⟨G⟩ - go to line.
- ⟨Ctrl⟩-⟨X⟩ + ⟨U⟩ - undo.
- ⟨Ctrl⟩-⟨X⟩ + ⟨O⟩ - select the other editor.
- ⟨Ctrl⟩-⟨V⟩ - go to the end of the editor.
- ⟨Ctrl⟩-⟨X⟩ + ⟨Ctrl⟩-⟨P⟩ - select all.
- ⟨Ctrl⟩-⟨X⟩ + ⟨Ctrl⟩-⟨U⟩ - transform selected text to upper case.
- ⟨Ctrl⟩-⟨X⟩ + ⟨Ctrl⟩-⟨L⟩ - transform selected text to lower case.
- ⟨Esc⟩ + ⟨D⟩ - delete the next word.
- ⟨Esc⟩ + ⟨V⟩ - move cursor to the beginning.
- ⟨Esc⟩ + ⟨Backspace⟩ - delete left word.
- ⟨Esc⟩ + ⟨;⟩ - comment/uncomment current line.

2.2.2 Loading key bindings

These commands perform Ciao or filesystem-related actions:

- ⟨Ctrl⟩-⟨X⟩ + ⟨Ctrl⟩-⟨S⟩ - save file (same as button).
- ⟨Ctrl⟩-⟨C⟩ + ⟨L⟩ - load code into top-level (same as button).
- ⟨Ctrl⟩-⟨C⟩ + ⟨U⟩ - run tests in current module (same as button).

2.3 Specific key bindings for the top-level

The top-level area includes some key bindings of its own:

- `Up arrow` - previous query in history.
- `Down arrow` - next query in history.
- ⟨Ctrl⟩-Ⓟ - previous query in history.
- ⟨Ctrl⟩-Ⓝ - next query in history.
- ⟨Ctrl⟩-Ⓒ - abort current query, if there is any running.

⟨Ctrl⟩-Ⓒ fully terminates the Ciao worker executing the Ciao process in the browser, so it may take some time to reload dependencies after aborting. The code present in the left (or top) window will be reloaded to the top-level.

2.4 Current playground limitations

Please be aware of some current limitations of the playground with respect to a full, native installation:

- Binaries and libraries are downloaded into your browser and code is executed locally (no connection required once loaded, no information about your code is gathered). To reduce download times, only some essential libraries are loaded by default.
- Currently this platform is limited to 32-bit binaries, runs around 2-3x slower than native binaries, and only offers partial POSIX features. Some Prolog libraries that depend on 3rd-party binaries (via the foreign interface) may not be currently available.
- The top level currently has some limitations regarding the loading of packages, debugging, portraying answers, and others. We are actively working on all of these and will be adding this functionality shortly.
- Reading input from standard input (`read/1`) is currently not directly supported, due to WebAssembly limitations. We plan also to fix this soon.

Please **ask us** if some useful library or feature is missing; if technically possible, we will add it. However, for intensive use we recommend installing Ciao Prolog natively (`/install.html`), for improved performance and full features.

3 Creating Documents with Editable and Runnable Examples

The Ciao Prolog Playground (`/playground`) provides an *easy way to embed editable and runnable short code snippets* in manuals, tutorials, slides, exercises, etc., and in general any kind of document. The following sections show how to embed *runnable and editable* snippets in `LPdoc` documentation, Google Docs, Jupyter notebooks, Word, Powerpoint, LaTeX, Pages, Keynote, Org mode, web site generators, etc., etc.

3.1 Editable and runnable examples using LPdoc

`LPdoc` offers seamless integration with the playground, allowing runnable and editable code blocks in `LPdoc` generated documentation. You can consult Editable and Runnable Examples (`/ciao/build/doc/lpdoc.html/Runnables.html`) in the LPdoc manual for more information and examples.

3.2 Adding runnable examples to arbitrary documents

Links to the playground that auto-upload examples can be easily included in any document (slides, manual, book, web site, tutorial, article, spreadsheet, etc.) provided the tool used for editing allows including links to URLs. This includes Google Docs, Jupyter notebooks, Word, Powerpoint, LaTeX, Pages, Keynote, HTML, Org mode, web site generators, etc., etc.

The URL to be included in the link can easily be obtained as follows:

1. Paste or upload the example program into the **playground editor**.
2. Click the ⟨Share!⟩ button. This will **copy into the clipboard** a link to an instance of the playground with the program included.
3. Go to the document with the example and **insert the link** in the document as a URL.

The examples are stored in the documents themselves (URI-encoded) and do not need to be uploaded to (or edited in) any server.

Caveats: Be aware that large code snippets may exceed the default maximum sizes for URI length (e.g., configure a larger size like `LimitRequestLine 100000` in `httpd.conf`). For arbitrary size links you can also embed the GitHub link to the full source code `https://ciao-lang.org/playground/#GITHUBURL`, e.g., this link (`https://ciao-lang.org/playground/#https://github.com/ciao-lang/ciao/blob/master/core/exa`

Now, if one clicks on this link, the playground will be opened with the example program loaded.

For example, this link (`https://ciao-lang.org/playground/?code=%25%20Try%3A%20%20%3F-%20is_in_list(X%2C%5B1%2C2%2C3%5D).%0A%0Ais_in_list(X%2C%5BX%7C_%5D).%0Ais_in_list(X%2C%5B_%7CT%5D)%20%3A-%0A%20%20%20%20%20%20is_in_list(X%2CT).`) (*obtained as described above*) opens the playground and loads into its editor the following program:

```
% Try:  ?- is_in_list(X,[1,2,3]).

is_in_list(X,[X|_]).
is_in_list(X,[_|T]) :-
      is_in_list(X,T).
```

We show below some examples of this embedding for different source formats.

3.2.1 LaTeX

This is a simple example in LaTeX:

```
\documentclass{article}

\usepackage{hyperref}

\begin{document}
\noindent
This is a classic example of a Prolog program, which appends two lists:

\begin{verbatim}
app([],X,X).
app([X|Y],Z,[X|W]) :- app(Y,Z,W).
\end{verbatim}

\noindent
Try now
% URL obtained from playground
\href{https://ciao-lang.org/playground/?code=app(%5B%5D%2CX%2CX).%0Aapp(%5BX%7CY%5D%2
{running it!}

\end{document}


%%% Local Variables:
%%% mode: latex
%%% TeX-master: t
%%% End:
```

and this the pdf output generated (`/playground/examples/append_latex_simple.pdf`).

These are a few more examples in LaTeX:

```
\documentclass{article}

\usepackage{cmbright}
\usepackage{graphicx}

\usepackage[ciao]{prologrun}

\begin{document}
\vspace*{-5mm}
\thispagestyle{empty}

\noindent
These are examples of several ways to list (Ciao)Prolog code in a
LaTeX document including a link for loading and running the
example on the \href{https://ciao-lang.org/playground}{Ciao
  playground}. The link is obtained by pasting the program into the
Ciao playground and using the
% \mbox{} \hfill
\includegraphics[scale=.4,trim=0 12 0 0]{ShareLight.png} button to
create the link (containing the program) and copy it into the
```

```
clipboard, from which it can be pasted into the LaTeX
document. This file uses the \texttt{prologrun.sty} style file provided with the play

\begin{enumerate}

\item Using a
  \href{https://ciao-lang.org/playground/?code=app(%5B%5D%2CX%2CX).%0Aapp(%5BX%7CY%5D
    {simple link}\ .
\ \\
This is generated by:
\begin{lstlisting}[language=TeX,basicstyle=\small\ttfamily]
\href{https://ciao-lang.org/playground/...}{simple link}
\end{lstlisting}

\bigskip
\item Using the \texttt{runlink} macro:
  \runlink{https://ciao-lang.org/playground/?code=app(%5B%5D%2CX%2CX).%0Aapp(%5BX%7C
\ \\
This is generated by:
\begin{lstlisting}[language=TeX,basicstyle=\small\ttfamily]
\runlink{https://ciao-lang.org/playground/...}
\end{lstlisting}

\bigskip
\item Using the \texttt{runlink} macro before a standard code area:
\hfill \runlink{https://ciao-lang.org/playground/?code=app(%5B%5D%2CX%2CX).%0Aapp(%5
\begin{prolog}
append([],X,X).
append([X|Y],Z,[X|W]) :-
     append(Y,Z,W).
\end{prolog}
\ \\
This is generated by:
\begin{lstlisting}[language=TeX,basicstyle=\small\ttfamily]
\hfill \runlink{https://ciao-lang.org/playground/...}
\begin{prolog}
append([],X,X).
append([X|Y],Z,[X|W]) :-
     append(Y,Z,W).
\end{prolog}
\end{lstlisting}

\bigskip
\item Declaring the link first and using a \texttt{prologrun} code area:

\codelink{https://ciao-lang.org/playground/?code=app(%5B%5D%2CX%2CX).%0Aapp(%5BX%7CY%

\begin{prologrun}
append([],X,X).
append([X|Y],Z,[X|W]) :-
     append(Y,Z,W).
```

```
\end{prologrun}
\ \\
This is generated by:
\begin{lstlisting}[language=TeX,basicstyle=\small\ttfamily]
\codelink{https://ciao-lang.org/playground/...}
\begin{prologrun}
append([],X,X).
append([X|Y],Z,[X|W]) :-
     append(Y,Z,W).
\end{prologrun}
\end{lstlisting}

% \ \\

% \item This is with a more bespoke macro (\texttt{prologrunalt}), with
%   extra argument to pass the URL, but the URL needs to be escaped:

% \begin{prologrunalt}[] % The [] is needed!
%     {https://ciao-lang.org/playground/?code=app(%5B%5D%2CX%2CX).%0Aapp(%5BX%7CY%5D%
% append([],X,X).
% append([X|Y],Z,[X|W]) :-
%      append(Y,Z,W).
% \end{prologrunalt}

\end{enumerate}

\end{document}

%%% Local Variables:
%%% mode: latex
%%% TeX-master: t
%%% End:
```

and this the pdf output generated (`/playground/examples/append_latex_nicer.pdf`).

3.2.2 Word

And this is a simple example in Word (note: image below not clickable):

This is the word file (`/playground/examples/append_word_simple.docx`), here exported to pdf (`/playground/examples/append_word_simple_cropped.pdf`).

3.2.3 Org

Another example, in Emacs Org mode:

```
#+OPTIONS:   toc:nil num:0

* The append example

This is a classic example of a Prolog program, which appends two lists:

#+BEGIN_SRC ciao
```

```
app([],X,X).
app([X|Y],Z,[X|W]) :- app(Y,Z,W).
#+END_SRC

Try now [[https://ciao-lang.org/playground/?code=app(%5B%5D%2CX%2CX).%0Aapp(%5BX%7CY%
```

And this is the result exported to pdf (`/playground/examples/append_org_simple.pdf`).

4 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!

5 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!

6 Creating Interactive Verification Tutorials

This part describes advanced uses of LPdoc and the Ciao Playground to create and maintain interactive tutorials for analysis and verification using CiaoPP.

These documents require the features provided in the `exfilter` bundle, which extends LPdoc and the playground with functionality to extract, filter, and compare the output of program analyses, both statically and dynamically.

6.1 Non-interactive examples (static content)

Messages and output generated during `Ciao` compilation or `CiaoPP` analysis, for a particular code snippet, can be filtered and included in manuals with the following LPdoc command:

`@`*exfilter*`{Code}{Action, FlagsValues, Filters}`.

In order to enable this functionality it is necessary to include this line in the LPdoc document configuration files (e.g., `SETTINGS.pl`):

```
load_doc_module := exfilter(exfilter_lpdoc).
```

This process happens in two phases: first the document is processed to identify the different (compilation, analysis, verification, etc.) tasks, then the `ciao-exfilter` command needs to be executed to process the tasks and filter the results necessary for the manual. Typically the results are statically committed in the respository, which enables easy detection of regressions.

For instance, consider the `app/3` program:

```
:- module(_,app/3,[assertions]).

:- pred app(Xs, Ys, Zs) : ( list(Xs), list(Ys) ) => list(Zs) .

app([],     Ys, Ys).
app([X|Xs], Ys, [X|Zs]) :-
  app(Xs, Ys, Zs).
```

where the `pred` assertion indicates how `app/3` should be called in this program. We would like to display the part of the analysis output that shows if the assertion has been checked. This fragment can be generated by including in the source file the following code:

```
@exfilter{app.pl}{V,output=on,filter=check_pred}
```

This command is composed of the file that we are looking to filter (`app.pl`), the analysis and the filter options that we want applied (`V`,`output=on`,`filter=check_pred`). The result is:

```
:- checked calls app(Xs,Ys,Zs)
   : ( list(Xs), list(Ys) ).

:- checked success app(Xs,Ys,Zs)
   : ( list(Xs), list(Ys) )
   => list(Zs).
```

6.1.1 CiaoPP/filters options

Usage: `@`*exfilter* `{Code }{Action, FlagsValues, Filters}`.

6.1.1.1 CiaoPP flags and actions

`Action` must be one of the following:

- **A** : Analyzes the code program.
- **O** : Optimizes the code program.
- **V** : Verifies the assertions of the code program.

`FlagsValues` is a list of options `FlagName=FlagValue` where `FlagName` is a valid `CiaoPP` flag name. This list is optional and the flags not included in this list will be assumed to take their default value. Check CiaoPP Reference Manual (`/ciao/build/doc/ciaopp.html/`) for the different available flags.

6.1.1.2 Filters

`Filters` is composed of an option `filter=Filter` and an optional list of supplementary filters. `Filter` must be one of the following:

- **all** : keep all data.
- **tpred** : all `true` (predicate) assertions.
- **tpred_plus** : all `true` assertions including *comp* properties.
- **tpred_regtype** : all `true` assertions and all *regtypes*.
- **regtype** : only all *regtypes* definitions.
- **warnings** : all warning messages
- **error** : all error messages.
- **check_pred** : all `check` assertions, `false` assertions and `checked` assertions.
- **warn_error** : all warning and error messages.
- **test** : all tests.

Include these options with one of the filters for more precise results.

- `name=Pred` : results of a specific predicate, with `Pred` being the predicate.
- `assertion=[Terms]` : assertions that contain a series of terms, where `Terms` is the list of terms to be matched.
- `comments=on` : If we add this option together with the filter `check_pred` then the comments of the assertions will be added as well.
- `absdomain=AD` : `AD` can be types or can be modes. We have to add this option together with filter `tpred`, `tpred_regtype` or `tpred_plus` to obtain the assertions related to the types or modes.

6.1.1.3 Example Usage

The following are very simple examples of use:

- Run analyse on `app.pl` file with default options and obtain the true assertions related to types:

```
@exfilter{app.pl}{A,filter=tpred,absdomain=types}
```

  Output result:

```
:- true pred app(Xs,Ys,Zs)
   : ( list(Xs), list(Ys), term(Zs) )
   => ( list(Xs), list(Ys), list(Zs) ).
```

- Setting a cost domain and extracting the true assertions including comp properties:

```
@exfilter{app.pl}{A,ana_cost=resources,filter=tpred_plus}
```

Output result:

```
:- true pred app(Xs,Ys,Zs)
   : ( list(Xs), list(Ys), term(Zs) )
   => ( list(Xs), list(Ys), list(Zs) )
   + cost(lb,steps,0).

:- true pred app(Xs,Ys,Zs)
   : ( list(Xs), list(Ys), term(Zs) )
   => ( list(Xs), list(Ys), list(Zs) )
   + cost(ub,steps,inf).
```

- Setting a cost domain and extracting the `true` assertions including comp properties, but only the assertion of the upper bound analysis:

```
@exfilter{app.pl}{A,ana_cost=resources,assertion=[ub],filter=tpred_plus}
```

Output result:

```
:- true pred app(Xs,Ys,Zs)
   : ( list(Xs), list(Ys), term(Zs) )
   => ( list(Xs), list(Ys), list(Zs) )
   + cost(ub,steps,inf).
```

6.2 Interactive examples (dynamic content)

The playground already included specific commands for embedding editable and runnable code in place. The `exfilter` bundle extends this functionality with additional steps that turns code snippents into exercises. In particular, the analysis and the filtering method are executed directly from the browser.

For example, in this exercise we want the user to correct the singleton variable written in the base case of predicate `app/3`:

```
:- module(_,app/3,[assertions]).
%! \begin{code}
app([],     Ys, Yss).
app([X|Xs], Ys, [X|Zs]) :-
  app(Xs, Ys, Zs).
%! \end{code}
%! \begin{opts}
solution=errors,message=singleton
%! \end{opts}
%! \begin{solution}
app([],     Ys, Ys).
app([X|Xs], Ys, [X|Zs]) :-
  app(Xs, Ys, Zs).

% In our case, we type Yss instead of Ys.
%! \end{solution}
```

When users submit their answers, the playground will return appropriate feedback showing the error messages related to the singleton variables. We have defined lpdoc-style commands which mark different parts of the program. The playground identifies the different parts and classifies them. For example, the playground will recognize a solution, a set of filters that we

want to apply, and the part of the interactive code that will be shown in the documentation. In this case, the solution will be shown only when the ⟨Show solution⟩ button is clicked. The code part is always visible. Hence, the previous exercise can be generated by including in the source file the following code:

```
```ciao_runnable
:- module(_,app/3,[assertions]).
%! \begin{code}
app([],     Ys, Yss).
app([X|Xs], Ys, [X|Zs]) :-
  app(Xs, Ys, Zs).
%! \end{code}
%! \begin{opts}
solution=errors,message=singleton
%! \end{opts}
%! \begin{solution}
app([],     Ys, Ys).
app([X|Xs], Ys, [X|Zs]) :-
  app(Xs, Ys, Zs).

% In our case, we type Yss instead of Ys.
%! \end{solution}
```
```

6.2.1 Modes of interactive exercises

For the inclusion of exercises in the documentation three distinct modes of exercises have been developed:

- `solution=equal` : Prints out the clauses from the user's answer that do not match with the file that contains the solution. It takes care of trivial differences such as different variable names and different formatting of the code in the files.
- `solution=errors` : Another task is to find bugs in a program and fix them. Users submit their solutions and the filter checks them. If there are none then the program has been corrected successfully. Additionally, a message filter has been created to extract a message or messages by a certain term.
- `solution=verify_assert` : Analysis information allows us to conclude that the program is incorrect or incomplete, i.e., that the program does not satisfy the requirements. The idea is that given the solution of the user, the filter checks if the user's program verifies the assertions (specifications).

6.2.1.1 Example Usage

Example 1. *What is the correct assertion?*

```
%! \begin{code}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{code}
%! \begin{opts}
solution=verify_assert
%! \end{opts}
%! \begin{solution}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => list(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{solution}
```

It can be generated by including in the source file the following code:

````
```ciao_runnable
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{code}
%! \begin{opts}
solution=verify_assert
%! \end{opts}
%! \begin{solution}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => list(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{solution}
```
````

Note that you can also generate interactive examples, for example:

```
%! \begin{code}
```

```
:- module(_,app/3,[assertions]).

:- pred app(Xs, Ys, Zs) : ( list(Xs), list(Ys) ) => list(Zs) .

app([],     Ys, Ys).
app([X|Xs], Ys, [X|Zs]) :-
  app(Xs, Ys, Zs).
%! \end{code}
%! \begin{opts}
A,ana_det=nfdet,filter=tpred_plus
%! \end{opts}
```

This block runs analysis on the program with a determinism domain and displays the `true` assertions including *comp* properties.

7 Source for the 'Checking predicate app/3' example

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

:- doc(filetype, documentation).

:- doc(title, "Checking predicate app/3").

:- doc(module, "

@section{Analyzing}

Let us analyze this implementation of the @pred{app/3} predicate:

```ciao_runnable
%! \\begin{code}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \\end{code}
%! \\begin{opts}
A,filter=all
%! \\end{opts}
```

Automatic analysis can be performed by clicking @key{?} button.

By default, @apl{CiaoPP} analyzes programs with a type domain (@tt{eterms}) and a
sharing/freeness domain (@tt{shfr}). The inferred information is expressed with
(@tt{true}) assertions (for a more
extensive tutorial of the assertion language see @ref{Using assertions for
preprocessing programs}).
@tt{true} represents abstractions of the behaviour of the program inferred by the
analyzer. In this case, it was inferred:

@exfilter{app_assrt_false.pl}{A,filter=tpred}

The first assertion contains types information and encodes @em{predicate @tt{app/3}
called with @var{A} and @var{B} as @tt{lists} and if it succeeds, @var{C} will
be a @tt{list}}.

The second one contains information about how variables are shared. It was
inferred that when @tt{app(A,B,C)} is called arguments @var{A}, @var{B}, and
@var{C} may be shared and if it succeeds they also will be shared, since @var{C}
is the concatenation of @var{A} and @var{B}.

@section{Assertion Checking}
```

```
In the example above, we have also added an assertion with properties that we want
to prove about the execution of the program.

'''ciao
:- pred app(A,B,C) : (list(A), list(B)) => var(C).
'''

This assertion is stating that if the predicate is called with a @var{A} and
@var{B} @tt{list}, if it succeeds @var{C} will be a free variable.
Of course, this assertion does not hold and we get a message saying so:

@exfilter{app_assrt_false.pl}{V,filter=warn_error}

Assertion checking can also be reported within the source code,
we can see that the analyzer does not verify the assertion that we had included.
Run the analysis again (clicking @key{?} button) to see the result.

@begin{cartouche}
@bf{Exercise. }@em{What assertion would we need to add?}
'''ciao_runnable
%! \\begin{code}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \\end{code}
%! \\begin{opts}
solution=verify_assert
%! \\end{opts}
%! \\begin{solution}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => list(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \\end{solution}
'''
@end{cartouche}


@section{Program Optimization}

The following program naively implements a predicate that duplicates the first
element of a list.

'''ciao_runnable
```

```
%! \\begin{code}
:- module(_, [dup_first/2], []).

dup_first([X|Xs], Zs) :-
    app([X], [X|Xs], Zs).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \\end{code}
%! \\begin{opts}
O,filter=all
%! \\end{opts}
'''

@apl{CiaoPP} can automatically optimize sources, see the result by clicking @key{?} l
").
```

8 Checking predicate app/3

8.1 Analyzing

Let us analyze this implementation of the `app/3` predicate:

```
%! \begin{code}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{code}
%! \begin{opts}
A,filter=all
%! \end{opts}
```

Automatic analysis can be performed by clicking ⑦ button.

By default, `CiaoPP` analyzes programs with a type domain (`eterms`) and a sharing/freeness domain (`shfr`). The inferred information is expressed with (`true`) assertions (for a more extensive tutorial of the assertion language see ⟨undefined⟩ [Using assertions for preprocessing programs], page ⟨undefined⟩). `true` represents abstractions of the behaviour of the program inferred by the analyzer. In this case, it was inferred:

```
:- true pred app(A,B,C)
   : mshare([[A],[A,B],[A,B,C],[A,C],[B],[B,C],[C]])
   => mshare([[A,B,C],[A,C],[B,C]]).

:- true pred app(A,B,C)
   : ( list(A), list(B), term(C) )
   => ( list(A), list(B), list(C) ).
```

The first assertion contains types information and encodes *predicate* `app/3` *is called with* `A` *and* `B` *as* `lists` *and if it succeeds,* `C` *will be a* `list`.

The second one contains information about how variables are shared. It was inferred that when `app(A,B,C)` is called arguments `A`, `B`, and `C` may be shared and if it succeeds they also will be shared, since `C` is the concatenation of `A` and `B`.

8.2 Assertion Checking

In the example above, we have also added an assertion with properties that we want to prove about the execution of the program.

```
:- pred app(A,B,C) : (list(A), list(B)) => var(C).
```

This assertion is stating that if the predicate is called with a `A` and `B` `list`, if it succeeds `C` will be a free variable. Of course, this assertion does not hold and we get a message saying so:

```
ERROR (ctchecks_pred_messages): (lns 3-3) False assertion:
:- check success app(A,B,C)
   : ( list(A), list(B) )
   => var(C).
because the success field is incompatible with inferred success:
[eterms] basic_props:list(A),basic_props:list(B),basic_props:list(C)
```

```
ERROR (auto_interface): Errors detected. Further preprocessing aborted.
```

Assertion checking can also be reported within the source code, we can see that the analyzer does not verify the assertion that we had included. Run the analysis again (clicking ⟨?⟩ button) to see the result.

Exercise. *What assertion would we need to add?*

```
%! \begin{code}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => var(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{code}
%! \begin{opts}
solution=verify_assert
%! \end{opts}
%! \begin{solution}
:- module(_, [app/3], [assertions]).

:- pred app(A,B,C) : (list(A), list(B)) => list(C).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{solution}
```

8.3 Program Optimization

The following program naively implements a predicate that duplicates the first element of a list.

```
%! \begin{code}
:- module(_, [dup_first/2], []).

dup_first([X|Xs], Zs) :-
    app([X], [X|Xs], Zs).

app([],Y,Y).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
%! \end{code}
%! \begin{opts}
O,filter=all
%! \end{opts}
```

`CiaoPP` can automatically optimize sources, see the result by clicking ⟨?⟩ button.

References

(this section is empty)

Library/Module Index

A

C

E

F

Predicate Index

(Index is nonexistent)

Property Index

(Index is nonexistent)

Regular Type Index

(Index is nonexistent)

Declaration Index

(Index is nonexistent)

Concept Index

Author Index

(Index is nonexistent)

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

C

E

F

I

L

P

R

U

V