The Ciao Preprocessor

A Program Analysis, Verification, Debugging, and Optimization Tool
REFERENCE MANUAL
The Ciao Documentation Series
`http://ciao-lang.org/`
Generated/Printed on: 27 March 2013
Technical Report CLIP 1/06 (first version 8/95).

Edited by:
Francisco Bueno
Manuel Hermenegildo
Pedro López
Germán Puebla

The Computational logic, Languages, Implementation, and Parallelism (CLIP) Lab
`http://www.cliplab.org/`
`webmaster@clip.dia.fi.upm.es`
School of CS, T. U. of Madrid (UPM)
IMDEA Software Institute

Table of Contents

Summary

CiaoPP is the abstract interpretation-based preprocessor of the Ciao multi-paradigm program development environment. CiaoPP can perform a number of program debugging, analysis, and source-to-source transformation tasks on (Ciao) Prolog programs. These tasks include:

- Inference of properties of the predicates and literals of the program, including types, modes and other variable instantiation properties, non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc.
- Certain kinds of static debugging and verification, finding errors before running the program. This includes checking how programs call system library predicates and also checking the assertions present in the program or in other modules used by the program. Such assertions represent essentially partial specifications of the program.
- Several kinds of source to source program transformations such as program specialization, slicing, partial evaluation of a program, program parallelization (taking granularity control into account), inclusion of run-time tests for assertions which cannot be checked completely at compile-time, etc.
- The abstract model of the program inferred by the analyzers is used in the system to certify that an untrusted mobile code is safe w.r.t. the given policy (i.e., an abstraction-carrying code approach to mobile code safety).

The information generated by analysis, the assertions in the system specifications are all written in the same assertion language, which is in turn also used by the Ciao system documentation generator, `lpdoc`.

CiaoPP is distributed under the GNU general public license.

1 Introduction

CiaoPP is the abstract interpretation-based preprocessor of the Ciao multi-paradigm program development environment. CiaoPP can perform a number of program debugging, analysis, and source-to-source transformation tasks on (Ciao) Prolog programs. These tasks include:

- Inference of properties of the predicates and literals of the program, including types, modes and other variable instantiation properties, non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc.
- Certain kinds of static debugging and verification, finding errors before running the program. This includes checking how programs call system library predicates and also checking the assertions present in the program or in other modules used by the program. Such assertions represent essentially partial specifications of the program.
- Several kinds of source to source program transformations such as program specialization, slicing, partial evaluation of a program, program parallelization (taking granularity control into account), inclusion of run-time tests for assertions which cannot be checked completely at compile-time, etc.
- The abstract model of the program inferred by the analyzers is used in the system to certify that an untrusted mobile code is safe w.r.t. the given policy (i.e., an abstraction-carrying code approach to mobile code safety).

The information generated by analysis, the assertions in the system specifications are all written in the same assertion language, which is in turn also used by the Ciao system documentation generator, `lpdoc`.

CiaoPP is distributed under the GNU general public license.

1.1 How to use this manual

This is a reference manual. You can use it to look up in it descriptions for the commands, flags, and options that can be used with CiaoPP. The Predicate/Method Definition Index may help you in locating commands. The Regular Type Definition Index may help in locating the definitions of the types associated to the arguments of commands. The Concept Definition Index may help in locating the part of the manual where a particular feature of CiaoPP is described. The Global Index includes all of the above plus references to pages where the command, type, or concept is used (not necessarily defined).

This chapter gives a brief overview of CiaoPP and its capabilities. It assumes some familiarity with the techniques that implement such functionalities. However, references are included to technical papers that explain in detail such techniques. An overview of the functionalities available is given in [BLGPH06] in the form of a tutorial on CiaoPP.

1.2 Note

We are in the process of merging all CiaoPP functionality into the 1.2 version. In the meantime, the current distribution is marked as alpha and you may find that some functionality documented in this manual is not available or not working properly. Please bear with us in the meantime. Sorry for any inconvenience.

1.3 Installation (ciaopp)

Currently there are two Ciao distributions, one which includes CiaoPP, and another one which does not. For installing the Ciao distributions which include CiaoPP, it is sufficient to follow the instructions enclosed in the Ciao distribution itself. This describes the installation

procedure for the Ciao Development Environment, including libraries and manuals, from a *source* distribution. This applies primarily to Unix-type systems (Linux, Mac OS X, Solaris, SunOS, etc.), and with some limitations to Windows (using the Cygnus Win32 development libraries).

It is recommended that you read the `INSTALLATION` file that comes with each component of CiaoDE. However, in many cases it suffices to follow this summary:

1.4 Software Requirements (ciaode)

For users of Linux distributions, you should install some software packages required by Ciao that do not come installed by default. Using the corresponding automatic software management tool, those are:

```
Debian/Ubuntu:
sudo apt-get install emacs build-essential \
    texi2html texlive texinfo imagemagick
Fedora:
        su
yum install gcc kernel-headers kernel-devel emacs texi2html \
texinfo ImageMagick
```

Some advanced libraries and components, like the cost analysis, require an additional set of software packages:

```
Debian/Ubuntu:
sudo apt-get install libgsl0-dev libgsl0ldbl ant ant-optional \
            sun-java6-jdk g++
sudo update-java-alternatives --set java-6-sun
```

Fedora:

```
su
yum install gsl gsl-devel ant gnu-g++
```

Debian/Ubuntu users for 64-bit systems would also need libraries for 32-bit compatibility:

```
sudo apt-get install gcc-multilib libc6-i386 \
     libc6-dev-i386 ia32-libs
# Optionally, for the Parma Polyhedra Library
sudo apt-get install g++-multilib
```

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

If you are a Ciao developer, it is highly recommended to install Subversion to access the latest source code in our repositories:

```
Debian/Ubuntu:
sudo apt-get install subversion
Fedora:
su
yum install subversion
```

Users of other Linux variants or operating systems should use similar tools to add the required software packages. In Windows, doing a full installation of CygWin ensures that you have all the required packages.

Note that the GNU implementation of the make Un*x command is (still) internally used during the installation process. It is available in many systems (including all Linux systems and Mac OS X) simply as `make`. If any of the installation steps stop right away with `make` error messages, you probably need to install `gmake`.

1.5 Obtaining the Sources (ciaode)

1. If you have obtained your copy of Ciao from a compressed source package, uncompress and unpackage it (using `gunzip` or `bzip2` and `tar -xpf`). This will put everything in a new directory whose name reflects the Ciao version.
2. Other method to get the sources is from the subversion repository (available for ciao developers).

1.6 Quick Installation from Source (ciaode)

The main command to build and install CiaoDE, located at the root of the source tree, is called `ciaosetup`. It provides useful commands to do quick installations from the sources with just one line:

1. System-wide installation (e.g. as administrator or `root` user):

   ```
   ./ciaosetup system-install
   ```

2. User-local installation (that will be accessible just for your user)

   ```
   ./ciaosetup user-install
   ```

   If you need a more advanced control of configuration read the following section.

1.7 Custom Installations (ciaode)

Advanced uses of CiaoDE would require the customization of the default build and installation options. In that case, the installation process usually follows the following steps:

1. From the directory where the sources are stored, run:

   ```
   ./ciaosetup configure
   ```

   It will perform a default configuration, where the system will be configured to be installed as the system administrator (`root`) in a standard directory available for all users in the machine (e.g., `/usr/local`).

   The additional options `--instype=local` will prepare CiaoDE to run from the sources directly, and configured in the user's home directory (recommended for CiaoDE developers).

   In case you want to install elsewhere, or change any of the installation options, you can use a customized configuration procedure. The configure command accepts several options. You can see a brief description of them with:

   ```
   ./ciaosetup configure --help
   ```

   Use the `--menu` option to select configuration options interactively. You must follow the instructions that appear on it. When asked for the configuration level, if you are happy with the default options, select the first option and no questions will be made. If you need a higher level of customization, select the last option.

2. Once the configuration process has finished, run:

   ```
   ./ciaosetup build
   ```

   This will build executables and compile libraries.

3. If you have obtained the CiaoDE source from the SVN repository, you need to generate the documentation. This can be done using:

   ```
   ./ciaosetup docs
   ```

4. After the compilation completes successfully, run:

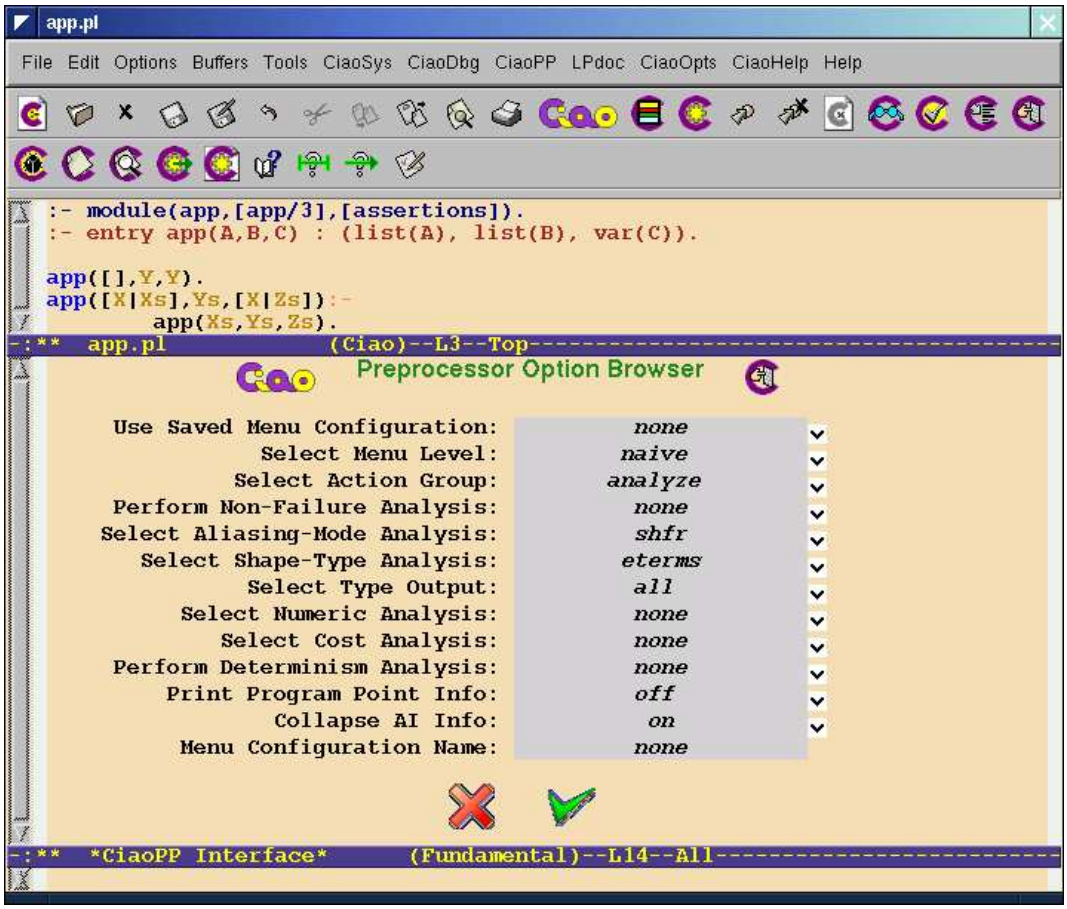
   ```
   ./ciaosetup install
   ```

   This will install everything in the specified directories.

If you want to see the other available commands, run `./ciaosetup help`.

1.8 Getting started

A CiaoPP session consists in the preprocessing of a file. The session is governed by a menu, where you can choose the kind of preprocessing you want to be done to your file among several analyses and program transformations available. Clicking on the icon in the buffer containing the file to be preprocessed displays the menu, which will look (depending on the options available in the current CiaoPP version) something like the "Preprocessor Option Browser" shown in the following figure:



Except for the first and last lines, which refer to loading or saving a menu configuration (a predetermined set of selected values for the different menu options), each line corresponds to an option you can select, each having several possible values. You can select either analysis (`analyze`) or assertion checking (`check_assertions`) or certificate checking (`check_certificate`) or program optimization (`optimize`), and you can later combine the four kinds of preprocessing. The relevant options for the `action group` selected are then shown, together with the relevant flags. A description of the values for each option will be given as it is used in the corresponding section of this manual.

CiaoPP can help you to analyze your program, in order to infer properties of the predicates and literals in your program (which might be useful in the subsequent steps during the same session). You can use Cost Analysis to infer both lower and upper bounds on the computational time cost and sizes of terms of procedures in a program. Mode Analyses obtain at compile-time accurate variable groundness and sharing information and other variable instantiation properties. Type Analysis infers regular types. Regular types are explained in detail in Chapter 8 [Declaring regular types], page 63. Non-failure and Determinacy Analyses detect procedures and goals that can be guaranteed to not fail and/or to be deterministic.

CiaoPP also can help to optimize your program (by means of source-to-source *program transformations*), using program specialization, partial evaluation, program parallelization and granularity control, and other program transformations. Specialization can help to simplify your program w.r.t. the analysis information (eliminating dead code, predicates that are guaranteed to either succeed or fail, etc.), specialize it and then simplify it, or just specialize it, i.e., to unfold all versions of the predicates in your program. CiaoPP can also perform automatic parallelization of your source program during precompilation using several *annotation* algorithms, and granularity control on parallel programs, transforming the program in order to perform run–time granularity control, i.e., deciding parallel or sequential execution of goals depending on the estimated amount of work under them (estimated by cost analysis).

CiaoPP also helps in *debugging* your programs. It makes possible to perform *static debugging*, i.e., finding errors at compile-time, before running the program, and also dynamic debugging, in the sense of including *run-time tests* that will perform the checking for errors at run-time. Static debugging is performed by *assertion checking*. This includes checking the ways in which programs call the system library predicates and also checking the assertions present in the program or in other modules used by the program. Such assertions essentially represent partial *specifications* of the program. For dynamic checking, CiaoPP will include run-time tests for the parts of assertions which cannot be checked completely at compile-time.

Chapter 5 [Using assertions for preprocessing programs], page 37, gives an overview on the use of the assertion language in CiaoPP. In that chapter and the following ones, several existing properties that can be used in assertions are described. Programmers can also define their own properties (see the abovementioned chapters).

1.9 CiaoPP interfaces

There are three main levels of interaction with CiaoPP. There is a graphical menu interface, based on the `emacs` editor, that allows the selection of configuration options and the use of the different features of CiaoPP. If `emacs` is not available, this menu interface can be used as a text-based menu interface. There are several supplementary predicates for assisting the user and providing a kind of scripting language (based on the Ciao language). This interface is described in Chapter 2 [The CiaoPP user menu interface], page 11.

The second level of interaction with CiaoPP is the low-level interface, detailed in Chapter 3 [The CiaoPP low-level programming interface], page 19. This interface is intended for advanced developers, and contains the primitives used by the abovementioned menu-based interface for implementing the main features of the system.

And finally, the command-line interface allows the use of CiaoPP without direct interaction of the user. With this feature, the CiaoPP system can be integrated into other systems (as for example interactive web sites) by means of batch commands. It is described in Chapter 4 [The CiaoPP command-line interface], page 33.

PART I - Using CiaoPP

Author(s): The CLIP Group.

2 The CiaoPP user menu interface

Author(s): David Trallero Mena.

This module defines a simplified user-level interface for CiaoPP. It complements the more expert-oriented interface defined in Chapter 3 [The CiaoPP low-level programming interface], page 19. This is also the interface called by the shortcuts available in menus and toolbars in the emacs mode.

The idea of this interface is to make it easy to perform some fundamental, prepackaged tasks, such as checking assertions in programs (i.e., types, modes, determinacy, non-failure, cost, etc.), performing optimizations such as specialization and parallelization, and performing several types of analysis of the program. The results can be observed as new or transformed assertions and predicates in a new version of the program.

In order to use CiaoPP, the user must provide two kinds of information: first, a number of preprocessing options must be set if necessary in order to configure the system; and then, the action that has to be done must be selected (analysis, assertion checking, optimization). Those options are controlled by a set of so-called flags. By default, all flags are initialized to the appropriate values in most of the cases. If the value of any of the flags has to be changed by the user, the flag must be changed before performing the corresponding action. There are two ways to change the flag values. The most usual way consists in calling `customize_and_preprocess/1` from the CiaoPP top-level shell with the file name as argument. In the `emacs` environment this can be done most easily by clicking on the options button in the toolbar or in the CiaoPP menus. It will prompt (with help) for the value of the different options and flags.

The second way to change flag values consist in executing in the CiaoPP top-level shell a number of calls to `set_menu_flag/3` with the right values, and then calling one of the following predicates:

- `auto_check_assert/1` with the file name as argument to **check a program**.
- `auto_optimize/1` with the file name as argument to **optimize (transform) a program**.
- `auto_analyze/1` with the file name as argument to **analyze a program**.

In the `emacs` environement these actions can be performed most easily by clicking on the corresponding button in the toolbar or in the CiaoPP menus.

The customization menus can be made to show more or less detail depending on the level of expertise of the user. This can be configured in the customization menu itself.

2.1 Usage and interface (auto_interface)

- **Library usage:**
  `:- use_module(library(auto_interface)).`
- **Exports:**
  - *Predicates:*
    `auto_analyze/1`, `auto_optimize/1`, `auto_check_assert/1`, `auto_analyze/2`, `auto_optimize/2`, `auto_check_assert/2`, `customize/0`, `customize/1`, `customize_and_preprocess/0`, `customize_and_preprocess/1`, `customize_but_dont_save/1`, `again/0`, `clean_aux_files/1`, `select_modules/1`, `customize_java/1`, `customize_and_preprocess_java/1`.
  - *Multifiles:*
    `hook_menu_flag_values/3`, `hook_menu_check_flag_value/3`, `hook_menu_flag_help/3`, `hook_menu_default_option/3`.
- **Imports:**
  - *Application modules:*
    `ciaopp(driver)`, `ciaopp(printer)`, `ciaopp(preprocess_flags)`, `ciaopp(p_unit(p_dump))`, `ciaopp(plai(fixpo_ops))`, `plai(acc_ops)`, `auto_interface(optim_comp)`, `plai(intermod)`, `infer(infer_db)`, `program(assrt_db)`, `program(p_unit)`, `program(itf_db)`, `program(aux_filenames)`, `infer(infer_dom)`.
  - *System library modules:*
    `menu/menu_generator`, `menu/menu_rt`, `lists`, `aggregates`, `prolog_sys`, `system`, `messages`, `prompt`, `filenames`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`, `assertions`, `api(api_menu)`, `menu/menu`, `argnames`, `fsyntax`.

2.2 Documentation on exports (auto_interface)

auto_analyze/1: PREDICATE

Usage: `auto_analyze(F)`

Analyze the module `F` with the current analysis options (use `customize(analyze)` to change these options).

auto_optimize/1: PREDICATE

Usage: `auto_optimize(F)`

Optimize file `F` with the current options (use `customize(optimize)` to change these options).

auto_check_assert/1: PREDICATE

Usage: `auto_check_assert(F)`

Check the assertions in file `F`, with the current options, giving errors if assertions are violated (use `customize(check_assertions)` to change these options).

auto_analyze/2: PREDICATE

Usage: `auto_analyze(F,OFile)`

Same as `auto_analyze/1` but the output file will be `OFile`.

auto_optimize/2: PREDICATE

Usage: `auto_optimize(F,OFile)`

Same as `auto_optimize/1` but the output file will be `OFile`.

auto_check_assert/2: PREDICATE

Usage: `auto_check_assert(F,OFile)`

Same as `auto_check_assrt/1` but the output file will be `OFile`.

customize/0: PREDICATE

Usage:

Enter an interactive menu to select the preprocessing action (analysis / assertion checking / transformation / optimization / ...) to be performed by deafult and the different options (i.e., setting the preprocessor flags).

customize/1: PREDICATE

Usage: `customize(X)`

Customize is used for changing the values of a set of flags. These flags are grouped into *analyze*, *check assertions* and *optimize*. X should take the values: `analyze`, `check_assertions` or `optimize`.

customize_and_preprocess/0: PREDICATE

Usage:

Select options using `customize/0`, and then call `auto_analyze/1`, `auto_optimize/1`, or `auto_check_assert/1` (as determined by the selected options) on the default file. If no default file is defined, prompt for the name of to be processed, which becomes from now on the default file.

customize_and_preprocess/1: PREDICATE

Usage: `customize_and_preprocess(File)`

Select options using `customize/0`, and then call `auto_analyze/1`, `auto_optimize/1`, or `auto_check_assert/1` (as determined by the selected options) with `File` as argument. `File` is from now on the default file.

customize_but_dont_save/1: PREDICATE

Usage: `customize_but_dont_save(Option)`

Same as customize(`Option`), but menu flags will not be modified.

again/0: PREDICATE

Usage:

Performs the last actions done by `customize_and_preprocess/1`, on the last file previously analyzed, checked, or optimized

clean_aux_files/1: PREDICATE

Usage: `clean_aux_files(File)`

Deletes any auxiliary file regarding `File` or its related files (e.g., imported modules in a modular program).

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

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

select_modules/1: PREDICATE

Usage:

Launch a menu to select module dependencies.

customize_java/1: PREDICATE

Usage: `customize_java(X)`

Customize is used for change the values of a set of flags in case of java analysis. So far, the value of `X` is only `'all'`.

customize_and_preprocess_java/1: PREDICATE

Usage: `customize_and_preprocess_java(File)`

It is like doing `customize_java(all)`, and then calling with `File` as argument.

get_menu_configs/1: PREDICATE

Usage: `get_menu_configs(X)`

Returns a list of atoms in `X` with the name of stored configurations.

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

  `X` is a free variable. (`var/1`)

- *The following properties should hold upon exit:*

  `X` is a list of `atom`s. (`list/2`)

save_menu_config/1: PREDICATE

Usage: `save_menu_config(Name)`

Save the current flags configuration under the `Name` key.

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

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

remove_menu_config/1: PREDICATE

Usage: `remove_menu_config(Name)`

Remove the configuration stored with the `Name` key (the same provided in `save_menu_config/1`).

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

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

restore_menu_config/1: PREDICATE

Usage: `restore_menu_config(Name)`

Restore the configuration saved with the `Name` key (the same provided in `save_menu_config/1`).

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

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

show_menu_configs/0: PREDICATE

Usage:

Show all stored configurations.

show_menu_config/1: PREDICATE

Usage: `show_menu_config(C)`

Show specific configuration values pointed by `C` key (the same provided in `save_menu_config/1`).

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

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

get_menu_flag/3: (UNDOC_REEXPORT)

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

get_menu_flag/3: (UNDOC_REEXPORT)

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

set_menu_flag/3: (UNDOC_REEXPORT)

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

set_menu_flag/3: (UNDOC_REEXPORT)
Imported from `menu_generator` (see the corresponding documentation for details).

menu_branch/4: (UNDOC_REEXPORT)
Imported from `menu_rt` (see the corresponding documentation for details).

menu_branch/3: (UNDOC_REEXPORT)
Imported from `menu_rt` (see the corresponding documentation for details).

true/2: (UNDOC_REEXPORT)
Imported from `menu_rt` (see the corresponding documentation for details).

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

functor1/2: (UNDOC_REEXPORT)
Imported from `menu_rt` (see the corresponding documentation for details).

2.3 Documentation on multifiles (`auto_interface`)

hook_menu_flag_values/3: PREDICATE
Usage: `hook_menu_flag_values(Menu,Flag,Values)`
Menu hook that determines the possible `Values` that a `Flag` can have in menu `Menu`.
The predicate is *multifile*.

hook_menu_check_flag_value/3: PREDICATE
Usage: `hook_menu_check_flag_value(Menu,Flag,Value)`
Menu hook that checks if `Value` is a correct option for `Flag` in menu `Menu`.
The predicate is *multifile*.

hook_menu_flag_help/3: PREDICATE
Usage: `hook_menu_flag_help(Menu,Flag,Values)`
Menu hook that determines the `Help` text for a `Flag` in menu `Menu`.
The predicate is *multifile*.

hook_menu_default_option/3: PREDICATE
Usage: `hook_menu_default_option(Menu,Flag,DOpt)`
Menu hook that determines the default option `DOpt` for the `Flag` in menu `Menu`.
The predicate is *multifile*.

2.4 Known bugs and planned improvements (`auto_interface`)

- 1 commented out the question for error file since we are generating it in any case (not yet implemented)
- 2 when auto_cthecks has the value 'on' (instead of 'auto'), the result of compile_time checking is not fully handled yet
- needed_to_prove/3 is a weird predicate, it must be more easy to read. –EMM.

3 The CiaoPP low-level programming interface

Author(s): The CLIP Group.

This module includes low-level primitives for interacting with CiaoPP. The exported predicates of this module are intended for developers only.

3.1 Usage and interface (ciaopp)

- **Library usage:**
  `:- use_module(library(ciaopp)).`
- **Exports:**
  - *Predicates:*
    `help/0`.
- **Imports:**
  - *Application modules:*
    `ciaopp(driver)`, `ciaopp(resources(resources_register))`, `ciaopp(infercost(infercost_register))`, `ciaopp(preprocess_flags)`, `ciaopp(printer)`, `auto_interface(auto_interface)`, `auto_interface(auto_help)`, `typeslib(typeslib)`, `program(p_asr)`.
  - *System library modules:*
    `messages`, `system`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`, `condcomp`, `assertions`, `ciaopp_options`.

3.2 Documentation on exports (ciaopp)

current_pp_flag/2: PREDICATE

Usage: `current_pp_flag(Name,Value)`

Preprocess flag `Name` has the value `Value`.

- *The following properties should hold at call time:*
  `Name` is a valid preprocessor flag. (`pp_flag/1`)
- *The following properties should hold upon exit:*
  `Value` is a valid value for preprocessor flag `Name`. (`valid_flag_value/2`)

set_pp_flag/2: PREDICATE

Usage: `set_pp_flag(Name,Value)`

Sets `Value` for preprocessor flag `Name`.

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

`Name` is a valid preprocessor flag. (`pp_flag/1`)

`Value` is a valid value for preprocessor flag `Name`. (`valid_flag_value/2`)

push_pp_flag/2: PREDICATE

(True) Usage: `push_pp_flag(Flag,Value)`

Sets `Value` for preprocessor flag `Flag`, storing the current value to restore it with `pop_pp_flag/1`.

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

`Flag` is a valid preprocessor flag. (`pp_flag/1`)

`Value` is a valid value for preprocessor flag `Flag`. (`valid_flag_value/2`)

pop_pp_flag/1: PREDICATE

(True) Usage: `pop_pp_flag(Flag)`

Restores the value of the preprocessor flag `Flag` previous to the last non-canceled `push_pp_flag/2` on it.

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

`Flag` is a valid preprocessor flag. (`pp_flag/1`)

pp_flag/1: PREDICATE

Valid flags:

- for the output:
  - `analysis_info` (`off, on`) Whether to output the results of analysis.
  - `point_info` (`off,on`) Whether to output analysis information for program points within clauses.
  - `collapse_ai_vers` (`off, on`) to output all the versions of call/success patterns inferred by analysis or just one version (summing-up all of them).
  - `type_output` (`defined, all`) to output the types inferred for predicates in terms only of types defined by the user or including types inferred anew.
- for analysis:
  - `fixpoint` (`plai, dd, di, check_di, check_di2, check_di3, check_di4`) The kind of fixpoint computation used.
  - `multi_success` (`off, on`) Whether to allow success multivariance.
  - `widen` (`off, on`) Whether to perform widening.
  - `intermod` (`off, on, auto`) The policy for inter-modular analysis.
  - `success_policy` (`best, first, all, top, botfirst, botbest, botall, bottom`) The policy for obtaining success information for imported predicates during inter-modular analysis.
  - `entry_policy` (`all, top_level, force, force_assrt`) The policy for obtaining entry call patterns for exported predicates during inter-modular analysis.
  - `process_libraries` (`on, off, no_engine`) Whether to perform the analysis of Ciao system libraries when a modular user program is analyzed.

- `initial_guess` (`botfirst, botbest, botall, bottom`) The policy for obtaining initial guess when computing the analysis of a predicate from the current module.
- `use_check_assrt` (`off, on`) Whether to use check assertions for imported predicates as if they were trust assertions.
- `depth` (a non-negative integer) The maximum depth of abstractions in analyses based on term depth.
- `type_eval` (`on, off`) Whether to attempt concrete evaluation of types being inferred.
- `type_precision` (`defined, all`) to use during type analysis only types defined by the user or also types inferred anew.

- for partial evaluation:
  - `global_control` (`off, id, inst, hom_emb`) The abstraction function to use to control the creation of new patterns to analyze as a result of unfolding.
  - `comp_rule` (`leftmost, safe_jb, bind_ins_jb, no_sideff_jb, jump_builtin, eval_builtin, local_emb`) The computation rule for the selection of atoms in a goal.
  - `local_control` (`off, orig, inst, det, det_la, depth, first_sol, first_sol_d, all_sol, hom_emb, hom_emb_anc, hom_emb_as, df_hom_emb_as, df_tree_hom_emb, df_hom_emb`) The unfolding rule to use during partial evaluation.
  - `unf_depth` (a non-negative integer) The depth limit for unfolding.
  - `rem_use_cls` (`off, pre, post, both`) Whether to remove useless clauses.
  - `abs_spec_defs` (`off, rem, exec, all`) Whether to exploit abstract substitutions while obtaining specialized definitions on unfolding.
  - `filter_nums` (`off, on`) Whether to filter away numbers in partial evaluation.
  - `exec_unif` (`off, on`) Whether to execute unifications during specialization time or not.
  - `pres_inf_fail` (`off, on`) Whether infinite failure should be preserved in the specialized program.
  - `part_concrete` (`off, mono, multi`) The kind of partial concretization to be performed.
- for parallelization and granularity control:
  - `granularity_threshold` (a non-negative integer) The threshold on computational cost at which parallel execution pays off.

flag_value/1: REGTYPE

Usage: `flag_value(V)`

`V` is a value for a flag.

valid_flag_value/2: PROPERTY

Usage: `valid_flag_value(Name,Value)`

`Value` is a valid value for preprocessor flag `Name`.

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

`Name` is a valid preprocessor flag. (`pp_flag/1`)

`Value` is a value for a flag. (`flag_value/1`)

ctcheck_sum/1: (UNDOC_REEXPORT)

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

transform/1: PREDICATE

Usage 1: `transform(Trans)`

Returns on backtracking all available program transformation identifiers.

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

  `Trans` is a free variable. (`var/1`)

- *The following properties should hold upon exit:*

  `Trans` is a valid transformation identifier. (`transformation/1`)

Usage 2: `transform(Trans)`

Performs transformation `Trans` on the current module.

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

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

  `Trans` is a valid transformation identifier. (`transformation/1`)

module/1: PREDICATE

Usage 1: `module(FileName)`

Reads the code of `FileName` and its preprocessing unit, and sets it as the current module.

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

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

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

Usage 2: `module(FileNameList)`

Reads the code of the list of file names `FileNameList` (and their preprocessing units), and sets them as the current modules.

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

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

  `FileNameList` is a list of `atm`s. (`list/2`)

acheck_summary/1: (UNDOC_REEXPORT)

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

acheck/0: PREDICATE

Usage:

Checks assertions w.r.t. analysis information.

analyze/1: PREDICATE

Usage 1: `analyze(Analysis)`

Returns on backtracking all available analyses.

– *The following properties should hold at call time:*
`Analysis` is a free variable. (`var/1`)
– *The following properties should hold upon exit:*
`Analysis` is a valid analysis identifier. (`analysis/1`)

Usage 2: `analyze(Analysis)`

Analyzes the current module with `Analysis`.

– *The following properties should hold at call time:*
`Analysis` is currently a term which is not a free variable. (`nonvar/1`)
`Analysis` is a valid analysis identifier. (`analysis/1`)
– *The following properties should hold globally:*
All the calls of the form `analyze(Analysis)` do not fail. (`not_fails/1`)
A call to `analyze(Analysis)` does not create choicepoints. (`no_choicepoints/1`)

menu_branch/4: (UNDOC_REEXPORT)

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

menu_branch/3: (UNDOC_REEXPORT)

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

true/2: (UNDOC_REEXPORT)

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

true/1: (UNDOC_REEXPORT)

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

functor1/2: (UNDOC_REEXPORT)

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

output/2: (UNDOC_REEXPORT)

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

output/1: PREDICATE

Usage: `output(Output)`

Outputs the current module preprocessing state to a file `Output`.

– *The following properties should hold at call time:*
`Output` is currently a term which is not a free variable. (`nonvar/1`)

output/0: PREDICATE

Usage:

Outputs the current Module preprocessing state to a file named Module_opt.pl, where Module is the current module.

menu_branch/4: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

menu_branch/3: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

true/2: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

true/1: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

functor1/2: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

customize_and_preprocess_java/1: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

customize_java/1: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

select_modules/1: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

clean_aux_files/1: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

customize_and_preprocess/0: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

customize/0: (UNDOC_REEXPORT)

Imported from auto_interface (see the corresponding documentation for details).

help/0: PREDICATE

3.3 Documentation on internals (ciaopp)

analysis/1: PROPERTY

Analyses can be integrated in CiaoPP in an ad-hoc way (see the Internals manual), in which the CiaoPP menu would not be aware of them. The current analyses supported in the menu are:

- for groundness and sharing:
  - `gr` tracks groundness in a very simple way.
  - `def` tracks groundness dependencies, which improves the accuracy in inferring groundness.
  - `share` tracks sharing among (sets of) variables [MH92], which gives a very accurate groundness inference, plus information on dependencies caused by unification.
  - `son` tracks sharing among pairs of variables, plus variables which are linear (see [Son86]).
  - `shareson` is a combination of the above two [CMB93], which may improve on the accuracy of any of them alone.
  - `shfr` tracks sharing and variables which are free (see [MH91]).
  - `shfrson` is a combination of `shfr` and `son`.
  - `shfrnv` augments `shfr` with knowledge on variables which are not free nor ground.
- for term structure:
  - `depth` tracks the structure of the terms bound to the program variables during execution, up to a certain depth; the depth is fixed with the `depth` flag.
  - `path` tracks sharing among variables which occur within the terms bound to the program variables during execution; the occurrence of run-time variables within terms is tracked up to a certain depth, fixed with the `depth` flag.
  - `aeq` tracks the structure of the terms bound to the program variables during execution plus the sharing among the run-time variables occurring in such terms, plus freeness and linearity. The depth of terms being tracked is set with the `depth` flag. Sharing can be selected between set-sharing or pair-sharing.
- for types:

  Type analysis supports different degrees of precision. For example, with the flag `type\_precision` with value `defined`, the analysis restricts the types to the finite domain of predefined types, i.e., the types defined by the user or in libraries, without generating new types. Another alternative is to use the normal analysis, i.e., creating new type definitions, but having only predefined types in the output. This is handled through the `type\_output` flag.
  - `eterms` performs structural widening (see [VB02]).

    Greater precision can be obtained evaluating builtins like `is/2` abstractly: `eterms` includes a variant which allows evaluation of the types, which is governed by the `type\_eval` flag.
  - `ptypes` uses the topological clash widening operator (see [VHCL95]).
  - `svterms` implements the rigid types domain of [JB92].
  - `terms` uses shortening as the widening operator (see [GdW94]), in several fashions, which are selected via the `depth` flag; depth 0 meaning the use of restricted shortening [SG94].

- for partial evaluation:

  Partial evaluation is performed during analysis when the `local\_control` flag is set to other than `off`. Flag `fixpoint` must be set to `di`. Unfolding will take place while analyzing the program, therefore creating new patterns to analyze. The unfolding rule is governed by flag `local\_control` (see `transformation(codegen)`).

  For partial evaluation to take place, an analysis domain capable of tracking term structure should be used (e.g., `eterms`, `pd`, etc.). In particular:

  - `pd` allows to perform traditional partial evaluation but using instead abstract interpretation with specialized definitions [PAH04].
  - `pdb` improves the precision of `pd` by detecting calls which cannot succeed, i.e., either loop or fail.

  Note that these two analyses will not infer useful information on the program. They are intended only to enable (classical) partial evaluation.
- for constraint domains:
  - `fr` [Dum94] determines variables which are not constraint to particular values in the constraint store in which they occur, and also keeps track of possible dependencies between program variables.
  - `frdef` is a combination of `fr` and `def`, determining at the same time variables which are not constraint to particular values and variables which are constraint to a definite value.
  - `lsign` [MS94] infers the signs of variables involved in linear constraints (and the possible number and form of such constraints).
  - `difflsign` is a simplified variant of `lsign`.
- for properties of the computation:
  - `det` detects procedures and goals that are deterministic (i.e. that produce at most one solution), or predicates whose clause tests are mutually exclusive (which implies that at most one of their clauses will succeed) even if they are not deterministic (because they call other predicates that can produce more than one solution).
  - `nfg` detects procedures that can be guaranteed not to fail (i.e., to produce at least one solution or not to terminate). It is a mono-variant non-failure analysis, in the sense that it infers non-failure information for only a call pattern per predicate [DLGH97].
  - `nf` detects procedures *and goals* that can be guaranteed not to fail and is able to infer separate non-failure information for different call patterns [BLGH04].
  - `seff` marks predicates as having side-effects or not.
- for size of terms:

  Size analysis yields functions which give bounds on the size of output data of procedures as a function of the size of the input data. The size can be expressed in various measures, e.g., term-size, term-depth, list-length, integer-value, etc.

  - `size\_ub` infers upper bounds on the size of terms.
  - `size\_lb` infers lower bounds on the size of terms.
  - `size\_ualb` infers both upper and lower bounds on the size of terms.
  - `size\_o` gives (worst case) complexity orders for term size functions (i.e. big O).
- for the number of resolution steps of the computation:

  Cost (steps) analysis yields functions which give bounds on the cost (expressed in the number of resolution steps) of procedures as a function of the size of their input data.

  - `steps\_ub` infers upper bounds on the number of resolution steps. Incorporates a modified version of the CASLOG [DL93] system, so that CiaoPP analyzers are used to supply automatically the information about modes, types, and size measures needed by the CASLOG system.
  - `steps\_lb` infers lower bounds on the number of resolution steps. Implements the analysis described in [DLGHL97].
  - `steps\_ualb` infers both upper and lower bounds on the number of resolution steps.
  - `steps\_o` gives (worst case) complexity orders for cost functions (i.e. big O).
- for the execution time of the computation:
  - `time\_ap` yields functions which give approximations on the execution time (expressed in milliseconds) of procedures as a function of the size of their input data.

Usage: `analysis(Analysis)`

`Analysis` is a valid analysis identifier.

transformation/1: PROPERTY

Transformations can be integrated in CiaoPP in an ad-hoc way (see the Internals manual), in which the CiaoPP menu would not be aware of them. The current transformations supported in the menu are:

- for program specialization:
  - `simp` This transformation tries to explote analysis information in order to *simplify* the program as much as possible. It includes optimizations such as abstract executability of literals, removal of useless clauses, and unfolding of literals for predicates which are defined by just a fact or a single clause with just one literal in its body (a *bridge*). It also propagates failure backwards in a clause as long as such propagation is safe.
  - `spec` This transformation performs the same optimizations as `simp` but it also performs multiple specialization when this improves the possibilities of optimization. The starting point for this transformation is not a program annotated with analysis information, as in the case above, but rather an *expanded program* which corresponds to the analysis graph computed by multi-variant abstract interpretation. A minimization algorithm is used in order to guarantee that the resulting program is minimal in the sense that further collapsing versions would represent losing opportunities for optimization.
  - `vers` This transformation has in common with `spec` that it takes as starting point the *expanded program* which corresponds to the analysis graph computed by abstract interpretation. However, this transformation performs no optimizations and does not minimize the program. As a result, it generates the expanded program.
- for partial evaluation:
  - `codegen` This generates the specialized program resulting from partial evaluation, obtained by unfolding goals during analysis. The kind of unfolding performed is governed by the `comp\_rule` flag, as follows:
    - `leftmost` unfolds the leftmost clause literal;
    - `eval\_builtin` selects for unfolding first builtins which can be evaluated;
    - `local\_emb` tries to select first atoms which do not endanger the embedding ordering or evaluable builtins whenever possible;

- `jump\_builtin` selects the leftmost goal but can 'jump' over (ignore) builtins when they are not evaluable. A main difference with the other computation rules is that unfolding is performed 'in situ', i.e., without reordering the atoms in the clause.
- `safe\_jb` same as `jump\_builtin` with the difference that it only jumps over a call to a builtin iff the call is safe [APG06] (i.e., it is error free, binding insensitive and side effect free).
- `bind\_ins\_jb` same as `safe\_jb` with the difference that it only jumps over a call to a builtin iff the call is binding insensitive and side effect free.
- `no\_sideff\_jb` same as `bind\_ins\_jb` with the difference that it only jumps over a call to a builtin iff it is side effect free.

Unfolding is performed continuously on the already unfolded clauses, until a condition for stopping the process is satisfied. This condition is stablished by the local control policy, governed by the `local\_control` flag, as follows:

- `inst` allows goal instantiation but no actual unfolding is performed.
- `orig` returns the clauses in the original program for the corresponding predicate.
- `det` allows unfolding while derivations are deterministic and stops them when a non-deterministic branch is required. Note that this may not be terminating.
- `det\_la` same as `det`, but with look-ahead. It can perform a number of non-deterministic steps in the hope that the computation will turn deterministic. This number is determined by flag `unf\_depth`.
- `depth` always performs the same number of unfolding steps for every call pattern. The number is determined by flag `unf\_depth`.
- `first\_sol` explores the SLD tree width-first and keeps on unfolding until a first solution is found. It can be non-terminating.
- `first\_sol\_d` same as above, but allows terminating when a given depth bound is reached without obtaining any solution. The bound is determined by `unf\_depth`.
- `all\_sol` tries to generate all solutions by exploring the whole SLD tree. This strategy only terminates if the SLD is finite.
- `hom\_emb` keeps on unfolding until the selected atom is homeomorphically embedded in an atom previously selected for unfolding.
- `hom\_emb\_anc` same as before, but only takes into account previously selected atoms which are ancestors of the currently selected atom.
- `hom\_emb\_as` same as before, but efficiently implemented by using a stack to store ancestors.
- `df\_hom\_emb\_as` same as before, but traverses the SLD tree on a depth-first fashion (all strategies above use wide-first search). This allows better performance.
- `df\_tree\_hom\_emb` same as above, but does not use the efficient stack-based implementation for ancestors.
- `df\_hom\_emb` same as above, but compares with all previously selected atoms, and not only ancestors. It is like `hom\_emb` but with depth-first traversal.

- `global\_control` In order to guarantee termination of the partial evaluation process, it is often required to abstract away information before unfolding. This is usually known as global control. This flag can have the following values:

- `off` unfolds always;
- `id` unfolds patterns which are not equal (modulo renaming) to a formerly analyzed pattern.
- `inst` unfolds patterns which are not an instance of a previous pattern.
- `hom\_emb` unfolds patterns which are not covered under the homeomorphic embedding ordering [Leu98].
- `hom\_emb\_num` same as `hom\_emb`, but also considers that any number embeds any other number.

Only `hom\_emb` guarantees termination. However, `id` and `inst` are more efficient, and terminating in many practical cases.

- `arg\_filtering` This transformation removes from program literals static values which are not needed any longer in the resulting program. This is typically the case when some information is known at compile-time about the run-time values of arguments.
- `codegen\_af` This performs `codegen` and `arg\_filtering` in a single traversal of the code. Good for efficiency.

- for code size reduction:
  - `slicing` This transformation is very useful for debugging programs since it isolates those predicates that are reachable from a given goal. The goals used are those exported by the module. The 'slice' being obtained is controlled by the following local control policies (described above): `df\_hom\_emb\_as`, `df\_hom\_emb`, `df\_tree\_hom\_emb`. It is also necessary to analyze the program with any of the currently available analyses for partial evaluation. Slicing is also very useful in order to perform other software engineering tasks, such as program understanding, maintenance, specialization, code reuse, etc.
- for program parallelization:

  Parallelization is performed by considering goals the execution of which can be deemed as *independent* [HR95,GHM00] under certain conditions. Parallel expressions (possibly conditional) are built from such goals, in the following fashions:
  - `mel` exploits parallel expressions which preserve the ordering of literals in the clauses;
  - `cdg` tries to exploit every possible parallel expression, without preserving the initial ordering;
  - `udg` is as above, but only exploits unconditional parallel expressions [MBdlBH99];
  - `urlp` exploits unconditional parallel expressions for NSIAP with *a posteriori* conditions [CH94].
  - `crlp` exploits conditional parallel expressions for NSIAP with *a posteriori* conditions.
  - `granul` This transformation allows to perform run-time task granularity control of parallelized code (see [LGHD96a]), so that the program will decide at run-time whether to run parallel expressions or not. The decision is based on the value of flag `granularity\_threshold`.
- for instrumenting the code for run-time assertion checking:
  - `rtchecks` Transforms the program so that it will check the predicate-level assertions at run-time.

Usage: `transformation(Transformation)`

`Transformation` is a valid transformation identifier.

3.4 Other information (ciaopp)

In this section the flags related with program analysis are explained in some detail. In particular, special attention is given to inter-modular program analysis and partial deduction (performed in CiaoPP during analysis).

3.4.1 Analysis with PLAI

Most of the analyses of CiaoPP are performed with the PLAI (Programming in Logic with Abstract Interpretation) framework [BGH94]. This framework is based on the computation of a fixed point for the information being inferred. Such a fixed point computation is governed by flag `fixpoint`, whose values are:

- `plai` for the classical fixed point computation [MH89a];
- `dd` for an incremental fixed point computation [HPMS00];
- `di` for the *depth independent* fixed point algorithm of [HPMS00];
- `check\_di` .

3.4.2 Inter-modular analysis

In inter-modular analysis CiaoPP takes into account the results of analyzing a module when other modules in the same program are analyzed. Thus, it collects analysis results (success patterns) for calls to predicates in other modules to improve the analysis of a given module. It also collects calls (call patterns) that are issued by the given module to other modules to reconsider them during analysis of such other modules.

Such flow of analysis information between modules while being analyzed can be performed when analyzing one single module. The information flow then affects only the modules imported by it. New call patterns will be taken into account when/if it is the turn for such imported modules to be analyzed. Improved success patterns will only be reused when/if the importing module is reanalyzed. However, CiaoPP can also iterate continuously over the set of modules of a given program, transferring the information from one module to others, and deciding which modules to analyze at which moment. This will be done until an inter-modular fixed point is reached in the analysis of the whole program (whereas analysis is performed one-module-at-a-time, anyway).

Inter-modular analysis is enabled with flag `intermod`. During inter-modular analysis there are several possible choices for selecting success patterns and call patterns. For example, when a success pattern is required for a given call pattern to an imported predicate, and there exist several that could be used, but none of them fit exactly with the given call pattern. Also, if, in that same case, there are no success patterns that fit (in which case CiaoPP has to make an *initial guess*). Finally, when there are new call patterns to a given module obtained during analysis of the modules that import it, which of them to use as entry points should be decided. All these features are governed by the following flags:

- `intermod` to activate inter-modular analysis.
  - `off` disables inter-modular analysis. This is the default value.
  - `on` enables inter-modular analysis.
  - `auto` allows the analysis of a modular program, using `intermod:auto\_analyze/2-3` with the main module of the program, iterating through the module graph until an inter-modular fixed point is reached. This value is set automatically by CiaoPP, and it should not be set by the user.
- `success\_policy` to obtain success information for given call patterns to imported predicates.

  - `best` selects the success pattern which corresponds to the best over-approximation of the sought call pattern; if there are several non-comparable best over-approximations, one of them is chosen randomly.
  - `first` selects the first success pattern which corresponds to a call pattern which is an over-approximation of the sought call pattern.
  - `all` computes the greatest lower bound of the success patterns that correspond to over-approximating call patterns.
  - `top` selects `Top` (no information) as answer pattern for any call pattern.
  - `botfirst` selects the first success pattern which corresponds to a call pattern which is an under-approximation of the sought call pattern.
  - `botbest` selects the success pattern which corresponds to the best under-approximation of the sought call pattern; if there are several non-comparable best under-approximations, one of them is chosen randomly.
  - `botall` computes the least upper bound of the success patterns that correspond to under-approximating call patterns.
  - `bottom` selects `Bottom` (failure) as answer pattern for any call pattern.
- `initial\_guess` to obtain an initial guess for the success pattern corresponding to a call pattern to an imported predicate when there is none that fully matches.
  - `botfirst` selects the success pattern already computed corresponding to the first call pattern which is an under-approximation of the given call pattern.
  - `botbest` selects the success pattern corresponding to the call pattern which best under-approximates the given call pattern (if there are several, non-comparable call patterns, one of them is selected randomly).
  - `botall` computes the least upper bound of the success patterns that correspond to under-approximating call patterns.
  - `bottom` selects `Bottom` as initial guess for any call pattern.
- `entry\_policy` to obtain entry call patterns for exported predicates.
  - `all` selects all entry call patterns for the current module which have not been analyzed yet, either from entry assertions found in the source code, or from the analysis of other modules that import the current module.
  - `top\_level` is only meaningful during `auto` inter-modular analysis, and it is set automatically by CiaoPP. If the current module is the top-level module (the main module of the modular program being analyzed), the entry policy behaves like `all`. In any other case, it selects entry call patterns for the current module from the analysis of other modules that import it, ignoring entry assertions found in the source code.
  - `force` forces the analysis of all entries of the module (from both the module source code and calling modules), even if they have been already analyzed.
  - `force\_assrt` forces the analysis of all entries coming from the module source code, but does not analyze entries relative to calling modules, even if they need to be (re)analyzed.
- `process\_libraries` to indicate that Ciao system libraries must also be analyzed when a modular user program is analyzed.
  - `off` disables the analysis of any Ciao system library.
  - `on` enables the analysis of all Ciao system libraries.
  - `no\_engine` enables the analysis of Ciao system libraries which are not engine libraries.
- `use\_check\_assrt` to indicate that check assertions for imported predicates will be used as trust assertions. This is specially interesting when performing intermodular compile-time checking.
  - `off` disables the use of check assertions as trust assertions for imported predicates.
  - `on` enables the use of check assertions as trust assertions.

3.4.3 Abstract partial deduction

Partial deduction (or partial evaluation) is a program transformation technique which specializes the program w.r.t. information known at compile-time. In CiaoPP this is performed during analysis of the program, so that not only concrete information but also abstract information (from the analysis) can be used for specialization. With analysis domain `pd` (and `pdb`) only concrete values will be used; with other analysis domains the domain abstract values inferred will also be used. This feature is governed by the following flags:

- `abs\_spec\_defs` to exploit abstract substitutions in order to:
  - `rem` try to eliminate clauses which are incompatible with the inferred substitution at each unfolding step;
  - `exec` perform abstract executability of atoms;
  - `all` do both.
- `part\_concrete` to try to convert abstract information into concrete information if possible, so that:
  - `mono` one concrete atom is obtained;
  - `multi` multiple atoms are allowed when the information in the abstract substitution is disjunctive.
- `rem\_use\_cls` to identify clauses which are incompatible with the abstract call substitution and remove them:
  - `pre` prior to performing any unfolding steps;
  - `post` after performing unfolding steps;
  - `both` both before and after performing unfolding steps.
- `filter\_nums` to filter away during partial evaluation numbers which:
  - `safe` are not safe, i.e., do not appear in the original program, or
  - `on` all numbers.

3.5 Known bugs and planned improvements (`ciaopp`)

- 1 The ciaopp version number is now hardwired instead of being automatically updated

4 The CiaoPP command-line interface

Author(s): The CLIP Group.

The command-line interface of CiaoPP allows the use of the system in batch mode, using command-line arguments for setting preprocessor flags and performing actions.

4.1 Command-line options

This interface can be used by means of the following command-line options:

```
Usage 1: (batch mode)
ciaoppcl [-o OutFile] Option Filename [FlagsValues]

  Where:
    -o OutFile  after processing Filename, the resulting source
                code is written to OutFile.  If this option is
                omitted, the output is written to a file
                automatically named depending on the actions
                performed.

    Option must be one of the following:
    -Q          runs the interactive (text-based) menu for
                preprocessing Filename.
    -A          analyzes Filename with the default options
                except the flag values set with -f at the
                command line.
    -O          optimizes Filename with the default options
                except the flag values set with -f at the
                command line.
    -V          verifies the assertions of Filename with
                the default options except the flag values set
                with -f at the command line.
    -U Config   processes Filename with the
                options set in the CiaoPP configuration Config.

    FlagsValues is a list of options -fFlagName=FlagValue
    separated by blank spaces, where FlagName is a valid
    CiaoPP flag name.  This list is optional, and does not need
    to include all flags applicable to the action to be performed:
    the flags not included in this list will be assumed to take
    their default value.  Examples:

    -flocal_control=on   where local_control is expected to be
                         a CiaoPP flag;
    -f local_control=on  same as above, with additional blank spaces

    Internal flags can also be changed using -pIntFlagName=Value.

Usage 2: (top-level mode)
       ciaoppcl -T

  -T option starts a CiaoPP top-level shell.  Any of the predicates
```

```
  described in the Section CiaoPP User Menu Interface of the CiaoPP
  Reference Manual can be used in this top-level.

Execution Examples:

  ciaoppcl -Q myfile.pl
  ciaoppcl -o myfile_checked.pl -V myfile.pl
  ciaoppcl -O myfile.pl
  ciaoppcl -A myfile.pl -ftypes=terms -f modes=pd
  ciaoppcl -T
```

4.2 Description of the execution examples

- The following command will prompt the user with the options needed to preprocess `myfile.pl`:

  `ciaoppcl -Q myfile.pl`
- If we want to verify the assertions of `myfile.pl`, and generate the resulting source code that will the new status of the assertions (either `checked`, if CiaoPP has proved that the assertion holds, or `false` if it has falsified the assertion), the command line is as follows:

  `ciaoppcl -o myfile_checked.pl -V myfile.pl`
- To optimize `myfile.pl`, and write the optimize code in a file named automatically (e.g., `myfile_pd_codegen_af_co.pl`), the following command line must be used:

  `ciaoppcl -O myfile.pl`
- If the default flag values need to be changed, the `-f` option can be used. For example, in order to analyze `myfile.pl` to change the types analysis domain to `terms` instead of the default one, and the mode-aliasing domain to `pd`, the command line to use should be:

  `ciaoppcl -A myfile.pl -ftypes=terms -f modes=pd`
- Finally, the following command line can be used to start a top-level CiaoPP shell:

  `ciaoppcl -T`

PART II - The Assertion Language and Its Use

Author(s): The CLIP Group.

5 Using assertions for preprocessing programs

Author(s): Francisco Bueno.

This chapter explains the use of assertions to specify a program behaviour and properties expected to hold of the program. It also clarifies the role of assertion-related declarations so that a program can be statically preprocessed with CiaoPP.

CiaoPP starts a preprocessing session from a piece of code, annotated with assertions. The code can be either a complete self-contained program or part of a larger program (e.g., a module, or a user file which is only partial). The assertions annotating the code describe some properties which the programmer requires to hold of the program. Assertions are used also to describe to the static analyzer some properties of the interface of the code being preprocessed at a given session with other parts of the program that code belongs to. In addition, assertions can be used to provide information to the static analyzer, in order to guide it, and also to control specialization and other program transformations.

This chapter explains the use of assertions in describing to CiaoPP: (1) the program specification, (2) the program interface, and (3) additional information that might help static preprocessing of the program.

In the following, the Ciao assertion language is briefly described and heavily used. In Chapter 6 [The Ciao assertion package], page 47, a complete reference description of assertions is provided. More detailed explanations of the language can be found in [PBH00].

This chapter also introduces and uses properties, and among them (regular) types. See Chapter 9 [Basic data types and properties], page 69, for a concrete reference of (some of) the Ciao properties. See Chapter 8 [Declaring regular types], page 63, for a presentation of the Ciao type language and an explanation on how you can write your own properties and types.

Most of the predicates used below which are not defined belong to the *ISO-Prolog* standard [DEDC96]. The builtin (or primitive) constraints used have also become more or less de-facto standard. For detailed descriptions of particular constraint logic programming builtins refer for example to the CHIP [COS96], PrologIV [PRO], and Ciao [BCC04] manuals.

5.1 Assertions

Predicate assertions can be used to declare properties of the execution states at the time of calling a predicate and upon predicate success. Also, properties of the computation of the calls to a predicate can be declared.

Assertions may be qualified by keywords `check` or `trust`. Assertions qualified with the former—or not qualifed—are known as check assertions; those qualified with the latter are known as trust assertions. Check assertions state the programmer's intention about the program and are used by the debugger to check for program inconsistencies. On the contrary, trust assertions are "trusted" by CiaoPP tools.

- The specification of a program is made of all check assertions for the program predicates.

5.1.1 Properties of success states

They are similar in nature to the *postconditions* used in program verification. They can be expressed in our assertion language using the basic assertion:

```
:- success Goal => Postcond.
```

This assertion should be interpreted as, "for any call of the form `Goal` which succeeds, on success `Postcond` should also hold" .

Note that, in contrast to other programming paradigms, calls to a predicate may either succeed or fail. The postcondition stated in a `success` assertion only refers to successful executions.

5.1.2 Restricting assertions to a subset of calls

Sometimes we are interested in properties which refer not to all invocations of a predicate, but rather to a subset of them. With this aim we allow the addition of preconditions (`Precond`) to predicate assertions as follows: '`Goal : Precond`'.

For example, `success` assertions can be restricted and we obtain an assertion of the form:

```
:- success Goal : Precond => Postcond.
```

which should be interpreted as, "for any call of the form `Goal` for which `Precond` holds, if the call succeeds then on success `Postcond` should also hold".

5.1.3 Properties of call states

It is also possible to use assertions to describe properties about the calls for a predicate which may appear at run-time. An assertion of the form:

```
:- calls Goal : Cond.
```

must be interpreted as, "all calls of the form `Goal` should satisfy `Cond`".

5.1.4 Properties of the computation

Many other properties which refer to the computation of the predicate (rather than the input-output behaviour) are not easily expressible using `calls` and `success` predicate assertions only. Examples of properties of the computation which we may be interested in are: non-failure, termination, determinacy, non-suspension, etc.

This sort of properties are expressed by an assertion of the form:

```
:- comp Goal : Precond + Comp-prop.
```

which must be interpreted as, "for any call of the form `Goal` for which `Precond` holds, `Comp-prop` should also hold for the computation of `Goal`". Again, the field '`: Precond`' is optional.

5.1.5 Compound assertions

In order to facilitate the writing of assertions, a compound predicate assertion can be used as syntactic sugar for the above mentioned basic assertions. Each compound assertion is translated into one or several basic assertions, depending on how many of the fields in the compound assertion are given. The compound assertion is as follows.

```
:- pred Pred : Precond => Postcond + Comp-prop.
```

Each such compound assertion corresponds to: a `success` assertion of the form:

```
:- success Pred : Precond => Postcond.
```

if the `pred` assertion has a `=>` field (and a `:` field). It also corresponds to a `comp` assertion of the form:

```
:- comp Pred : Precond + Comp-prop.
```

if the `pred` assertion has a `+` field (and a `:` field).

All compound assertions given for the same predicate correspond to a single `calls` assertion. This `calls` assertion states as properties of the calls to the predicate a disjunction of the properties stated by the different compound assertions in their `:` field. Thus, it is of the form:

```
:- calls Pred : ( Precond1 ; ... ; Precondn ).
```

for all the `Precondi` in the `:` fields of (all) the different `pred` assertions.

Note that when compound assertions are used, `calls` assertions are always implicitly generated. If you do not want the `calls` assertion to be generated (for example because the set of assertions available does not cover all possible uses of the predicate) basic `success` or `comp` assertions rather than compound (`pred`) assertions should be used.

5.1.6 Examples of compound assertions

Consider the classical qsort program for sorting lists. We can use the following assertion in order to require that the output of procedure `qsort` be a list:

```
:- success qsort(A,B) => list(B).
```

Alternatively, we may require that if `qsort` is called with a list in the first argument position and the call succeeds, then on success the second argument position should also be a list. This is declared as follows:

```
:- success qsort(A,B) : list(A) => list(B).
```

The difference with respect to the previous assertion is that `B` is only expected to be a list on success of predicate `qsort/2` if `A` was a list at the call.

In addition, we may also require that in all calls to predicate `qsort` the first argument should be a list. The following assertion will do:

```
:- calls qsort(A,B) : list(A).
```

The `qsort` procedure should be able to sort all lists. Thus, we also require that all calls to it that have a list in the first argument and a variable in the second argument do not fail:

```
:- comp qsort(A,B) : (list(A) , var(B)) + does_not_fail.
```

Instead of the above basic assertions, the following compound one could be given:

```
:- pred qsort(A,B) : (list(A) , var(B)) => list(B) + does_not_fail.
```

which will be equivalent to:

```
:- calls qsort(A,B) : (list(A), var(B)).
:- success qsort(A,B) : (list(A), var(B)) => list(B).
:- comp qsort(A,B) : (list(A) , var(B)) + does_not_fail.
```

This will not allow to call `qsort` with anything else than a variable as second argument. If this use of `qsort` is expected, one should have added the assertion:

```
:- pred qsort(A,B) : list(A) => list(B).
```

which, together with the above one, will imply:

```
:- calls qsort(A,B) : ((list(A), var(B)) ; list(A)).
```

Then it is only required that `A` be a list.

5.2 Properties

Whereas each kind of assertion indicates *when*, i.e., in which states or sequences of states, to check the given properties, the properties themselves define *what* to check. Properties are used to say things such as "X is a list of integers," "Y is ground," "p(X) does not fail," etc. and in Ciao they are logic predicates, in the sense that the evaluation of each property either succeeds or fails. The failure or success of properties typically needs to be determined at the time when the assertions in which they appear are checked. Assertions can be checked both at compile-time by CiaoPP and at run-time by Ciao itself (after the instrumentation of the program by CiaoPP). In this section we will concentrate exclusively on run-time checking.

A property may be a predefined predicate in the language (such as `integer(X)`) or constraint (such as `X>5`). Properties may include extra-logical predicates such as `var(X)`). Also, expressions built using conjunctions of properties,[1] or, in principle, any predicate defined by the user, using the full underlying (C)LP language. As an example, consider defining the predicate `sorted(B)` and using it as a postcondition to check that a more involved sorting algorithm such as `qsort(A,B)` produces correct results.

[1] Although disjunctions are also supported, we restrict our attention to only conjunctions.

While user-defined properties allow for properties that are as general as allowed by the full source language syntax, some limitations are useful in practice. Essentially, the behaviour of the program should not change in a fundamental way depending on whether the run-time tests are being performed or not. For example, turning on run-time checking should not introduce non-termination in a program which terminates without run-time checking. To this end, it is required that the user ensure that the execution of properties terminate for any possible initial state. Also, checking a property should not change the answers computed by the program or produce unexpected side-effects. Regarding computed answers, in principle properties are not allowed to further instantiate their arguments or add new constraints. Regarding side-effects, it is required that the code defining the property does not perform input/output, add/delete clauses, etc. which may interfere with the program behaviour. It is the user's responsibility to only use predicates meeting these conditions as properties. The user is required to identify in a special way the predicates which he or she has determined to be legal properties. This is done by means of a declaration of the form

```
:- prop Spec.
```

where `Spec` is a predicate specification in the form `PredName/Arity`.

Given the classes of assertions presented previously, there are two fundamental classes of properties. The properties used in the `Cond` of calls assertions, `Postcond` of success assertions, and `Precond` of success and comp assertions refer to a particular execution state and we refer to them as *properties of execution states*. The properties used in the `Comp-prop` part of comp assertions refer to a sequence of states and we refer to them as *properties of computations*.

Basic properties, including instantiation and compatibility state properties, types, and properties of computations (all discussed in Chapter 8 [Declaring regular types], page 63) are documented in Chapter 9 [Basic data types and properties], page 69.

5.3 Preprocessing units

The preprocessing unit is the piece of code that is made available to CiaoPP at a given preprocessing session. Normally, this is a file, but not all the code of a program is necessarily contained in one single file: in order to statically manipulate the code in a file, CiaoPP needs to know the interactions of this code with other pieces of the program—probably scattered over other files—, as well as what the user's interaction with the code will be upon execution. This is also done through the use of assertions.

If the preprocessing unit is self-contained the only interaction of its code (apart from calling the builtin predicates of the language) is with the user. The user's interaction with the program consists in querying the program. The predicates that may be directly queried by the user are entry points to the preprocessing unit.

Entry points can be declared in two ways: using a module declaration specifying the entry points, or using one entry declaration for each entry point. If entry declarations are used, instead of, or in addition to, the module declaration, they can also state properties which will hold at the time the predicate is called.

However, if the preprocessing unit is not self-contained, but only part of a larger program, then other interactions may occur. The interactions of the preprocessing unit include: the user's queries, calls from other parts of the program to the unit code, calls to the unit code from unit code which does not appear explicitely in the unit text, and calls from the unit code to other parts of the program.

First, other parts of the program can call predicates defined in the preprocessing unit. CiaoPP needs to know this information. It must be declared by specifying additional entry points, together with those corresponding to the user's queries.

Second, the preprocessing unit itself may contain meta-calls which may call any unspecified predicate. All predicates that may be called in such a way should be declared also as entry points.

Additional entry points also occur when there are predicates defined in the preprocessing unit which can be dynamically modified. In this case the code dynamically added can contain new predicate calls. These calls should be declared also as entry points.

Note that *all* entry points to the preprocessing unit should be declared: entry points including query calls that the user may issue to the program, or another part of the program can issue to the unit, but also *dynamic calls*: goals that may be run within the unit which do not appear explicitely in the unit text, i.e., from meta-predicates or from dynamic clauses which may be asserted during execution. In all cases, *entry* declarations are used to declare entry points.[2]

Third, the unit code may call predicates defined in other parts of the program. The code defining such predicates is termed *foreign code*, since it is foreign to the preprocessing unit. It is important that CiaoPP knows information about how calls to foreign code will succeed (if they succeed), in order to improve its accuracy. This can be done using *trust* declarations.

Also, trust declarations can be used to provide the preprocessor with extra information. They can be used to describe calls to predicates defined within the preprocessing unit, in addition to those describing foreign code. This can improve the information available to the preprocessor and thus help it in its task. Trust declarations state properties that the programmer knows to hold of the program.

The builtin predicates is one particular case of predicates the definitions of which are never contained in the program itself. Therefore, preprocessing units never contain code to define the builtins that they use. However, the Ciao Program Precompiler makes no assumptions on the underlying language (except that it is constraint logic programming). Thus, all information on the behaviour of the language builtins should be made available to it by means of assertions (although this does not concern the application programmer who is going to preprocess a unit, rather it concerns the system programmer when installing the Ciao Program Precompiler).

The rest of this document summarizes how assertions can be used to declare the preprocessing unit interactions. It shows the use of entry and trust declarations in preprocessing programs with CiaoPP.[3]

5.4 Foreign code

A program preprocessing unit may make use of predicates defined in other parts of the program. Such predicates are foreign to the preprocessing unit, i.e., their code is not in the unit itself. In this case, CiaoPP needs to know which is the effect that such predicates may cause on the execution of the predicates defined in the unit. For this purpose, trust declarations are used.

Foreign code includes predicates defined in other modules which are used by the preprocessing unit, predicates defined in other files which do not form part of the preprocessing unit but which are called by it, builtin predicates[4] used by the code in the preprocessing unit, and code written in a foreign language which will be linked with the program. All foreign calls (except to builtin predicates) need to be declared.[5]

---

[2] When the language supports a module system, entry points are implicitely declared by the exported predicates. In this case entry declarations are only used for local predicates if there are dynamic calls.

[3] This manual concentrates on one particular use of the declarations for solving problems related to compile-time program analysis. However, there are other possible solutions. For a complete discussion of these see [BCHP96].

[4] However, builtin predicates are usually taken care of by the system programmer, and the preprocessor, once installed, already "knows" them.

[5] However, if the language supports a module system, and the preprocessor is used in modular analysis operation mode, trust declarations are imported from other modules and do not need to be declared in the preprocessing unit.

- The effect of calls to foreign predicates may be declared by using trust declarations for such predicates.

Trust declarations have the following form:

```
:- trust success Goal : ( Prop, ..., Prop )
                     => ( Prop, ..., Prop ).
```

where `Goal` is an atom of the foreign predicate, with all arguments single distinct variables, and `Prop` is an atom which declares a property of one (or several) of the goal variables.

The first list of properties states the information at the time of calling the goal and the second one at the time of success of the goal. Thus, such a trust assertion declares that for any call to the predicate where the properties in the first list hold, those of the second will also hold upon success of the call.

Simplified versions of trust assertions can also be used, much the same as with entry declarations. See Section 5.1 [Assertions], page 37.

Trust declarations are a means to provide the preprocessor with extra information about the program states. This information is guaranteed to hold, and for this reason the preprocessor *trusts* it. Therefore, it should be used with great care, since if it is wrong the precompilation of your program will possibly be wrong.

5.4.1 Examples of trust assertions

The following annotations describe the behavior of the predicate `p/2` for two possible call patterns:

```
:- trust success p/2 : def * free => def * def.
:- trust success p/2 : free * def => free * def.
```

This would allow performing the analysis even if the code for `p/2` is not present. In that case the corresponding success information in the annotation can be used ("trusted") as success substitution.

In addition, trust declarations can be used to improve the results of compile-time program analysis when they are imprecise. This may improve the accuracy of the debugging, possibly allowing it to find more bugs.

5.5 Dynamic predicates

Predicate definitions can be augmented, reduced, and modified during program execution. This is done through the database manipulation builtins, which include `assert`, `retract`, `abolish`, and `clause`. These builtins (with the exception of `clause`) dynamically manipulate the program itself by adding to or removing clauses from it. Predicates that can be affected by such builtins are called dynamic predicates.

There are at least two possible classes of dynamic predicates which behave differently from the point of view of static manipulation. First, clauses can be asserted and/or retracted to maintain an information database that the program uses. In this case, usually only facts are asserted. Second, full clauses can be asserted for predicates which are also called within the program.

The first class of dynamic predicates are declared by data declarations. The second class by dynamic declarations. The form of both declarations is as follows:

```
:- data Spec, ..., Spec.
:- dynamic Spec, ..., Spec.
```

where `Spec` is a predicate specification in the form `PredName/Arity`.

- Dynamic predicates which are called must be declared by using a dynamic declaration.

Of course, the preprocessor cannot know of the effect that dynamic clauses added to the definition of a predicate may cause in the execution of that predicate. However, this effect can be described to the preprocessor by adding trust declarations for the dynamic predicates.

- The effect of calls to predicates which are dynamically modified may be declared by using trust declarations for such predicates.

5.6 Entry points

In a preprocessing session (at least) one entry point to the preprocessing unit is required. It plays a role during preprocessing similar to that of the query that is given to the program to run. Several entry points may be given. Entry points are given to the preprocessor by means of entry or module declarations.

If the preprocessing unit is a module, only the exported predicates can be queried. If the preprocessing unit is not a module, all of its predicates can be queried: all the unit predicates may be entry points to it. Entry declarations can then be used by the programmer to specify additional information about the properties that hold of the arguments of a predicate call when that predicate is queried.

Note that if the unit is not a module all of its predicates are considered entry points to the preprocessor. However, if the unit incorporates some entry declarations the preprocessor will act as if the predicates declared were the only entry points (the preprocessing session being valid for a particular use of the unit code—that specified by the entry declarations given).

- All predicates that can be queried by the user and all predicates that can be called from parts of the program which do not explicitely appear in the preprocessing unit should (but need not) be declared as entry points by using entry declarations.

The entry declaration has the following form:

```
:- entry Goal : ( Prop, ..., Prop ).
```

where `Goal` is an atom of the predicate that may be called, with all arguments single distinct variables, and `Prop` is an atom which declares a property of one (or several) of the goal variables. The list of properties is optional.

There are alternative formats in which the properties can be given: as the arguments of `Goal` itself, or as keywords of the declaration. For a complete reference of the syntax of assertions, see Section 5.1 [Assertions], page 37.

5.6.1 Examples of entry declarations

Consider the following program:

```
append([], L, L).
append([H|T], L, [H|R]) :- append(T, L, R).
```

It may be called in a classical way with the first two arguments bound to lists, and the third argument a free variable. This can be annotated in any of the following three ways:

```
:- entry append(X,Y,Z) : ( list(X), list(Y), var(Z) ).
:- entry append/3 : list * list * var.
:- entry append(list,list,var).
```

Assume you have the following program:

```
p(X,Y):- q(X,Y,Z).
q(X,Y,Z):- X = f(Y,Z), Y + Z = 3.
```

Assume that `p/2` is the only entry point. If you include the following declaration:

```
:- entry p/2.
```

or, equivalently,

```
:- entry p(X,Y).
```

the code will be preprocessed as if goal `p(X,Y)` was called with the most general call pattern (i.e., as if `X` and `Y` may have any two values, or no value at all—the variables being free).

However, if you know that `p/2` will always be called with the first argument uniquely defined and the second unconstrained, you can then provide more accurate information by introducing one of the following declarations:

```
:- entry p(X,Y) : ( def(X), free(Y) ).
:- entry p(def,free).
```

Now assume that `p/2` will always be called with the first argument bound to the compound term `f(A,B)` where `A` is definite and `B` is unconstrained, and the second argument of `p/2` is unconstrained. The entry declaration for this call pattern is:

```
:- entry p(X,Y) : ( X=f(A,B), def(A), free(B), free(Y) ).
```

If both call patterns are possible, the most accurate approach is to include both entry declarations in the preprocessing unit. The preprocessor will then analyze the program for each declaration. Another alternative is to include an entry declaration which approximates both call patterns, such as one of the following two:

```
:- entry p(X,Y) : free(Y).
:- entry p(X,free).
```

which state that `Y` is known to be free, but nothing is known of `X` (since it may or may not be definite).

5.7 Modules

Modules provide for encapsulation of code, in such a way that (some) predicates defined in a module can be used by other parts of the program (possibly other modules), but other (auxiliary) predicates can not. The predicates that can be used are exported by the module defining them and imported by the module(s) which use(s) them. Thus, modules provide for a natural declaration of the allowed entry points to a piece of a program.

A module is identified by a module declaration at the beginning of the file defining that module. The module declaration has the following form:

```
:- module(Name, [ Spec,...,Spec ] ).
```

where the module is named `Name` and it exports the predicates in the different `Spec`'s.

Note that such a module declaration is equivalent, for the purpose of static preprocessing, to as many entry declarations of the form:

```
:- entry Spec.
```

as there are exported `Spec`'s.

5.8 Dynamic calls

In addition to entry points there are other calls that may occur from within a piece of code which do not explicitely appear in the code itself. Among these are metacalls, callbacks, and calls from clauses which are asserted during program execution.

Metacalls are literals which call one of their arguments at run-time, converting at the time of the call a term into a goal. Predicates in this class are not only `call`, but also `bagof`, `findall`, `setof`, negation by failure, and `once` (single solution).

Metacalls may be static, and this kind of calls need not be declared. A static metacall is, for example, `once(p(X))`, where the predicate being called is statically identifiable (since it appears in the code). On the other hand, metacalls of the form `call(Y)` are dynamic, since the predicate being called will only be determined at runtime.[6]

Callbacks are also metacalls. A callback occurs when a piece of a program uses a different program module (or object) in such a way that it provides to that module the call that it should issue upon return. Callbacks, much the same as metacalls, can be either dynamic or static. Only the predicates of the preprocessing unit which can be dynamically called-back need be declared.

Clauses that are asserted during program execution correspond to code which is dynamically created; thus, the preprocessor cannot be aware of such code during a (compile-time) preprocessing session. The calls that may appear from the body of a clause which is dynamically created and asserted are also dynamic calls.

- All dynamic calls must be declared by using entry declarations for the predicates that can be called in a dynamic way.

5.8.1 Examples of dynamic calls

Consider a program where you use the `bagof` predicate to collect all solutions to a goal, and the program call looks like:

```
p(X,...) :- ..., bagof(P,X,L), ...
```

However, you know that, upon execution, only the predicates `p/2` and `q/3` will be called by `bagof`, i.e., `X` will only be bound to terms with functors `p/2` and `q/3`. Moreover, such terms will have all of their arguments constrained to definite values. This information should be given to the preprocessor using the declarations:

```
:- entry p(def,def).
:- entry q(def,def,def).
```

Assume you have a graphics library predicate `menu_create/5` which creates a graphic menu. The call must specify, among other things, the name of the menu, the menu items, and the menu handler, i.e., a predicate which should be called upon the selection of a menu item. The predicate is used as:

```
top :- ..., menu_create(Menu,0,Items,Callback,[]), ...
```

but the program is coded so that there are only two menu handlers: `app_menu/2` and `edit_menu/2`. The first one handles menu items of the type `app_item` and the second one items of the type `edit_item`. This should be declared with:

```
:- entry app_menu(gnd,app_item).
:- entry edit_menu(gnd,edit_item).
```

[6] However, sometimes analysis techniques can be used to transform dynamic metacalls into static ones.

Let a program have a dynamic predicate `dyn_calls/1` to which the program asserts clauses, such that these clauses do only have in their bodies calls to predicates `p/2` and `q/3`. This should be declared with:

```
:- entry p/2.
:- entry q/3.
```

Moreover, if the programmer knows that every call to `dyn_calls/1` which can appear in the program is such that upon its execution the calls to `p/2` and `q/3` have all of their arguments constrained to definite values, then the two entry declarations at the beginning of the examples may be used.

5.9 An overview

To process programs with the Ciao Program Precompiler the following guidelines might be useful:

1. Add

   ```
   :- use_package(assertions).
   ```

   to your program.
2. Declare your specification of the program using calls, success, comp, or pred assertions.
3. Use entry declarations to declare all entry points to your program.
4. The preprocessor will notify you during the session of certain program points where a meta-call appears that may call unknown (at compile-time) predicates.

   Add entry declarations for all the predicates that may be dynamically called at such program points.
5. Use data or dynamic declarations to declare all predicates that may be dynamically modified.
6. Add entry declarations for the dynamic calls that may occur from the code that the program may dynamically assert.
7. Optionally, you can interact with the preprocessor using trust assertions.

   For example, the preprocessor will notify you during the session of certain program points where a call appears to an unknown (at compile-time) predicate.

   Add trust declarations for such predicates.

6 The Ciao assertion package

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.

Giving specifications for predicates and other program elements is the main functionality documented here. The exact syntax of comments is described in the autodocumenter (`lpdoc` [Knu84,Her99]) manual, although some support for adding machine-readable comments in assertions is also mentioned here.

There are two kinds of assertions: predicate assertions and program point assertions. All predicate assertions are currently placed as directives in the source code, i.e., preceded by "`:-`". Program point assertions are placed as goals in clause bodies.

6.1 More info

The facilities provided by the library are documented in the description of its component modules. This documentation is intended to provide information only at a "reference manual" level. For a more tutorial introduction to the subject and some more examples please see [PBH00]. The assertion language implemented in this library is modeled after this design document, although, due to implementation issues, 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.

6.2 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.

6.3 Usage and interface (assertions_doc)

- **Library usage:**
  The recommended procedure in order to make use of assertions in user programs is to include the `assertions` syntax library, using one of the following declarations, as appropriate:

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

- **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], `test/1` [1150,fx], `test/2` [1150,xfx], `texec/1` [1150,fx], `texec/2` [1150,xfx], `comp/1` [1150,fx], `comp/2` [1150,xfx], `entry/1` [1150,fx], `exit/1` [1150,fx], `exit/2` [1150,xfx].
- **New declarations defined:**
  `pred/1`, `pred/2`, `texec/1`, `texec/2`, `calls/1`, `calls/2`, `success/1`, `success/2`, `test/1`, `test/2`, `comp/1`, `comp/2`, `prop/1`, `prop/2`, `entry/1`, `exit/1`, `exit/2`, `modedef/1`, `decl/1`, `decl/2`, `doc/2`, `comment/2`.
- **Imports:**
  - *System library modules:*
    `assertions/assertions_props`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`.

6.4 Documentation on new declarations (assertions_doc)

pred/1: DECLARATION

This assertion provides information on a predicate. The body of the assertion (its only argument) contains properties or comments in the formats defined by `assrt_body/1`.

More than one of these assertions may appear 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). All of them together cover all possible modes of usage.

For example, the following assertions describe (all the and the only) modes of usage of predicate `length/2` (see `lists`):

```
:- pred length(L,N) : list * var => list * integer
# "Computes the length of L.".
:- pred length(L,N) : var * integer => list * integer
```

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

Usage: :- `pred AssertionBody`.

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

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

pred/2: DECLARATION

This assertion is similar to a `pred/1` assertion but it is explicitely qualified. Non-qualified `pred/1` assertions are assumed the qualifier `check`.

Usage: :- `AssertionStatus pred 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`)

texec/1: DECLARATION

This assertion is similar to a `calls/1` assertion but it is used to provide input data and execution commands to the unit-test driver.

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: :- `AssertionStatus texec 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`)

calls/1: DECLARATION

This assertion is similar to a `pred/1` assertion but it only provides information about the calls to a predicate. If one or several calls assertions are given they are understood to 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

This assertion is similar to a `calls/1` assertion but it is explicitely qualified with an assertion status. Non-qualified `calls/1` assertions are assumed to have `check` status.

Usage: :- `AssertionStatus calls 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

This assertion is similar to a `pred/1` assertion but it only provides information about the answers to a predicate. The described answers might be conditioned to a particular way of calling the predicate.

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 compelled the predicate 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

success assertion This assertion is similar to a `success/1` assertion but it is explicitely qualified with an assertion status. The status of non-qualified `success/1` assertions is assumed to be `check`.

Usage: :- `AssertionStatus success 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`)

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: :- `AssertionStatus test 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

This assertion is similar to a `pred/1` assertion but it only provides information about the global execution properties of a predicate (note that such kind of information is also conveyed by pred assertions). The described properties might be conditioned to a particular way of calling the predicate.

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 compel the predicate to be called 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

This assertion is similar to a `comp/1` assertion but it is explicitely qualified. Non-qualified `comp/1` assertions are assumed the qualifier `check`.

Usage: :- `AssertionStatus comp 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

This assertion is similar to a `pred/1` assertion but it flags 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 8 [Declaring regular types], page 63 for some considerations applicable to writing properties.)

The set of properties is thus a strict subset of the set of predicates. Note that properties can be used to describe characteristics of arguments in assertions and 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: :- `AssertionStatus prop 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`)

entry/1: DECLARATION

This assertion provides information about the *external* calls to a predicate. It is identical syntactically to a `calls/1` assertion. However, they describe only external calls, i.e., 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. As a result, if their descriptions are erroneous they can introduce bugs in programs. Thus, `entry/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. 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.

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.
```

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: :- `AssertionStatus exit 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) # "A is bound upon predicate entry.".

:- pred p(+A,B) : integer(A) =>  ground(B).
```

is equivalent to:

```
:- pred p(A,B) : (nonvar(A),integer(A)) =>  ground(B)
   # "A is bound upon predicate entry.".
```

Usage: :- `modedef AssertionBody`.

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

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

decl/1: DECLARATION

This assertion is similar to a `pred/1` assertion but it is used for declarations instead than for 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 is explicitely qualified. Non-qualified `decl/1` assertions are assumed the qualifier `check`.

Usage: :- `AssertionStatus decl 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`.

– *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`)

6.5 Documentation on exports (`assertions_doc`)

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 Chapter 11 [Run-time checking of assertions], page 101.

– *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`)

7 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.

7.1 Usage and interface (`assertions_props`)

- **Library usage:**
  `:- use_module(library(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`.
- **Imports:**
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`, `dcg`, `assertions`, `regtypes`.

7.2 Documentation on exports (`assertions_props`)

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.

  Acceptable modes are documented in `library(basicmodes)` and `library(isomodes)`. User defined modes are documented in `modedef/1`.
- 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(A,int)` 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(Parameter,int)` 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(Y,integer))`: `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.

8 Declaring regular types

Author(s): Manuel Hermenegildo, Pedro López, Francisco Bueno.

This library package adds declarations and new operator definitions which provide simple syntactic sugar to write regular type definitions in source code. Regular types are just properties which have the additional characteristic of being regular types (`basic_props:regtype/1`), defined below.

For example, this library package allows writing:

```
:- regtype tree(X) # "X is a tree.".
```

instead of the more cumbersome:

```
:- prop tree(X) + regtype # "X is a tree.".
```

Regular types can be used as properties to describe predicates and play an essential role in program debugging (see the Ciao Prolog preprocessor (`ciaopp`) manual).

In this chapter we explain some general considerations worth taking into account when writing properties in general, not just regular types.

8.1 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 Chapter 11 [Run-time checking of assertions], page 101) 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_fail/1` (note that although it has been used as in `:- comp append(Xs,Ys,Zs) + not_fail`, it is in fact read as `not_fail(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_fail/1` for run-time checking:

```
not_fail(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.

8.2 Usage and interface (regtypes_doc)

- **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`.
- **Imports:**
  - *System library modules:*
    `assertions/assertions_props`.
  - *Internal (engine) modules:*
    `term_basic`.
  - *Packages:*
    `prelude`, `assertions`, `pure`.

8.3 Documentation on new declarations (regtypes_doc)

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 lelow). 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(x, v_1, ..., v_n)  :- 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(y, t_1, ..., t_m)` 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 first 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 9 [Basic data types and properties], page 69.

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: :- `AssertionStatus regtype 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`)

9 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 can be used both as type testing builtins within programs (by calling them explicitly) and as properties in assertions.

9.1 Usage and interface (basic_props)

- **Library usage:**
  These predicates are builtin in Ciao, so nothing special has to be done to use them.
- **Exports:**
  - *Properties:*
    `member/2`, `compat/2`, `inst/2`, `iso/1`, `deprecated/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`, `callable/1`, `operator_specifier/1`, `list/1`, `list/2`, `nlist/2`, `sequence/2`, `sequence_or_list/2`, `character_code/1`, `string/1`, `num_code/1`, `predname/1`, `atm_or_atm_list/1`, `flag_values/1`.
- **Imports:**
  - *System library modules:*
    `assertions/native_props`, `terms_check`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`, `assertions`, `nortchecks`, `nativeprops`.

9.2 Documentation on exports (basic_props)

term/1: REGTYPE

The most general type (includes all possible terms).

(True) Usage: `term(X)`

`X` is any term.

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP. (`native/1`)

General properties:

True: `term(X)`

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

True: `term(X)`

– *The following properties hold globally:*

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

True: `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.

(True) Usage: `int(T)`

`T` is an integer.

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`native/1`)

General properties:

True: `int(T)`

– *The following properties hold globally:*

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

True: `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`)

Trust: `int(T)`

– *The following properties hold upon exit:*

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

Trust:

– *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.

(True) Usage: `nnegint(T)`

`T` is a non-negative integer.

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`native/1`)

General properties:

True: `nnegint(T)`

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

True: `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`)

Trust: `nnegint(T)`

– *The following properties hold upon exit:*
`T` is a non-negative integer. (`nnegint/1`)

Trust:

– *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`

(True) Usage: `flt(T)`

`T` is a float.

– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (`native/1`)

General properties:

True: `flt(T)`

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

True: `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`)

Trust: `flt(T)`

– *The following properties hold upon exit:*
`T` is a float. (`flt/1`)

Trust:

– *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.

(True) Usage: `num(T)`

`T` is a number.

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`native/1`)

General properties:

True: `num(T)`

– *The following properties hold globally:*

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

`num(T)` is binding insensitive. (`bind_ins/1`)

True: `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`)

Trust: `num(T)`

– *The following properties hold upon exit:*

`T` is a number. (`num/1`)

Trust:

– *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.

(True) Usage: `atm(T)`

`T` is an atom.

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`native/1`)

General properties:

True: `atm(T)`

– *The following properties hold globally:*

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

True: `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`)

Trust: `atm(T)`

– *The following properties hold upon exit:*
`T` is an atom. (`atm/1`)

Trust:

– *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.

(True) Usage: `struct(T)`

`T` is a compound term.

– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (`native/1`)

General properties:

True: `struct(T)`

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

True: `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`)

Trust: `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.

(True) Usage: `gnd(T)`

`T` is ground.

– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (`native/1`)

General properties:

True: `gnd(T)`

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

True: `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`)

Trust: `gnd(T)`

– *The following properties hold upon exit:*
`T` is ground. (`gnd/1`)

Trust:

– *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

(True) Usage: `gndstr(T)`

`T` is a ground compound term.

– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (`native/1`)

General properties:

True: `gndstr(T)`

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

True: `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`)

Trust: `gndstr(T)`

– *The following properties hold upon exit:*
`T` is a ground compound term. (`gndstr/1`)

constant/1: REGTYPE

(True) Usage: `constant(T)`

`T` is an atomic term (an atom or a number).

General properties:

True: `constant(T)`

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

True: `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`)

Trust: `constant(T)`

– *The following properties hold upon exit:*
`T` is an atomic term (an atom or a number). (`constant/1`)

callable/1: REGTYPE

(True) Usage: `callable(T)`

`T` is a term which represents a goal, i.e., an atom or a structure.

General properties:

True: `callable(T)`

– *The following properties hold globally:*

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

True: `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`)

Trust: `callable(T)`

– *The following properties hold upon exit:*

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

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.

(True) Usage: `operator_specifier(X)`

`X` specifies the type and associativity of an operator.

General properties:

True: `operator_specifier(X)`

– *The following properties hold globally:*

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

True: `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`)

Trust: `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).
```

(True) Usage: `list(L)`
`L` is a list.

General properties:
True: `list(L)`
– *The following properties hold globally:*
`list(L)` is side-effect `free`. (`sideff/2`)

True: `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`)

Trust: `list(T)`
– *The following properties hold upon exit:*
`T` is a list. (`list/1`)

list/2: REGTYPE

`list(L,T)`
`L` is a list, and for all its elements, `T` holds.
(True) Usage: `list(L,T)`
`L` is a list of `T`s.

Meta-predicate with arguments: `list(?,(pred 1))`.
General properties:
True: `list(L,T)`
– *The following properties hold globally:*
`list(L,T)` is side-effect `free`. (`sideff/2`)

True: `list(L,T)`

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

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

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

then the following properties hold globally:

`list(L,T)` is evaluable at compile-time. (`eval/1`)

Trust: `list(X,T)`

– *The following properties hold upon exit:*

`X` is a list. (`list/1`)

nlist/2: REGTYPE

(True) Usage: `nlist(L,T)`

`L` is `T` or a nested list of `T`s. Note that if `T` is term, this type is equivalent to term, this fact explain why we do not have a `nlist/1` type

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

General properties:

True: `nlist(L,T)`

– *The following properties hold globally:*

`nlist(L,T)` is side-effect `free`. (`sideff/2`)

True: `nlist(L,T)`

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

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

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

then the following properties hold globally:

`nlist(L,T)` is evaluable at compile-time. (`eval/1`)

Trust: `nlist(X,T)`

– *The following properties hold upon exit:*

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

member/2: PROPERTY

(True) Usage: `member(X,L)`

`X` is an element of `L`.

General properties:

True: `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`)

True: `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`)

Trust: `member(_X,L)`

– *The following properties hold upon exit:*

`L` is a list. (`list/1`)

Trust: `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.

(True) Usage: `sequence(S,T)`

`S` is a sequence of `T`s.

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

General properties:

True: `sequence(S,T)`

– *The following properties hold globally:*

`sequence(S,T)` is side-effect `free`. (`sideff/2`)

True: `sequence(S,T)`

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

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

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

then the following properties hold globally:

`sequence(S,T)` is evaluable at compile-time. (`eval/1`)

Trust: `sequence(E,T)`

– *The following properties hold upon exit:*

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

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

sequence_or_list/2: REGTYPE

(True) Usage: `sequence_or_list(S,T)`

`S` is a sequence or list of `T`s.

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

General properties:

True: `sequence_or_list(S,T)`

– *The following properties hold globally:*

`sequence_or_list(S,T)` is side-effect `free`. (`sideff/2`)

True: `sequence_or_list(S,T)`

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

Trust: `sequence_or_list(E,T)`

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

character_code/1: REGTYPE

(True) Usage: `character_code(T)`

`T` is an integer which is a character code.

General properties:

True: `character_code(T)`

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

True: `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`)

Trust: `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]`.

(True) Usage: `string(T)`

`T` is a string (a list of character codes).

General properties:

True: `string(T)`

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

True: `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`)

Trust: `string(T)`

– *The following properties hold upon exit:*

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

num_code/1: REGTYPE

These are the ASCII codes which can appear in decimal representation of floating point and integer numbers, including scientific notation and fractionary part.

predname/1: REGTYPE

(True) Usage: `predname(P)`

`P` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        nnegint(A).
```

General properties:

True: `predname(P)`

– *The following properties hold globally:*

`predname(P)` is side-effect `free`. (`sideff/2`)

True: `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`)

Trust: `predname(P)`

– *The following properties hold upon exit:*

`P` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        nnegint(A).
```

(`predname/1`)

atm_or_atm_list/1: REGTYPE

(True) Usage: `atm_or_atm_list(T)`

`T` is an atom or a list of atoms.

General properties:

True: `atm_or_atm_list(T)`

– *The following properties hold globally:*

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

True: `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`)

Trust: `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.

(True) Usage: `compat(Term,Prop)`

`Term` is *compatible* with `Prop`

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

General properties:

True: `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

(True) Usage: `inst(Term,Prop)`

`Term` is instantiated enough to satisfy `Prop`.

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

General properties:

True: `inst(Term,Prop)`
– *The following properties hold globally:*
`inst(Term,Prop)` is side-effect `free`. (`sideff/2`)

True: `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

(True) Usage: `iso(G)`

Complies with the ISO-Prolog standard.

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

General properties:

True: `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.

(True) Usage: `deprecated(G)`

DEPRECATED.

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

General properties:

True: `deprecated(G)`

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

not_further_inst/2: PROPERTY

(True) Usage: `not_further_inst(G,V)`

`V` is not further instantiated.

Meta-predicate with arguments: `not_further_inst(goal,?)`.

General properties:

True: `not_further_inst(G,V)`

– *The following properties hold globally:*
`not_further_inst(G,V)` is side-effect `free`. (`sideff/2`)
Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`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).

(True) Usage: `sideff(G,X)`

`G` is side-effect `X`.

– *If the following properties hold at call time:*
`G` is a term which represents a goal, i.e., an atom or a structure. (`callable/1`)
`X` is an element of `[free,soft,hard]`. (`member/2`)

Meta-predicate with arguments: `sideff(goal,?)`.

General properties:

True: `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`)

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

regtype/1: PROPERTY

(True) Usage: `regtype G`

Defines a regular type.

Meta-predicate with arguments: `regtype goal`.

General properties:

True: `regtype G`

– *The following properties hold globally:*

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

native/1: PROPERTY

(True) Usage: `native(Pred)`

This predicate is understood natively by CiaoPP.

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

General properties:

True: `native(P)`

– *The following properties hold globally:*

`native(P)` is side-effect `free`. (`sideff/2`)

native/2: PROPERTY

(True) Usage: `native(Pred,Key)`

This predicate is understood natively by CiaoPP as `Key`.

Meta-predicate with arguments: `native(goal,?)`.

General properties:

True: `native(P,K)`

– *The following properties hold globally:*

`native(P,K)` is side-effect `free`. (`sideff/2`)

rtcheck/1: PROPERTY

(True) Usage: `rtcheck(G)`

Equivalent to rtcheck(G, complete).

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

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

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

General properties:

True: `rtcheck(G)`

– *The following properties hold globally:*

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

rtcheck/2: PROPERTY

(True) Usage: `rtcheck(G,Status)`

The runtime check of the property have the status `Status`.

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

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

Status of the runtime-check implementation for a given property. Valid values are:

- unimplemented: No run-time checker has been implemented for the property. Althought it can be implemented further.
- incomplete: The current run-time checker is incomplete, which means, under certain circunstances, no error is reported if the property is violated.
- unknown: We do not know if current implementation of run-time checker is complete or not.
- complete: The opposite of incomplete, error is reported always that the property is violated. Default.
- impossible: The property must not be run-time checked (for theoretical or practical reasons).

(`rtc_status/1`)

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

General properties:

True: `rtcheck(G,Status)`

– *The following properties hold globally:*

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

no_rtcheck/1: PROPERTY

(True) Usage: `no_rtcheck(G)`

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible).

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

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

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

General properties:

True: `no_rtcheck(G)`

– *The following properties hold globally:*

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

eval/1: PROPERTY

(True) Usage: `eval(Goal)`

`Goal` is evaluable at compile-time.

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

equiv/2: PROPERTY

(True) Usage: `equiv(Goal1,Goal2)`

`Goal1` is equivalent to `Goal2`.

Meta-predicate with arguments: `equiv(goal,goal)`.

bind_ins/1: PROPERTY

(True) Usage: `bind_ins(Goal)`

`Goal` is binding insensitive.

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

error_free/1: PROPERTY

(True) Usage: `error_free(Goal)`

`Goal` is error free.

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

memo/1: PROPERTY

(True) Usage: `memo(Goal)`

`Goal` should be memoized (not unfolded).

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

filter/2: PROPERTY

(True) Usage: `filter(Vars,Goal)`

`Vars` should be filtered during global control).

flag_values/1: REGTYPE

(True) Usage: `flag_values(X)`

Define the valid flag values

pe_type/1: PROPERTY

(True) 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)`.

9.3 Known bugs and planned improvements (basic_props)

- Run-time checks have been reported not to work with this code. That means that either the assertions here, or the code that implements the run-time checks are erroneous.

10 Properties which are native to analyzers

Author(s): Francisco Bueno, Manuel Hermenegildo, Pedro López, Edison Mera.

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.

10.1 Usage and interface (`native_props`)

- **Library usage:**
  `:- use_module(library(assertions(native_props)))`
  or also as a package `:- use_package(nativeprops).`
  Note the different names of the library and the package.
- **Exports:**
  - *Properties:*
    `clique/1`, `clique_1/1`, `compat/1`, `constraint/1`, `covered/1`, `covered/2`, `exception/1`, `exception/2`, `fails/1`, `finite_solutions/1`, `have_choicepoints/1`, `indep/1`, `indep/2`, `instance/1`, `is_det/1`, `linear/1`, `mshare/1`, `mut_exclusive/1`, `no_choicepoints/1`, `no_exception/1`, `no_exception/2`, `no_signal/1`, `no_signal/2`, `non_det/1`, `nonground/1`, `not_covered/1`, `not_fails/1`, `not_mut_exclusive/1`, `num_solutions/2`, `solutions/2`, `possibly_fails/1`, `possibly_nondet/1`, `relations/2`, `sideff_hard/1`, `sideff_pure/1`, `sideff_soft/1`, `signal/1`, `signal/2`, `signals/2`, `size/2`, `size/3`, `size_lb/2`, `size_o/2`, `size_ub/2`, `size_metric/3`, `size_metric/4`, `succeeds/1`, `steps/2`, `steps_lb/2`, `steps_o/2`, `steps_ub/2`, `tau/1`, `terminates/1`, `test_type/2`, `throws/2`, `user_output/2`.
- **Imports:**
  - *System library modules:*
    `terms_check`, `terms_vars`, `hiordlib`, `sort`, `lists`, `streams`, `file_utils`, `system`, `odd`, `rtchecks/rtchecks_send`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`, `internals`.
  - *Packages:*
    `prelude`, `nonpure`, `assertions`, `hiord`.

10.2 Documentation on exports (`native_props`)

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 pattern is `X`.

– *The following properties should hold globally:*

This predicate is understood natively by CiaoPP as `clique(X)`. (`native/2`)

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`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 pattern is `X`.

– *The following properties should hold globally:*

This predicate is understood natively by CiaoPP as `clique_1(X)`. (`native/2`)

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

compat/1: PROPERTY

Usage: `compat(Prop)`

Use `Prop` as a compatibility property.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

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

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.

(True) Usage: `constraint(C)`

`C` is a list of linear equations

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`native/1`)

covered/1: PROPERTY

`covered(X)`

For any call of the form `X` there is at least one clause whose test succeeds (i.e., all the calls of the form `X` are covered) [DLGH97].

Usage: `covered(X)`

All the calls of the form `X` are covered.

– *The following properties should hold globally:*

The runtime check of the property have the status `unimplemented`. (`rtcheck/2`)

covered/2: PROPERTY

`covered(X,Y)`

All variables occuring in `X` occur also in `Y`.

(True) Usage: `covered(X,Y)`

`X` is covered by `Y`.

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`native/1`)

exception/1: PROPERTY

Usage: `exception(Goal)`

Calls of the form `Goal` throw an exception.

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

exception/2: PROPERTY

Usage: `exception(Goal,E)`

Calls of the form `Goal` throw an exception that unifies with `E`.

Meta-predicate with arguments: `exception(goal,?)`.

fails/1: PROPERTY

`fails(X)`

Calls of the form `X` fail.

(True) Usage: `fails(X)`

Calls of the form `X` fail.

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`native/1`)

Meta-predicate with arguments: `fails(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:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

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

have_choicepoints/1: PROPERTY

Usage: `have_choicepoints(X)`

A call to `X` creates choicepoints.

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

indep/1: PROPERTY

(True) Usage: `indep(X)`

The variables in pairs in `X` are pairwise independent.

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP as `indep(X)`. (`native/2`)

indep/2: PROPERTY

(True) Usage: `indep(X,Y)`

`X` and `Y` do not have variables in common.

- *The following properties hold globally:*
  This predicate is understood natively by CiaoPP as `indep([[X,Y]])`. (`native/2`)

instance/1: PROPERTY

Usage: `instance(Prop)`

Use Prop as an instantiation property. Verify that execution of `Prop` does not produce bindings for the argument variables.

- *The following properties should hold globally:*
  Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

Meta-predicate with arguments: `instance(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.

Usage: `is_det(X)`

All calls of the form `X` are deterministic.

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

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.

(True) Usage: `linear(X)`

`X` is instantiated to a linear term.

– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (`native/1`)

mshare/1: PROPERTY

`mshare(X)`

`X` contains all *sharing sets* [JL88,MH89b] 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 is `X`.

– *The following properties should hold globally:*
This predicate is understood natively by CiaoPP as `sharing(X)`. (`native/2`)
Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

General properties:

Test: `mshare(L)`

– *If the following properties should hold at call time:*
`L=[[A],[p(A)]]` (`=/2`)
then the following properties should hold globally:
Calls of the form `mshare(L)` fail. (`fails/1`)

Test: `mshare(L)`

– *If the following properties should hold at call time:*
`L=[[A],[p(B)]]` (`=/2`)
then the following properties should hold globally:
All the calls of the form `mshare(L)` do not fail. (`not_fails/1`)

mut_exclusive/1: PROPERTY

`mut_exclusive(X)`

For any call of the form `X` at most one clause succeeds, i.e., clauses are pairwise exclusive.

Usage: `mut_exclusive(X)`

For any call of the form `X` at most one clause succeeds.

– *The following properties should hold globally:*
The runtime check of the property have the status `unimplemented`. (`rtcheck/2`)

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

no_choicepoints/1: PROPERTY

Usage: `no_choicepoints(X)`

A call to `X` does not create choicepoints.

Meta-predicate with arguments: `no_choicepoints(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 exception `E`.

Meta-predicate with arguments: `no_exception(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 the signal `E`.

Meta-predicate with arguments: `no_signal(goal,?)`.

non_det/1: PROPERTY

`non_det(X)`

All calls of the form `X` are non-deterministic, i.e., produce several solutions.

Usage: `non_det(X)`

All calls of the form `X` are non-deterministic.

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

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`)

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 the property have the status `unimplemented`. (`rtcheck/2`)

not_fails/1: PROPERTY

`not_fails(X)`

Calls of the form `X` produce at least one solution, or do not terminate [DLGH97].

(True) Usage: `not_fails(X)`

All the calls of the form `X` do not fail.

– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (`native/1`)

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

not_mut_exclusive/1: PROPERTY

`not_mut_exclusive(X)`

For calls of the form `X` more than one clause may succeed. I.e., clauses are not disjoint for some call.

Usage: `not_mut_exclusive(X)`

For some calls of the form `X` more than one clause may succeed.

– *The following properties should hold globally:*
The runtime check of the property have the status `unimplemented`. (`rtcheck/2`)

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

num_solutions/2: PROPERTY

Usage 1: `num_solutions(X,N)`

All the calls of the form `X` have `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. (`callable/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. (`callable/1`)
`Check` is a term which represents a goal, i.e., an atom or a structure. (`callable/1`)

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. (`callable/1`)
`Sols` is a list. (`list/1`)

possibly_fails/1: PROPERTY

`possibly_fails(X)`

Non-failure is not ensured for any call of the form `X` [DLGH97]. 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:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

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

possibly_nondet/1: PROPERTY

`possibly_nondet(X)`

Non-determinism is not ensured for all calls of the form `X`. In other words, nothing can be ensured about determinacy nor termination of such calls.

Usage: `possibly_nondet(X)`

Non-determinism is not ensured for calls of the form `X`.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

relations/2: PROPERTY

`relations(X,N)`

The goal `X` produces `N` solutions. In other words, `N` is the cardinality of the solution set of `X`.

Usage: `relations(X,N)`

Goal `X` produces `N` solutions.

– *The following properties should hold globally:*

The runtime check of the property have the status `unimplemented`. (`rtcheck/2`)

Meta-predicate with arguments: `relations(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:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

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

sideff_pure/1: PROPERTY

Usage: `sideff_pure(X)`

`X` is pure, i.e., has no side-effects.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`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:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

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

signal/1: PROPERTY

Usage: `signal(Goal)`

Calls of the form `Goal` throw a signal.

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

signal/2: PROPERTY

Usage: `signal(Goal,E)`

A call to `Goal` sends a signal that unifies with `E`.

Meta-predicate with arguments: `signal(goal,?)`.

signals/2: PROPERTY

Usage: `signals(Goal,Es)`

Calls of the form `Goal` can generate only the signals that unify with the terms listed in `Es`.

– *The following properties should hold globally:*

The runtime check of the property have the status `unimplemented`. (`rtcheck/2`)

Meta-predicate with arguments: `signals(goal,?)`.

size/2: PROPERTY

Usage: `size(X,Y)`

`Y` is the size of argument `X`, for any approximation.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

size/3: PROPERTY

Usage: `size(A,X,Y)`

`Y` is the size of argument `X`, for the approximation `A`.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

size_lb/2: PROPERTY

`size_lb(X,Y)`

The minimum size of the terms to which the argument `Y` 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,LGHD96b].

Usage: `size_lb(X,Y)`

`Y` is a lower bound on the size of argument `X`.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

size_o/2: PROPERTY

Usage: `size_o(X,Y)`

The size of argument `X` is in the order of `Y`.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

size_ub/2: PROPERTY

`size_ub(X,Y)`

The maximum size of the terms to which the argument `Y` 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,LGHD96b].

Usage: `size_ub(X,Y)`

`Y` is a upper bound on the size of argument `X`.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

size_metric/3: PROPERTY

Usage: `size_metric(Head,Var,Metric)`

`Metric` is the metric of the variable `Var`, for any approximation.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

Meta-predicate with arguments: `size_metric(goal,?,?)`.

size_metric/4: PROPERTY

Usage: `size_metric(Head,Approx,Var,Metric)`

`Metric` is the metric of the variable `Var`, for the approximation `Approx`. Currently, `Metric` can be: `int/1`, `size/1`, `length/1`, `depth/2`, and `void/1`.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

Meta-predicate with arguments: `size_metric(goal,?,?,?)`.

succeeds/1: PROPERTY

Usage: `succeeds(Prop)`

A call to `Prop` succeeds.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

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

steps/2: PROPERTY

`steps(X,Y)`

The time (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`.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

Meta-predicate with arguments: `steps(goal,?)`.

steps_lb/2: PROPERTY

`steps_lb(X,Y)`

The minimum computation time (in resolution steps) spent by any call of the form `X` is given by the expression `Y` [DLGHL97,LGHD96b]

Usage: `steps_lb(X,Y)`

`Y` is a lower bound on the cost of any call of the form `X`.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

Meta-predicate with arguments: `steps_lb(goal,?)`.

steps_o/2: PROPERTY

Usage: `steps_o(X,Y)`

`Y` is the complexity order of the cost of any call of the form `X`.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

Meta-predicate with arguments: `steps_o(goal,?)`.

steps_ub/2: PROPERTY

`steps_ub(X,Y)`

The maximum computation time (in resolution steps) spent by any call of the form `X` is given by the expression `Y` [DL93,LGHD96b].

Usage: `steps_ub(X,Y)`

`Y` is a upper bound on the cost of any call of the form `X`.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

Meta-predicate with arguments: `steps_ub(goal,?)`.

tau/1: PROPERTY

`tau(Types)`

`Types` contains a list with the type associations for each variable, in the form `V/[T1,..,TN]`. Note that tau is used in object-oriented programs only

(True) Usage: `tau(TypeInfo)`

`Types` is a list of associations between variables and list of types

– *The following properties hold globally:*

This predicate is understood natively by CiaoPP. (`native/1`)

terminates/1: PROPERTY

`terminates(X)`

Calls of the form `X` always terminate [DLGH97].

Usage: `terminates(X)`

All calls of the form `X` terminate.

– *The following properties should hold globally:*

Declares that the assertion in which this comp property appears must not be checked at run-time. Equivalent to rtcheck(G, impossible). (`no_rtcheck/1`)

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

test_type/2: PROPERTY

Usage: `test_type(X,T)`

Indicates the type of test that a predicate performs. Required by the nonfailure analyisis.

Meta-predicate with arguments: `test_type(goal,?)`.

throws/2: PROPERTY

Usage: `throws(Goal,Es)`

Calls of the form `Goal` can throw only the exceptions that unify with the terms listed in `Es`.

– *The following properties should hold globally:*
The runtime check of the property have the status `unimplemented`. (`rtcheck/2`)

Meta-predicate with arguments: `throws(goal,?)`.

user_output/2: PROPERTY

Usage: `user_output(Goal,S)`

Calls of the form `Goal` write `S` to standard output.

Meta-predicate with arguments: `user_output(goal,?)`.

instance/2: PROPERTY

(True) Usage: `instance(Term1,Term2)`

`Term1` is an instance of `Term2`.

– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (`native/1`)

10.3 Known bugs and planned improvements (native_props)

- A missing property is succeeds (not_fails = succeeds or not_terminates. – EMM

11 Run-time checking of assertions

Author(s): Edison Mera.

This package provides a complete implementation of run-time checks of predicate assertions. The program is instrumented to check such assertions at run time, and in case a property does not hold, the error is reported. Note that there is also an older package called rtchecks, by David Trallero. The advantage of this one is that it can be used independently of CiaoPP and also has updated functionality.

There are two main applications of run-time checks:

- To improve debugging of certain predicates, specifying some expected behavior that is checked at run-time with the assertions.
- To avoid manual implementation of run-time checks that should be done in some predicates, leaving the code clean and understandable.

The run-time checks can be configured using prolog flags. Below we itemize the valid prolog flags with its values and a brief explanation of the meaning:

- `rtchecks_level`
  - `exports`: Only use rtchecks for external calls of the exported predicates.
  - `inner` : Use also rtchecks for internal calls. Default.
- `rtchecks_trust`
  - `no` : Disable rtchecks for trust assertions.
  - `yes` : Enable rtchecks for trust assertions. Default.
- `rtchecks_entry`
  - `no` : Disable rtchecks for entry assertions.
  - `yes` : Enable rtchecks for entry assertions. Default.
- `rtchecks_exit`
  - `no` : Disable rtchecks for exit assertions.
  - `yes` : Enable rtchecks for exit assertions. Default.
- `rtchecks_test`
  - `no` : Disable rtchecks for test assertions. Default.
  - `yes` : Enable rtchecks for test assertions. Used for debugging purposes, but is better to use the unittest library.
- `rtchecks_inline`
  - `no` : Instrument rtchecks using call to library predicates present in `rtchecks_rt.pl`, `nativeprops.pl` and `basic_props.pl`. In this way, space is saved, but sacrifying performance due to usage of meta calls and external methods in the libraries. Default.
  - `yes` : Expand library predicates inline as far as possible. In this way, the code is faster, because its avoids metacalls and usage of external methods, but the final executable could be bigger.
- `rtchecks_asrloc` Controls the usage of locators for the assertions in the error messages. The locator says the file and lines that contains the assertion that had failed. Valid values are:
  - `no` : Disabled.
  - `yes` : Enabled. Default.
- `rtchecks_predloc` Controls the usage of locators for the predicate that caused the run-time check error. The locator says the first clause of the predicate that the violated assertion refers to.
  - `no` : Disabled.

  - `yes` : Enabled, Default.
- `rtchecks_callloc`
  - `no` : Do not show the stack of predicates that caused the failure
  - `predicate`: Show the stack of predicates that caused the failure. Instrument it in the predicate. Default.
  - `literal` : Show the stack of predicates that caused the failure. Instrument it in the literal. This mode provides more information, because reports also the literal in the body of the predicate.
- `rtchecks_namefmt`
  - `long` : Show the name of predicates, properties and the values of the variables
  - `short` : Only show the name of the predicate in a reduced format. Default.
- `rtchecks_abort_on_error`

  Controls if run time checks must abort the execution of a program (by raising an exception), or if the execution of the program have to continue.

  Note that this option only affect the default handler and the predicate `call_rtc/1`, so if you use your own handler it will not have effect.
  - `yes` : Raising a run time error will abort the program.
  - `no` : Raising a run time error will not stop the execution, but a message will be shown. Default.

11.1 Usage and interface (`rtchecks_doc`)

- **Library usage:**

  `:- use_package(rtchecks).`

  or

  `:- module(...,...,[rtchecks]).`
- **Imports:**
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*

    `prelude`, `nonpure`, `assertions`.

PART III - Extending CiaoPP

Author(s): The CLIP Group.

12 Adding a new analysis domain to CiaoPP

Author(s): The CLIP Group.

One of the most relevant features of the CiaoPP system is that it allows the addition of new analysis domains to the abstract interpretation-based framework in an easy way.

The next Chapter (Chapter 13 [Plug-in points for abstract domains], page 107) describes the module `domains.pl`, the general interface used by the analyzer for accessing the operations of the different domains implemented in the system. The developer of a new domain must edit this module, adding the necessary clauses to link the general interface with the specific operations of the new domain.

The procedure of adding a new domain is illustrated in Chapter 14 [Simple groundness abstract domain], page 117 by means of a simple abstract domain for groundness information inference. It includes a list of the predicates that need to be implemented by the developer in order to add the abstract domain to CiaoPP.

13 Plug-in points for abstract domains

Author(s): Maria Garcia de la Banda, Francisco Bueno.

This module contains the predicates for selecting the abstract operations that correspond to an analysis domain. The selection depends on the name of the domain given as first argument to all predicates. Whenever a new domain is added to the system, a new clause for each predicate exported here will be needed to call the corresponding domain operation in the domain module. Some local operations used but not exported by this module would have to be defined, too. See the following chapter for an example domain module.

Adding an analysis domain to PLAI requires only changes in this module. However, in order for other CiaoPP operations to work, you may need to change other modules. See, for example, module `infer_dom`.

In this chapter, arguments referred to as `Sv`, `Hv`, `Fv`, `Qv`, `Vars` are lists of program variables and are supposed to always be sorted. Abstract substitutions are referred to as `ASub`, and are also supposed sorted (except where indicated), although this depends on the domain.

13.1 Usage and interface (domains)

- **Library usage:**

  `:- use_module(library(domains)).`
- **Exports:**
  - *Predicates:*

    `init_abstract_domain/2`, `amgu/5`, `call_to_entry/9`, `exit_to_prime/8`, `project/5`, `extend/5`, `widen/4`, `widencall/4`, `normalize_asub/3`, `compute_lub/3`, `glb/4`, `less_or_equal/3`, `less_or_equal_proj/5`, `identical_abstract/3`, `identical_proj/5`, `identical_proj_1/7`, `abs_sort/3`, `augment_asub/4`, `augment_two_asub/4`, `abs_subset/3`, `eliminate_equivalent/3`, `call_to_success_fact/9`, `body_succ_builtin/9`, `special_builtin/6`, `concrete/4`, `part_conc/5`, `multi_part_conc/4`, `obtain_info/5`, `info_to_asub/5`, `full_info_to_asub/4`, `asub_to_info/5`, `asub_to_native/5`, `unknown_call/5`, `unknown_call/4`, `unknown_entry/4`, `unknown_entry/3`, `empty_entry/3`, `collect_types_in_abs/4`, `rename_types_in_abs/4`, `dom_statistics/2`, `abstract_instance/5`, `contains_parameters/2`.
  - *Multifiles:*

    `aidomain/1`.
- **Imports:**
  - *Application modules:*

    `program(p_unit)`, `ciaopp(preprocess_flags)`, `plai(plai_errors)`, `plai(fixpo_ops)`, `domain(pd)`, `domain(pdb)`, `domain(gr)`, `domain(java_nullity)`, `domain(def)`, `domain(fd)`, `domain(fr_top)`, `domain(lsign)`, `domain(share)`, `domain(shfret)`, `domain(shareson)`, `domain(shfrson)`, `domain(sondergaard)`, `domain(oo_son)`, `domain(oo_shnltau)`, `domain(share_amgu)`, `domain(share_clique)`, `domain(bshare(bshare))`, `domain(aeq_top)`, `domain(depthk)`, `domain(top_path_sharing)`, `domain(eterms)`, `domain(svterms)`, `domain(termsd)`, `domain(ptypes)`, `domain(polyhedra)`, `domain(oo_types)`, `domain(deftypes)`, `domain(java_cha)`, `domain(nfplai)`, `domain(detplai)`, `infer(low_level_props)`.
  - *System library modules:*

    `terms_check`, `terms_vars`, `sets`, `sort`, `messages`, `assertions/native_props`.
  - *Internal (engine) modules:*

    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*

    `prelude`, `nonpure`, `assertions`, `regtypes`.

13.2 Documentation on exports (domains)

init_abstract_domain/2: PREDICATE

`init_abstract_domain(AbsInt,Norm)`

Initializes abstract domain `AbsInt`. Tells whether `AbsInt` requires a normalized program.

amgu/5: PREDICATE

`amgu(AbsInt,Sg,Head,ASub,AMGU)`

Perform the abstract unification `AMGU` between `Sg` and `Head` given an initial abstract substitution `ASub` and abstract domain `AbsInt`.

call_to_entry/9: PREDICATE

`call_to_entry(AbsInt,Sv,Sg,Hv,Head,Fv,Proj,Entry,ExtraInfo)`

Obtains the abstract substitution `Entry` which results from adding the abstraction of the unification `Sg` = `Head` to abstract substitution `Proj` (the call substitution for `Sg` projected on its variables `Sv`) and then projecting the resulting substitution onto `Hv` (the variables of `Head`) plus `Fv` (the free variables of the relevant clause). `ExtraInfo` is information which may be reused later in other abstract operations.

exit_to_prime/8: PREDICATE

`exit_to_prime(AbsInt,Sg,Hv,Head,Sv,Exit,ExtraInfo,Prime)`

Computes the abstract substitution `Prime` which results from adding the abstraction of the unification `Sg` = `Head` to abstract substitution `Exit` (the exit substitution for a clause `Head` projected over its variables `Hv`), projecting the resulting substitution onto `Sv`.

project/5: PREDICATE

`project(AbsInt,Vars,HvFv_u,ASub,Proj)`

Projects the abstract substitution `ASub` onto the variables of list `Vars` resulting in the projected abstract substitution `Proj`.

extend/5: PREDICATE

`extend(AbsInt,Prime,Sv,Call,Succ)`

`Succ` is the extension the information given by `Prime` (success abstract substitution over the goal variables `Sv`) to the rest of the variables of the clause in which the goal occurs (those over which abstract substitution `Call` is defined on). I.e., it is like a conjunction of the information in `Prime` and `Call`, except that they are defined over different sets of variables, and that `Prime` is a successor substitution to `Call` in the execution of the program.

widen/4: PREDICATE

`widen(AbsInt,ASub0,ASub1,ASub)`

`ASub` is the result of widening abstract substitution `ASub0` and `ASub1`, which are supposed to be consecutive approximations to the same abstract value.

widencall/4: PREDICATE

`widencall(AbsInt,ASub0,ASub1,ASub)`

`ASub` is the result of widening abstract substitution `ASub0` and `ASub1`, which are supposed to be consecutive call patterns in a fixpoint computation.

normalize_asub/3: PREDICATE

`normalize_asub(AbsInt,ASub0,ASub1)`

`ASub1` is the result of normalizing abstract substitution `ASub0`. This is required in some domains, specially to perform the widening.

compute_lub/3: PREDICATE

`compute_lub(AbsInt,ListASub,LubASub)`

`LubASub` is the least upper bound of the abstract substitutions in list `ListASub`.

glb/4: PREDICATE

`glb(AbsInt,ASub0,ASub1,GlbASub)`

`GlbASub` is the greatest lower bound of abstract substitutions `ASub0` and `ASub1`.

less_or_equal/3: PREDICATE

`less_or_equal(AbsInt,ASub0,ASub1)`

Succeeds if `ASub1` is more general or equivalent to `ASub0`.

less_or_equal_proj/5: PREDICATE

`less_or_equal_proj(AbsInt,Sg,Proj,Sg1,Proj1)`

Abstract pattern `Sg`:`Proj` is less general or equivalent to abstract pattern `Sg1`:`Proj1` in domain `AbsInt`.

identical_abstract/3: PREDICATE

`identical_abstract(AbsInt,ASub1,ASub2)`

Succeeds if, in the particular abstract domain, the two abstract substitutions `ASub1` and `ASub2` are defined on the same variables and are equivalent.

identical_proj/5: PREDICATE

`identical_proj(AbsInt,Sg,Proj,Sg1,Proj1)`

Abstract patterns `Sg`:`Proj` and `Sg1`:`Proj1` are equivalent in domain `AbsInt`. Note that `Proj` is assumed to be already sorted.

identical_proj_1/7: PREDICATE

`identical_proj_1(AbsInt,Sg,Proj,Sg1,Proj1,Prime1,Prime2)`

Abstract patterns `Sg`:`Proj` and `Sg1`:`Proj1` are equivalent in domain `AbsInt`. Note that `Proj` is assumed to be already sorted. It is different from `identical_proj/5` because it can be true although `Sg` and `Sg1` are not variant

abs_sort/3: PREDICATE

`abs_sort(AbsInt,ASub_u,ASub)`

`ASub` is the result of sorting abstract substitution `ASub_u`.

augment_asub/4: PREDICATE

`augment_asub(AbsInt,ASub,Vars,ASub0)`

Augment the abstract substitution `ASub` adding the variables `Vars` and then resulting the abstract substitution `ASub0`.

augment_two_asub/4: PREDICATE

`augment_two_asub(AbsInt,ASub0,ASub1,ASub)`

`ASub` is an abstract substitution resulting of augmenting two abstract substitutions: `ASub0` and `ASub1` whose domains are disjoint.

abs_subset/3: PREDICATE

`abs_subset(AbsInt,LASub1,LASub2)`

Succeeds if each abstract substitution in list `LASub1` is equivalent to some abstract substitution in list `LASub2`.

eliminate_equivalent/3: PREDICATE

`eliminate_equivalent(AbsInt,TmpLSucc,LSucc)`

The list `LSucc` is reduced wrt the list `TmpLSucc` in that it does not contain abstract substitutions which are equivalent.

call_to_success_fact/9: PREDICATE

`call_to_success_fact(AbsInt,Sg,Hv,Head,Sv,Call,Proj,Prime,Succ)`

Specialized version of call_to_entry + entry_to_exit + exit_to_prime for a fact `Head`.

body_succ_builtin/9: PREDICATE

`body_succ_builtin(Type,AbsInt,Sg,Vs,Sv,Hv,Call,Proj,Succ)`

Specialized version of call_to_entry + entry_to_exit + exit_to_prime + extend for predicate `Sg` considered a "builtin" of type `Type` in domain `AbsInt`. Whether a predicate is "builtin" in a domain is determined by `special_builtin/5`. There are two different ways to treat these predicates, depending on `Type`: `success_builtin` handles more usual types of "builtins", `call_to_success_builtin` handles particular predicates. The later is called when `Type` is of the form `special(SgKey)`.

special_builtin/6: PREDICATE

`special_builtin(AbsInt,SgKey,Sg,Subgoal,Type,Condvars)`

Predicate `Sg` is considered a "builtin" of type `Type` in domain `AbsInt`. Types are domain dependent. Domains may have two different ways to treat these predicates: see `body_succ_builtin/9`.

concrete/4: PREDICATE

`concrete(AbsInt,Var,ASub,List)`

`List` are (all) the terms to which `Var` can be bound in the concretization of `ASub`, if they are a finite number of finite terms. Otherwise, the predicate fails.

part_conc/5: PREDICATE

`part_conc(AbsInt,Sg,Subs,NSg,NSubs)`

This operation returns in `NSg` an instance of `Sg` in which the deterministic structure information available in `Subs` is materialized. The substitution `NSubs` refers to the variables in `NSg`.

multi_part_conc/4: PREDICATE

`multi_part_conc(AbsInt,Sg,Subs,List)`

Similar to part_conc but it gives instantiations of goals even in the case types are not deterministic, it generates a `List` of pairs of goals and substitutions. It stops unfolding types as soon as they are recursive.

obtain_info/5: PREDICATE

`obtain_info(AbsInt,Prop,Vars,ASub,Info)`

Obtains variables `Info` for which property `Prop` holds given abstract substitution `ASub` on variables `Vars` for domain `AbsInt`.

info_to_asub/5: PREDICATE

`info_to_asub(AbsInt,Kind,InputUser,Qv,ASub)`

Obtains the abstract substitution `ASub` on variables `Qv` for domain `AbsInt` from the user supplied information `InputUser` refering to properties on `Qv`. It works by calling `input_interface/5` on each property of `InputUser` which is a native property, so that they are accumulated, and then calls `input_user_interface/4`.

full_info_to_asub/4: PREDICATE

`full_info_to_asub(AbsInt,InputUser,Qv,ASub)`

Same as `info_to_asub(AbsInt,InputUser,Qv,ASub)` except that it fails if some property in `InputUser` is not native or not relevant to the domain `AbsInt`.

asub_to_info/5: PREDICATE

`asub_to_info(AbsInt,ASub,Qv,OutputUser,CompProps)`

Transforms an abstract substitution `ASub` on variables `Qv` for a domain `AbsInt` to a list of state properties `OutputUser` and computation properties `CompProps`, such that properties are visible in the preprocessing unit. It fails if `ASub` represents bottom. It works by calling `asub_to_native/4`.

asub_to_native/5: PREDICATE

`asub_to_native(AbsInt,ASub,Qv,NativeStat,NativeComp)`

`NativeStat` and `NativeComp` are the list of native (state and computational, resp.) properties that are the concretization of abstract substitution `ASub` on variables `Qv` for domain `AbsInt`. These are later translated to the properties which are visible in the preprocessing unit.

unknown_call/5: PREDICATE

`unknown_call(AbsInt,Sg,Vars,Call,Succ)`

`Succ` is the result of adding to `Call` the "topmost" abstraction in domain `AbsInt` of the variables `Vars` involved in a literal `Sg` whose definition is not present in the preprocessing unit. I.e., it is like the conjunction of the information in `Call` with the top for a subset of its variables.

unknown_call/4: PREDICATE

`unknown_call(AbsInt,Vars,Call,Succ)`

`Succ` is the result of adding to `Call` the "topmost" abstraction in domain `AbsInt` of the variables `Vars` involved in a literal whose definition is not present in the preprocessing unit. I.e., it is like the conjunction of the information in `Call` with the top for a subset of its variables.

unknown_entry/4: PREDICATE

`unknown_entry(AbsInt,Sg,Vars,Entry)`

`Entry` is the "topmost" abstraction in domain `AbsInt` of variables `Vars` corresponding to literal `Sg`.

unknown_entry/3: PREDICATE

`unknown_entry(AbsInt,Vars,Entry)`

`Entry` is the "topmost" abstraction in domain `AbsInt` of variables `Vars`.

empty_entry/3: PREDICATE

`empty_entry(AbsInt,Vars,Entry)`

`Entry` is the "empty" abstraction in domain `AbsInt` of variables `Vars`. I.e., it is the abstraction of a substitution on `Vars` in which all variables are unbound: free and unaliased.

collect_types_in_abs/4: PREDICATE

`collect_types_in_abs(ASub,AbsInt,Types,Tail)`

Collects the type symbols occurring in `ASub` of domain `AbsInt` in a difference list `Types-Tail`.

rename_types_in_abs/4: PREDICATE

`rename_types_in_abs(ASub0,AbsInt,Dict,ASub1)`

Renames the type symbols occurring in `ASub0` of domain `AbsInt` for the corresponding symbols as in (avl-tree) `Dict` yielding `ASub1`.

dom_statistics/2: PREDICATE

`dom_statistics(AbsInt,Info)`

Obtains in list `Info` statistics about the results of the abstract interpreter `AbsInt`.

abstract_instance/5: PREDICATE

Usage: `abstract_instance(AbsInt,Sg1,Proj1,Sg2,Proj2)`

The pair <Sg1,Proj1> is an abstract instance of the pair <Sg2,Proj2>, i.e., the concretization of <Sg1,Proj1> is included in the concretization of <Sg2,Proj2>.

contains_parameters/2: PREDICATE

`contains_parameters(AbsInt,Subst)`

True if an abstract substitution `Subst` contains type parameters

13.3 Documentation on multifiles (`domains`)

aidomain/1: PREDICATE

`aidomain(AbsInt)`

Declares that `AbsInt` identifies an abstract domain. The predicate is *multifile*.

13.4 Documentation on internals (`domains`)

success_builtin/7: PREDICATE

`success_builtin(AbsInt,Type,Sv,Condvars,HvFv_u,Call,Succ)`

`Succ` is the success substitution on domain `AbsInt` for a call `Call` to a goal of a "builtin" (domain dependent) type `Type` with variables `Sv`. `Condvars` can be used to transfer some information from `special_builtin/5`.

call_to_success_builtin/7: PREDICATE

`call_to_success_builtin(AbsInt,Type,Sg,Sv,Call,Proj,Succ)`

`Succ` is the success substitution on domain `AbsInt` for a call `Call` to a goal `Sg` with variables `Sv` considered of a "builtin" (domain dependent) type `Type`. `Proj` is `Call` projected on `Sv`.

input_interface/5: PREDICATE

`input_interface(AbsInt,Prop,Kind,Struc0,Struc1)`

`Prop` is a native property that is relevant to domain `AbsInt` (i.e., the domain knows how to fully –`+Kind`=perfect– or approximately –`-Kind`=approx– abstract it) and `Struct1` is a (domain defined) structure resulting of adding the (domain dependent) information conveyed by `Prop` to structure `Struct0`. This way, the properties relevant to a domain are being accumulated.

input_user_interface/4: PREDICATE

`input_user_interface(AbsInt,Struct,Qv,ASub)`

`ASub` is the abstraction in `AbsInt` of the information collected in `Struct` (a domain defined structure) on variables `Qv`.

13.5 Known bugs and planned improvements (`domains`)

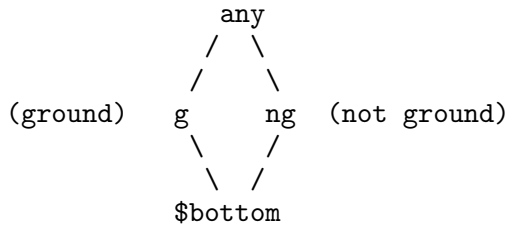
- When interpreting assertions (and native) should take into account things like sourcename(X):- atom(X) and true pred atom(X) => atm(X).
- `body_succ_builtin/9` seems to introduce spurious choice-points.
- Property `covered/2` is not well understood by the domains.
- Operation `amgu/5` is missing.

14 Simple groundness abstract domain

Author(s): Claudio Vaucheret.

This module implements the abstract operations of a simple groundness domain for the PLAI framework of abstract interpretation. An abstract substitution is a list of Var/Mode elements, where Var is a variable and Mode is "any", "g" or "ng".

The abstract domain lattice is:



14.1 Usage and interface (gr)

- **Library usage:**
  `:- use_module(library(gr)).`
- **Exports:**
  - *Predicates:*
    `gr_call_to_entry/8`, `gr_exit_to_prime/7`, `gr_project/3`, `gr_extend/4`, `gr_compute_lub/2`, `gr_glb/3`, `gr_less_or_equal/2`, `gr_sort/2`, `gr_call_to_success_fact/8`, `gr_special_builtin/4`, `gr_success_builtin/5`, `gr_call_to_success_builtin/6`, `gr_input_interface/4`, `gr_input_user_interface/3`, `gr_asub_to_native/3`, `gr_unknown_call/3`, `gr_unknown_entry/2`, `gr_empty_entry/2`.
  - *Regular Types:*
    `extrainfo/1`.
- **Imports:**
  - *System library modules:*
    `messages`, `sort`, `terms_vars`, `terms_check`, `sets`.
  - *Internal (engine) modules:*
    `term_basic`, `arithmetic`, `atomic_basic`, `basic_props`, `basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_support`.
  - *Packages:*
    `prelude`, `nonpure`, `assertions`, `regtypes`, `basicmodes`.

14.2 Documentation on exports (gr)

gr_call_to_entry/8: PREDICATE

Usage: `gr_call_to_entry(Sv,Sg,Hv,Head,Fv,Proj,Entry,ExtraInfo)`

It obtains the abstract substitution `Entry` which results from adding the abstraction of the `Sg` = `Head` to `Proj`, later projecting the resulting substitution onto `Hv`. This is done as follows:

- If `Sg` and `Head` are identical up to renaming it is just renaming `Proj` and adding the `Fv`
- If `Hv` = [], `Entry` is just adding the `Fv`
- Otherwise, it will
  - obtain in Binds the primitive equations corresponding to Sg=Head
  - add to Proj the variables in Hv as not ground in Temp1
  - update Temp1, grounding some variables obtaining Temp2
  - insert Fv in Temp2 as 'any' obtaining Temp3
  - projects Temp3 onto Hv + Fv obtaining Entry

The meaning of the variables is

- `Sv` is a list of subgoal variables.
- `Sg` is the subgoal being analysed.
- `Head` is the Head of the clause being analysed.
- `Fv` is a list of free variables in the body of the clause being considered.
- `Proj` is the abstract substitution `Call` projected onto `Sv`.
- `Entry` is the Abstract entry substitution (i.e. the abstract subtitution obtained after the abstract unification of `Sg` and `Head` projected onto `Hv + Fv`).
- `ExtraInfo` Info computed during the call_to_entry that can be reused during the exit_to_prime step.

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

`Sv` is currently a term which is not a free variable.	(`nonvar/1`)
`Sg` is currently a term which is not a free variable.	(`nonvar/1`)
`Hv` is currently a term which is not a free variable.	(`nonvar/1`)
`Head` is currently a term which is not a free variable.	(`nonvar/1`)
`Fv` is currently a term which is not a free variable.	(`nonvar/1`)
`Proj` is currently a term which is not a free variable.	(`nonvar/1`)
`Entry` is a free variable.	(`var/1`)
`ExtraInfo` is a free variable.	(`var/1`)
`Sv` is a list.	(`list/1`)
`Sg` is a term which represents a goal, i.e., an atom or a structure.	(`callable/1`)
`Hv` is a list.	(`list/1`)
`Head` is a term which represents a goal, i.e., an atom or a structure.	(`callable/1`)
`Fv` is a list.	(`list/1`)
`Proj` is an abstract substitution	(`absu/1`)
`Entry` is an abstract substitution	(`absu/1`)
`ExtraInfo` is a par (absu,binds)	(`extrainfo/1`)

gr_exit_to_prime/7: PREDICATE

Usage: `gr_exit_to_prime(Sg,Hv,Head,Sv,Exit,ExtraInfo,Prime)`

It computes the prime abstract substitution `Prime`, i.e. the result of going from the abstract substitution over the head variables `Exit`, to the abstract substitution over the variables in the subgoal. It will:

- If `Exit` is '$bottom', `Prime` will be also '$bottom'.

- If `Flag` = yes (`Head` and `Sg` identical up to renaming) it is just renaming Exit %
- If `Hv` = [], `Prime` = `{ X/g | forall X in Sv }`
- Otherwise: it will
  - obtain the primitive equations corresponding to Sg=Head from `Extrainfo`.
  - projects `Exit` onto `Hv` obtaining `BPrime`.
  - merge `Proj` from `Extrainfo` and `BPrime` obtaining `TempPrime`.
  - update `TempPrime`, grounding some variables obtaining `NewTempPrime`.
  - projects `NewTempPrime` onto `Sv` obtaining `Prime`.

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

`Sg` is currently a term which is not a free variable.	(`nonvar/1`)
`Hv` is currently a term which is not a free variable.	(`nonvar/1`)
`Head` is currently a term which is not a free variable.	(`nonvar/1`)
`Sv` is currently a term which is not a free variable.	(`nonvar/1`)
`Exit` is currently a term which is not a free variable.	(`nonvar/1`)
`ExtraInfo` is a free variable.	(`var/1`)
`Prime` is a free variable.	(`var/1`)
`Sg` is a list.	(`list/1`)
`Hv` is a list.	(`list/1`)
`Head` is a term which represents a goal, i.e., an atom or a structure.	(`callable/1`)
`Sv` is a term which represents a goal, i.e., an atom or a structure.	(`callable/1`)
`Exit` is an abstract substitution	(`absu/1`)
`ExtraInfo` is a par (absu,binds)	(`extrainfo/1`)
`Prime` is an abstract substitution	(`absu/1`)

gr_project/3: PREDICATE

Usage: `gr_project(Asub,Vars,Proj)`

`Proj` is the result of eliminating from `Asub` all `X/Value` such that `X` is not in `Vars`

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

`Asub` is currently a term which is not a free variable.	(`nonvar/1`)
`Vars` is currently a term which is not a free variable.	(`nonvar/1`)
`Proj` is a free variable.	(`var/1`)
`Asub` is an abstract substitution	(`absu/1`)
`Vars` is a list.	(`list/1`)
`Proj` is an abstract substitution	(`absu/1`)

gr_extend/4: PREDICATE

Usage: `gr_extend(Prime,Sv,Call,Succ)`

If `Prime` = '$bottom', `Succ` = '$bottom'. If `Sv` = [], `Call` = `Succ`. Otherwise, `Succ` is computed updating the values of `Call` with those in `Prime`

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

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

`Call` is currently a term which is not a free variable. (`nonvar/1`)
`Succ` is a free variable. (`var/1`)
`Prime` is an abstract substitution (`absu/1`)
`Sv` is a list. (`list/1`)
`Call` is an abstract substitution (`absu/1`)
`Succ` is an abstract substitution (`absu/1`)

gr_compute_lub/2: PREDICATE

Usage: `gr_compute_lub(ListASub,Lub)`

It computes the least upper bound of a set of abstract substitutions. For each two abstract substitutions `ASub1` and `ASub2` in `ListASub`, obtaining the lub is just:

foreach X/Value1 in `ASub1` and X/Value2 in `ASub2`:

- if Value1 == Value2, X/Value1 in Lub
- otherwise, X/any in Lub

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

`ListASub` is currently a term which is not a free variable. (`nonvar/1`)
`Lub` is a free variable. (`var/1`)
`ListASub` is a list of `absu`s. (`list/2`)
`Lub` is an abstract substitution (`absu/1`)

gr_glb/3: PREDICATE

Usage: `gr_glb(ASub0,ASub1,Glb)`

`Glb` is the great lower bound of `ASub0` and `ASub1`

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

`ASub0` is currently a term which is not a free variable. (`nonvar/1`)
`ASub1` is currently a term which is not a free variable. (`nonvar/1`)
`Glb` is a free variable. (`var/1`)
`ASub0` is an abstract substitution (`absu/1`)
`ASub1` is an abstract substitution (`absu/1`)
`Glb` is an abstract substitution (`absu/1`)

gr_less_or_equal/2: PREDICATE

Usage: `gr_less_or_equal(ASub0,ASub1)`

Succeeds if `ASub1` is more general or equal to `ASub0`. it's assumed the two abstract substitutions are defined on the same variables

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

`ASub0` is currently a term which is not a free variable. (`nonvar/1`)
`ASub1` is currently a term which is not a free variable. (`nonvar/1`)
`ASub0` is an abstract substitution (`absu/1`)
`ASub1` is an abstract substitution (`absu/1`)

gr_sort/2: PREDICATE

Usage: `gr_sort(Asub,Asub_s)`

It sorts the set of `X/Value` in `Asub` ontaining `Asub_s`

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

`Asub` is currently a term which is not a free variable.	(`nonvar/1`)
`Asub_s` is a free variable.	(`var/1`)
`Asub` is an abstract substitution	(`absu/1`)
`Asub_s` is an abstract substitution	(`absu/1`)

gr_call_to_success_fact/8: PREDICATE

Usage: `gr_call_to_success_fact(Sg,Hv,Head,Sv,Call,Proj,Prime,Succ)`

Specialized version of call_to_entry + exit_to_prime + extend for facts

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

`Sg` is currently a term which is not a free variable.	(`nonvar/1`)
`Hv` is currently a term which is not a free variable.	(`nonvar/1`)
`Head` is currently a term which is not a free variable.	(`nonvar/1`)
`Sv` is currently a term which is not a free variable.	(`nonvar/1`)
`Call` is currently a term which is not a free variable.	(`nonvar/1`)
`Proj` is currently a term which is not a free variable.	(`nonvar/1`)
`Prime` is a free variable.	(`var/1`)
`Succ` is a free variable.	(`var/1`)
`Sg` is a term which represents a goal, i.e., an atom or a structure.	(`callable/1`)
`Hv` is a list.	(`list/1`)
`Head` is a term which represents a goal, i.e., an atom or a structure.	(`callable/1`)
`Sv` is a list.	(`list/1`)
`Call` is an abstract substitution	(`absu/1`)
`Proj` is an abstract substitution	(`absu/1`)
`Prime` is an abstract substitution	(`absu/1`)
`Succ` is an abstract substitution	(`absu/1`)

gr_special_builtin/4: PREDICATE

Usage: `gr_special_builtin(SgKey,Sg,Type,Condvars)`

Satisfied if the builtin does not need a very complex action. It divides builtins into groups determined by the flag returned in the second argument + some special handling for some builtins:

1. *new_ground* if the builtin makes all variables ground whithout imposing any condition on the previous freeness values of the variables
2. *old_ground* if the builtin requires the variables to be ground
3. *old_new_ground* if the builtin requires some variables to be ground and grounds the rest
4. unchanged if we cannot infer anything from the builtin, the substitution remains unchanged and there are no conditions imposed on the previous freeness values of the variables.

5. *some* if it makes some variables ground without imposing conditions
6. Sgkey, special handling of some particular builtins

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

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

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

`Type` is a free variable. (`var/1`)

`Condvars` is a free variable. (`var/1`)

`SgKey` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        nnegint(A).
```

(`predname/1`)

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

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

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

gr_success_builtin/5: PREDICATE

Usage: `gr_success_builtin(Type,Sv_u,Condv,Call,Succ)`

Obtains the success for some particular builtins:

- If Type = new_ground, it updates Call making all vars in Sv_u ground
- If Type = bottom, Succ = '$bottom'
- If Type = unchanged, Succ = Call
- If Type = some, it updates Call making all vars in Condv ground
- If Type = old_ground, if grouds all variables in Sv and checks that no free variables has becomed ground
- If Type = old_ground, if grounds all variables in OldG and checks thatno free variables has becomed ground. If so, it grounds all variables in NewG
- Otherwise Type is the SgKey of a particular builtin for each the Succ is computed

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

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

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

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

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

`Succ` is a free variable. (`var/1`)

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

`Sv_u` is a list. (`list/1`)

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

`Call` is an abstract substitution (`absu/1`)

`Succ` is an abstract substitution (`absu/1`)

gr_call_to_success_builtin/6: PREDICATE

Usage: `gr_call_to_success_builtin(SgKey,Sh,Sv,Call,Proj,Succ)`

Handles those builtins for which computing `Proj` is easier than `Succ`

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

`SgKey` is currently a term which is not a free variable. (`nonvar/1`)
`Sh` is currently a term which is not a free variable. (`nonvar/1`)
`Sv` is currently a term which is not a free variable. (`nonvar/1`)
`Call` is currently a term which is not a free variable. (`nonvar/1`)
`Proj` is currently a term which is not a free variable. (`nonvar/1`)
`Succ` is a free variable. (`var/1`)
`SgKey` is a Name/Arity structure denoting a predicate name:

```
predname(P/A) :-
        atm(P),
        nnegint(A).
```

(`predname/1`)
`Sh` is a term which represents a goal, i.e., an atom or a structure. (`callable/1`)
`Sv` is a list. (`list/1`)
`Call` is an abstract substitution (`absu/1`)
`Proj` is an abstract substitution (`absu/1`)
`Succ` is an abstract substitution (`absu/1`)

gr_input_interface/4: PREDICATE

Usage: `gr_input_interface(Prop,Kind,Struc0,Struc1)`

Adds native property `Prop` to the structure accumulating the properties relevant to this domain, namely: ground/1, free/1, and not_ground/1.

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

`Prop` is currently a term which is not a free variable. (`nonvar/1`)
`Struc0` is currently a term which is not a free variable. (`nonvar/1`)
`Struc1` is currently a term which is not a free variable. (`nonvar/1`)

gr_input_user_interface/3: PREDICATE

Usage: `gr_input_user_interface(InputUser,Qv,ASub)`

Obtains the abstract substitution for gr from the native properties found in the user supplied info.

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

`InputUser` is currently a term which is not a free variable. (`nonvar/1`)
`Qv` is currently a term which is not a free variable. (`nonvar/1`)
`ASub` is currently a term which is not a free variable. (`nonvar/1`)
`InputUser` is any term. (`term/1`)
`Qv` is a list. (`list/1`)
`ASub` is an abstract substitution (`absu/1`)

gr_asub_to_native/3: PREDICATE

Usage: `gr_asub_to_native(ASub,Qv,ASub_user)`

The user friendly format consists in extracting the ground variables and the nonground variables

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

`ASub` is currently a term which is not a free variable. (`nonvar/1`)
`Qv` is currently a term which is not a free variable. (`nonvar/1`)
`ASub_user` is a free variable. (`var/1`)
`ASub` is an abstract substitution (`absu/1`)
`Qv` is a list. (`list/1`)
`ASub_user` is any term. (`term/1`)

gr_unknown_call/3: PREDICATE

Usage: `gr_unknown_call(Call,Vars,Succ)`

Gives the "top" value for the variables involved in a literal whose definition is not present, and adds this top value to `Call`

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

`Call` is currently a term which is not a free variable. (`nonvar/1`)
`Vars` is currently a term which is not a free variable. (`nonvar/1`)
`Succ` is a free variable. (`var/1`)
`Call` is an abstract substitution (`absu/1`)
`Vars` is a list. (`list/1`)
`Succ` is an abstract substitution (`absu/1`)

gr_unknown_entry/2: PREDICATE

Usage: `gr_unknown_entry(Qv,Call)`

Gives the "top" value for the variables involved in a literal whose definition is not present, and adds this top value to Call. In this domain the top value is X/any forall X in the set of variables

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

`Qv` is currently a term which is not a free variable. (`nonvar/1`)
`Call` is a free variable. (`var/1`)
`Qv` is a list. (`list/1`)
`Call` is an abstract substitution (`absu/1`)

gr_empty_entry/2: PREDICATE

Usage: `gr_empty_entry(Vars,Entry)`

Gives the "empty" value in this domain for a given set of variables `Vars`, resulting in the abstract substitution `Entry`. I.e., obtains the abstraction of a substitution in which all variables `Vars` are unbound: free and unaliased. In this domain the empty value is equivalent to the unknown value

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

`Vars` is currently a term which is not a free variable. (`nonvar/1`)
`Entry` is a free variable. (`var/1`)
`Vars` is a list. (`list/1`)
`Entry` is an abstract substitution (`absu/1`)

extrainfo/1: REGTYPE

Usage: `extrainfo(E)`

`E` is a par (absu,binds)

14.3 Documentation on internals (gr)

absu/1: REGTYPE

Usage: `absu(A)`

`A` is an abstract substitution

absu_elem/1: REGTYPE

Usage: `absu_elem(E)`

`E` is a single substitution

gr_mode/1: REGTYPE

Usage: `gr_mode(M)`

`M` is g (ground), ng (nonground), or any

binds/1: REGTYPE

Usage: `binds(B)`

`B` is a list of bindings

binding/1: REGTYPE

Usage: `binding(B)`

`B` is a triple (X,Term,Vars), where X is a variable, Term is a term and Vars is the set of variables in Term

References

[**APG06**] E. Albert, G. Puebla, and J. Gallagher.
Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates.
In *15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, number 3901 in LNCS, pages 115–132. Springer-Verlag, April 2006.

[**BCC04**] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla-(Eds.).
The Ciao System. Reference Manual (v1.10).
Technical report, School of Computer Science (UPM), 2004.
Available at `http://www.ciaohome.org`.

[**BCHP96**] F. Bueno, D. Cabeza, M. 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.

[**BGH94**] F. Bueno, M. García de la Banda, and M. Hermenegildo.
The PLAI Abstract Interpretation System.
Technical Report CLIP2/94.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, February 1994.

[**BLGH04**] F. Bueno, P. López-García, and M. 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.

[**BLGPH06**] F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo.
A Tutorial on Program Development and Optimization using the Ciao Preprocessor.
Technical Report CLIP2/06, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2006.

[**CH94**] D. Cabeza and M. Hermenegildo.
Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information.
In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.

[**CMB93**] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo.
Improving Abstract Interpretations by Combining Domains.
In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 194–206. ACM, June 1993.

[**COS96**] The COSYTEC Team.
CHIP System Documentation, April 1996.

[**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. 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. 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.

[**Dum94**] Veroniek Dumortier.
Freeness and Related Analyses of Constraint Logic Programs Using Abstract Interpretation.
PhD thesis, K.U.Leuven, Dept. of Computer Science, October 1994.

[**GdW94**] J.P. Gallagher and D.A. de Waal.
Fast and precise regular approximations of logic programs.
In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[**GHM00**] M. García de la Banda, M. Hermenegildo, and K. Marriott.
Independence in CLP Languages.
ACM Transactions on Programming Languages and Systems, 22(2):269–339, March 2000.

[**Her99**] M. Hermenegildo.
A Documentation Generator for Logic Programming Systems.
Technical Report CLIP10/99.0, Facultad de Informática, UPM, September 1999.

[**HPMS00**] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey.
Incremental Analysis of Constraint Logic Programs.
ACM Transactions on Programming Languages and Systems, 22(2):187–223, March 2000.

[**HR95**] M. Hermenegildo and F. Rossi.
Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions.
Journal of Logic Programming, 22(1):1–45, 1995.

[**JB92**] G. Janssens and M. Bruynooghe.
Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation.
Journal of Logic Programming, 13(2 and 3):205–258, July 1992.

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

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

[**Leu98**] M. Leuschel.
On the Power of Homeomorphic Embedding for Online Termination.

In Giorgio Levi, editor, *Proceedings of SAS'98*, volume 1503 of *LNCS*, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[**LGHD96a**]
P. López-García, M. Hermenegildo, and S. K. Debray.
A Methodology for Granularity Based Control of Parallelism in Logic Programs.
J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation, 21:715–734, 1996.

[**LGHD96b**]
P. López-García, M. 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.

[**MBdlBH99**]
K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo.
Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism.
Journal of Logic Programming, 38(2):165–218, February 1999.

[**MH89a**] K. Muthukumar and M. Hermenegildo.
Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation.
Technical Report ACA-ST-232-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, March 1989.

[**MH89b**] 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.

[**MH91**] K. Muthukumar and M. Hermenegildo.
Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation.
In *International Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[**MH92**] K. Muthukumar and M. Hermenegildo.
Compile-time Derivation of Variable Dependency Using Abstract Interpretation.
Journal of Logic Programming, 13(2/3):315–347, July 1992.

[**MS94**] K. Marriott and P. Stuckey.
Approximating Interaction Between Linear Arithmetic Constraints.
In *1994 International Symposium on Logic Programming*, pages 571–585. MIT Press, 1994.

[**PAH04**] G. Puebla, E. Albert, and M. Hermenegildo.
Abstract Interpretation with Specialized Definitions.
Technical Report CLIP12/2004.0, Technical University of Madrid, School of Computer Science, UPM, September 2004.

[**PBH00**] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Constraint Logic Programs.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[**PRO**] The PROLOG IV Team.
PROLOG IV Manual.

[**SG94**] H. Saglam and J. Gallagher.
Approximating Logic Programs Using Types and Regular Descriptions.
Technical Report CSTR-94-19, Department of Computer Science, University of Bristol, Bristol BS8 1TR, 1994.

[**Son86**] H. Søndergaard.
An application of abstract interpretation of logic programs: occur check reduction.
In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[**VB02**] C. Vaucheret and F. Bueno.
More Precise yet Efficient Type Inference for Logic Programs.
In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

[**VHCL95**] P. Van Hentenryck, A. Cortesi, and B. Le Charlier.
Type analysis of prolog using type graphs.
Journal of Logic Programming, 22(3):179–209, 1995.

Library/Module Index

Predicate/Method Index

L

M

N

O

P

R

S

T

U

W

Property Index

S

T

U

V

Regular Type Index

Declaration Index

Concept Index

Author Index

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, etc., are referred to in the text of the document.

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Y