



ASAP

IST-2001-38059

Advanced Analysis and Specialization for
Pervasive Systems

First Prototype

Deliverable number:	D8
Workpackage:	Integrated Tool (WP7)
Preparation date:	1 March 2004
Due date:	1 March 2004
Classification:	Public
Lead participant:	Tech. Univ. of Madrid (UPM)
Partners contributed:	Tech. Univ. of Madrid (UPM), Univ. of Southampton, Roskilde Univ

Project funded by the European Community under the “Information Society Technologies” (IST) Programme (1998–2002).

Short description:

D8 corresponds to the work in task 7.2, the implementation of the prototype of the integrated tool on top of Ciao. This task has taken as starting point the previous prototype incorporating the tools of the partners: *mECCE*, BTA (a first version of) the binding time analysis for offline specialization, and *CiaoPP-1.0*. The prototype now includes also Logen, a complete BTA, and *CiaoPP-1.1*, incorporating new techniques for program transformation.

The present deliverable is a second version of D8, already delivered in the first period of the project. This second version reflects the changes and improvements made to the prototype. It includes two parts in this document and two attachments. Part I provides a tutorial-like overview of the use of *CiaoPP* in program development, which is quite comprehensive and focuses on giving a high-level glance of the possibilities of the tool (it corresponds to the final version of a paper of the same title to appear in *Science of Computer Programming*, Elsevier Science, 2005). Part II provides a tutorial-like overview of *PyLogen*. The first attachment (see also Section 3 below) is an updated version of the *CiaoPP-1.1 Reference Manual* which, in addition to updates, now includes a tutorial guide for using *CiaoPP*, including descriptions of the new program transformations which have been incorporated, the new menu interface, etc. A second first (virtual) attachment (see Section 2 below) is the *Ciao-1.11 Manual* (available at <http://clip.dia.fi.upm.es/>) which corresponds to a new distribution of the Ciao system. Since this is a very large document it is not included herein, and instead it can be obtained from <http://clip.dia.fi.upm.es/ASAP/Software>.

1 Bottom-Up Analyses

A set of tools based on *bottom-up analysis* has been integrated into *CiaoPP*. Bottom-up analysis has an elegant semantic basis based on the declarative semantics of logic programs, straightforward implementation, and flexible application. In addition, goal-directed or top-down analyses can be simulated through the use of query-answer transformations, of which the so-called “magic set” method is one. These transformations can also serve to increase precision with respect to bottom-up analysis. A toolkit of bottom-up analysis tools and query-answer transformations has been built up. Efficient algorithms for bottom-up analysis are at the heart of the toolkit. A systematic method for implementing analyses using the tools is then set out.

The method is based on (i) abstract compilation of a program into a “domain program”, (ii) computation of (an approximation to) the model of the domain program, and (iii) use of various

query-answer transformations to simulate goal-directed analysis and improve precision. Stages (ii) and (iii) can also be used directly on the original program in some applications. If the domain program has a finite model, then a precise model can be computed in step (ii). If the model is infinite or very large, methods of approximation including regular approximation and others can be used. Abstract compilation can be achieved very flexibly, based on the construction of pre-interpretations based on arbitrary regular types [GH04].

1.1 Interface to the Bottom-Up Analysis Tools

The bottom-up tools have been applied in the automatic BTA tool [CGLH04] associated with LOGEN and with the backwards analysis work [Gal03]. They are primarily intended as an analysis engine to be incorporated in other analyses, rather than used directly from the user interface. The main components of the tools are:

- An implementation of the least model computation, which is the least fixed point of the T_P function.
- Abstraction of a program with respect to a pre-interpretation.
- Derivation of a pre-interpretation by converting a non-deterministic finite tree automaton (NFTA) to a deterministic finite tree automaton (DFTA).
- The backwards analysis transformation.
- Analysis of a program over a domain of NFTAs (i.e. the NFTAs are derived from the program, in contrast to the derivation of a pre-interpretation from a given NFTA).

An interface to the analysis tools is provided by the following predicates:

- `tpr(Cls, M1, M2)`: where `Cls` is a list of clauses from a module P (see below), `M1` is a model of predicates external to `Cls`, and `M2` is the output model, the least fixed point of T_P . This is the core engine of the bottom-up analysis.
- `dm(Module, RegTypeFile, SDV)`: where `Module` is a module name and `RegTypeFile` is a file containing rules for some regular types, and `SDV` is an indicator of which standard types are added to the regular types (e.g. `SDV=sd` means that standard types `static` and `dynamic` are added. The predicate determinizes the regular types, computes and abstract domain program over the corresponding pre-interpretation, and calls `tpr` to compute the model over the pre-interpretation. The output is sent to a file.

- `ba(File, Outfile)`: performs backwards analysis of the module in `File`, with the default assumption that all built-in predicates are observed. The output is a model of the relation between the entry program calls and the observed program points.
- `dfta(RegTypes, File, SDV, OutFile)`: Determinizes the regular types in `RegTypes` with respect to the signature of the program in `File`. `SDV` are the added standard types (see above).
- `tdv(File, Query)`: where `File` is an input program and `Query` is an atomic goal for the program. The output is an NFTA approximation of the program's calls and success patterns w.r.t. computation of `Query`. This is an implementation of the analysis over Nondeterministic Finite Tree Automata (NFTAs) [GP02a].

An interface from the internal representation of programs in the `CiaoPP` system to the above tools has been provided. This has been exploited in analyses that access the assertions in the `CiaoPP` database, for example. These applications are described in Deliverable D17 (Combined Static and Dynamic Checking).

2 CiaoDE: The Ciao Development Environment

A new distribution of the prototype analysis tool¹ has been generated. This new distribution includes in a single package and with a single setup procedure tools which were distributed and installed separately until now, which caused compatibility and dependency problems. This new source-code distribution alleviates those problems, as a single source tree contains highly integrated code whose configuration, installation, and usage is easier than it was before.

3 CiaoPP: The Ciao Program Preprocessor

This integrated tool runs on top of the `Ciao` public domain program development environment, except for some well defined external components, and it has been improved and extended to incorporate most of the new techniques developed in this period (which are described in further detail in other deliverables). In particular, the enclosed reference manual describes the inter-modular fixpoint algorithm for the analysis of modular programs reported in D6; and the efficient, stack-based local unfolding rule based on covering ancestors also reported in D6; the new

¹To be found at <http://clip.dia.fi.upm.es/ASAP/Software>.

determinacy analysis developed which takes advantage of the type and mode analysis already integrated in the tool (reported in D15), etc.

Contents

Deliverable Description	1
1 Bottom-Up Analyses	1
1.1 Interface to the Bottom-Up Analysis Tools	2
2 CiaoDE: The Ciao Development Environment	3
3 CiaoPP: The Ciao Program Preprocessor	3
I Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)	3
4 The Role of Abstract Interpretation	4
4.1 Abstract Verification and Debugging	5
4.2 Abstract Executability and Program Transformation	8
5 Static Analysis and Program Assertions	9
5.1 Modular Static Analysis Basics:	10
5.2 Assertions and Properties:	11
5.3 Type Analysis:	13
5.4 Non-failure and Determinacy Analysis:	14
5.5 Size, Cost, and Termination Analysis:	14
5.6 Decidability, Approximations, and Safety:	15
6 Program Debugging and Assertion Validation	15
6.1 Static Debugging:	16
6.2 Static Checking of Assertions in System Libraries:	18
6.3 Static Checking of User Assertions and Program Validation:	19
6.4 Dynamic Debugging with Run-time Checks:	21
6.5 Performance Debugging and Validation:	22
7 Source Program Optimization	23
7.1 Abstract Specialization:	23
7.2 Parallelization:	24
7.3 Resource and Granularity Control:	25

7.4	Multiple Specialization:	26
7.5	Integration of Abstract Interpretation and Partial Evaluation:	28
II	A Tutorial Overview of the PyLogen System	29
8	Starting PYLOGEN	29
9	Specialising the Regular Expression Interpreter	30
10	Using the Automatic Binding-time Analysis	34
	References	36

Part I

Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)

The technique of Abstract Interpretation [CC77] has allowed the development of sophisticated program analyses which are at the same time provably correct and practical. The semantic approximations produced by such analyses have been traditionally applied to high- and low-level *optimizations* during program compilation, including *program transformation*. More recently, novel and promising applications of semantic approximations have been proposed in the more general context of program development, such as *verification* and *debugging*.

We present a novel programming framework which uses extensively abstract interpretation as a fundamental tool in the program development process. The framework uses modular, incremental abstract interpretation to obtain information about the program, which is then used to validate programs, to detect bugs with respect to partial specifications written using assertions (in the program itself and/or in system libraries), to generate run-time tests for properties which cannot be checked completely at compile-time and simplify them, and to perform high-level program transformations such as multiple abstract specialization, parallelization, and resource usage control, all in a provably correct way.

After introducing some of the basic concepts underlying the approach, the framework is described in a tutorial fashion through the presentation of its implementation in CiaoPP, the preprocessor of the Ciao program development system [BCC⁺97].² Ciao is a multi-paradigm programming system, allowing programming in logic, constraint, and functional styles (as well as a particular form of object-oriented programming). At the heart of Ciao is an efficient logic programming-based kernel language. This allows the use of the very large body of approximation domains, inference techniques, and tools for abstract interpretation-based semantic analysis which have been developed to a powerful and mature level in this area (see, e.g., [MH92, CV94, GdW94, BCHP96, dlBHB⁺96a, HBPLG99] and their references). These techniques and systems

²The first, abridged version of this paper was prepared as a companion to an invited talk at the 2003 Symposium of Static Analysis, SAS'03, and a demonstration of Ciao and CiaoPP at work was performed at the meeting.

can approximate at compile-time, always safely, and with a significant degree of precision, a wide range of properties which is much richer than, for example, traditional types. This includes data structure shape (including pointer sharing), independence, storage reuse, bounds on data structure sizes and other operational variable instantiation properties, as well as procedure-level properties such as determinacy, termination, non-failure, and bounds on resource consumption (time or space cost).

In the rest of the paper we first discuss briefly the specific role of abstract interpretation in different parts of our program development framework (Section 4) and then illustrate it by presenting what is arguably the first and most complete implementation of this idea: CiaoPP [PBH00a, HBPLG99].³ We do this in a tutorial fashion, elaborating on different aspects of how the actual process of program development is aided in an implementation of our framework, by showing examples of CiaoPP at work. Section 5 presents CiaoPP at work performing program analysis, while Section 6 does the same for program debugging and validation, and Section 7 for program transformation and optimization.

Space constraints prevent us from providing a complete set of references to related work on the many topics touched upon in the paper. Thus, we only provide the references most directly related to the papers where all the techniques used in CiaoPP are discussed in detail, which are often our own work. We ask the reader to kindly forgive this. The publications referenced do themselves contain much more comprehensive references to the related work.

4 The Role of Abstract Interpretation

We start by recalling some basic concepts from abstract interpretation. We consider the important class of semantics referred to as *fixpoint semantics*. In this setting, a (monotonic) semantic operator (which we refer to as S_P) is associated with each program P . This S_P function operates on a semantic domain D which is generally assumed to be a complete lattice or, more generally, a chain complete partial order. The meaning of the program (which we refer to as $\llbracket P \rrbracket$) is defined as the least fixpoint of the S_P operator, i.e., $\llbracket P \rrbracket = \text{lfp}(S_P)$. A well-known result is that if S_P is continuous, the least fixpoint is the limit of an iterative process involving at most ω applications of S_P and starting from the bottom element of the lattice.

³In fact, the implementation of the preprocessor is generic in that it can be easily customized to different programming systems and dialects and in that it is designed to allow the integration of additional analyses in a simple way. As a particularly interesting example, the preprocessor has been adapted for use with the CHIP CLP(FD) system. This has resulted in CHIPRE, a preprocessor for CHIP which has been shown to detect non-trivial programming errors in CHIP programs. More information on the CHIPRE system and an example of a debugging session with it can be found in [PBH00a].

In the abstract interpretation technique, the program P is interpreted over a non-standard domain called the *abstract* domain D_α which is simpler than the *concrete* domain D . The abstract domain D_α is usually constructed with the objective of computing safe approximations of the semantics of programs, and the semantics w.r.t. this abstract domain, i.e., the *abstract semantics* of the program, is computed (or approximated) by replacing the operators in the program by their abstract counterparts. The abstract domain D_α also has a lattice structure. The concrete and abstract domains are related via a pair of monotonic mappings: *abstraction* $\alpha : D \mapsto D_\alpha$, and *concretization* $\gamma : D_\alpha \mapsto D$, which relate the two domains by a Galois insertion (or a Galois connection) [CC77].

One of the fundamental results of abstract interpretation is that an abstract semantic operator S_P^α for a program P can be defined which is correct w.r.t. S_P in the sense that $\gamma(\text{lfp}(S_P^\alpha))$ is an approximation of $\llbracket P \rrbracket$, and, if certain conditions hold (e.g., ascending chains are finite in the D_α lattice), then the computation of $\text{lfp}(S_P^\alpha)$ terminates in a finite number of steps. We will denote $\text{lfp}(S_P^\alpha)$, i.e., the result of abstract interpretation for a program P , as $\llbracket P \rrbracket_\alpha$.

Typically, abstract interpretation guarantees that $\llbracket P \rrbracket_\alpha$ is an *over*-approximation of the abstract semantics of the program itself, $\alpha(\llbracket P \rrbracket)$. Thus, we have that $\llbracket P \rrbracket_\alpha \supseteq \alpha(\llbracket P \rrbracket)$, which we will denote as $\llbracket P \rrbracket_{\alpha+}$. Alternatively, the analysis can be designed to safely *under*-approximate the actual semantics, and then we have that $\llbracket P \rrbracket_\alpha \subseteq \alpha(\llbracket P \rrbracket)$, which we denote as $\llbracket P \rrbracket_{\alpha-}$.

4.1 Abstract Verification and Debugging

Both program verification and debugging compare the *actual semantics* of the program, i.e., $\llbracket P \rrbracket$, with an *intended semantics* for the same program, which we will denote by I . This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. In Table 1 we define classical verification problems in a set-theoretic formulation as simple relations between $\llbracket P \rrbracket$ and I .

Using the exact actual or intended semantics for automatic verification and debugging is in general not realistic, since the exact semantics can be typically only partially known, infinite, too expensive to compute, etc. On the other hand the abstract interpretation technique allows computing *safe* approximations of the program semantics. The key idea in our approach [BDD⁺97, HPB99, PBH00c] is to use the abstract approximation $\llbracket P \rrbracket_\alpha$ directly in program verification and debugging tasks.

A number of approaches have already been proposed which make use to some extent of abstract interpretation in verification and/or debugging tasks. Abstractions were used in the context of algorithmic debugging in [LS88]. Abstract interpretation for debugging of imperative programs has been studied by Bourdoncle [Bou93], by Comini et al. for the particular case of algo-

Property	Definition
P is partially correct w.r.t. I	$\llbracket P \rrbracket \subseteq I$
P is complete w.r.t. I	$I \subseteq \llbracket P \rrbracket$
P is incorrect w.r.t. I	$\llbracket P \rrbracket \not\subseteq I$
P is incomplete w.r.t. I	$I \not\subseteq \llbracket P \rrbracket$

Table 1: Set theoretic formulation of verification problems

rithmic debugging of logic programs [CLV95] (making use of partial specifications) [CLMV99], and very recently by P. Cousot [Cou03].

Our first objective herein is to present the implications of the use of *approximations* of both the intended and actual semantics in the verification and debugging process. As we will see, the possible loss of accuracy due to approximation prevents full verification in general. However, and interestingly, it turns out that in many cases useful verification and debugging conclusions can still be derived by comparing the approximations of the actual semantics of a program to the (also possibly approximated) intended semantics.

In our approach we actually compute the abstract approximation $\llbracket P \rrbracket_\alpha$ of the concrete semantics of the program $\llbracket P \rrbracket$ and compare it directly to the (also approximate) intention (which is given in terms of *assertions* [PBH00b]), following almost directly the scheme of Table 1. This approach can be very attractive in programming systems where the compiler already performs such program analysis in order to use the resulting information to, e.g., optimize the generated code, since in these cases the compiler will compute $\llbracket P \rrbracket_\alpha$ anyway. Alternatively, $\llbracket P \rrbracket_\alpha$ can always be computed on demand.

For now, we assume that the program specification is given as a semantic value $I_\alpha \in D_\alpha$. Comparison between actual and intended semantics of the program is most easily done in the same domain, since then the operators on the abstract lattice, that are typically already defined in the analyzer, can be used to perform this comparison. Thus, it is interesting to study the implications of comparing I_α and $\llbracket P \rrbracket_\alpha$, which is an approximation of $\alpha(\llbracket P \rrbracket)$.

In Table 2 we propose (sufficient) conditions for correctness and completeness w.r.t. I_α , which can be used when $\llbracket P \rrbracket$ is approximated. Several instrumental conclusions can be drawn from these relations.

Analyses which over-approximate the actual semantics (i.e., those denoted as $\llbracket P \rrbracket_{\alpha+}$), are specially suited for proving partial correctness and incompleteness with respect to the abstract specification I_α . It will also be sometimes possible to prove incorrectness in the extreme case in which the semantics inferred for the program is incompatible with the abstract specification,

Property	Definition	Sufficient condition
P is partially correct w.r.t. I_α	$\alpha(\llbracket P \rrbracket) \subseteq I_\alpha$	$\llbracket P \rrbracket_{\alpha^+} \subseteq I_\alpha$
P is complete w.r.t. I_α	$I_\alpha \subseteq \alpha(\llbracket P \rrbracket)$	$I_\alpha \subseteq \llbracket P \rrbracket_{\alpha^-}$
P is incorrect w.r.t. I_α	$\alpha(\llbracket P \rrbracket) \not\subseteq I_\alpha$	$\llbracket P \rrbracket_{\alpha^-} \not\subseteq I_\alpha$, or $\llbracket P \rrbracket_{\alpha^+} \cap I_\alpha = \emptyset \wedge \llbracket P \rrbracket_{\alpha^-} \neq \emptyset$
P is incomplete w.r.t. I_α	$I_\alpha \not\subseteq \alpha(\llbracket P \rrbracket)$	$I_\alpha \not\subseteq \llbracket P \rrbracket_{\alpha^+}$

Table 2: Validation problems using approximations

i.e., when $\llbracket P \rrbracket_{\alpha^+} \cap I_\alpha = \emptyset$. We also note that it will only be possible to prove total correctness if the abstraction is *precise*, i.e., $\llbracket P \rrbracket_\alpha = \alpha(\llbracket P \rrbracket)$. According to Table 2 completeness requires $\llbracket P \rrbracket_{\alpha^-}$ and partial correctness requires $\llbracket P \rrbracket_{\alpha^+}$. Thus, the only possibility is that the abstraction is precise.

On the other hand, we use $\llbracket P \rrbracket_{\alpha^-}$ to denote the (less frequent) case in which analysis under-approximates the actual semantics. In such case, it will be possible to prove completeness and incorrectness. In this case, partial correctness and incompleteness can only be proved if the analysis is precise.

If analysis information allows us to conclude that the program is incorrect or incomplete w.r.t. I_α , an (abstract) symptom has been found which ensures that the program does not satisfy the requirement. Thus, debugging should be initiated to locate the program construct responsible for the symptom. Since $\llbracket P \rrbracket_{\alpha^+}$ often contains information associated to program points, it is often possible to use this information directly and/or the analysis graph itself to locate the earliest program point where the symptom occurs (see Section 6). Also, note that the whole setting is even more interesting if the I_α itself is considered an approximation (i.e., we consider I_α^+ and I_α^-), as is the case in the assertions providing upper- and lower-bounds on cost in the examples of Section 6.

It is important to point out that the use of safe approximations is what gives the essential power to the approach. As an example, consider that classical examples of assertions are type declarations. However, herein we are interested in supporting a much more powerful setting in which assertions can be of a much more general nature, stating additionally other properties, some of which cannot always be determined statically for all programs. These properties may include properties defined by means of user programs and extend beyond the predefined set which may be natively understandable by the available static analyzers. Also, only a small number of (even zero) assertions may be present in the program, i.e., the assertions are *optional*. In general, we do not wish to limit the programming language or the language of assertions unnecessarily in

Property	Definition	Sufficient condition
L is abstractly executable to <i>true</i> in P	$RT(L, P) \subseteq TS(L, P)$	$\exists \lambda' \in A_{TS}(\overline{B}, D_\alpha) : \lambda_L \sqsubseteq \lambda'$
L is abstractly executable to <i>false</i> in P	$RT(L, P) \subseteq FF(L, P)$	$\exists \lambda' \in A_{FF}(\overline{B}, D_\alpha) : \lambda_L \sqsubseteq \lambda'$

Table 3: Abstract Executability

order to make the validity of the assertions statically decidable (and, consequently, the proposed framework needs to deal throughout with approximations).

Additional discussions and more details about the foundations and implementation issues of our approach can be found in [BDD⁺97, HPB99, PBH00c, PBH00a].

4.2 Abstract Executability and Program Transformation

In our program development framework, abstract interpretation also plays a fundamental role in the areas of program transformation and program optimization. Optimizations are performed by means of the concept of *abstract executability* [GH91, PH97]. This allows reducing at compile-time certain program fragments to the values *true*, *false*, or *error*, or to a simpler program fragment, by application of the information obtained via abstract interpretation. This allows optimizing and transforming the program (and also detecting errors at compile-time in the case of *error*).

For simplicity, we will limit herein the discussion to reducing a procedure call or program fragment L (for example, a “literal” in the case of logic programming) to either *true* or *false*. Each run-time invocation of the procedure call L will have a *local environment* which stores the particular values of each variable in L for that invocation. We will use θ to denote this environment (composed of assignments of values to variables, i.e., *substitutions*) and the restriction (projection) of the environment θ to the variables of a procedure call L is denoted $\theta|_L$.

We now introduce some definitions. Given a procedure call L without side-effects in a program P we define the *trivial success set* of L in P as $TS(L, P) = \{\theta|_L : L\theta \text{ succeeds exactly once in } P \text{ with empty answer substitution } (\epsilon)\}$. Similarly, given a procedure call L from a program P we define the *finite failure set* of L in P as $FF(L, P) = \{\theta|_L : L\theta \text{ fails finitely in } P\}$.

Finally, given a procedure call L from a program P we define the *run-time substitution set* of L in P , denoted $RT(L, P)$, as the set of all possible substitutions (run-time environments) in the execution state just prior to executing the procedure call L in any possible execution of program

P .

Table 3 shows the conditions under which a procedure call L is abstractly executable to either *true* or *false*. In spite of the simplicity of the concepts, these definitions are not directly applicable in practice since $RT(L, P)$, $TS(L, P)$, and $FF(L, P)$ are generally not known at compile time. However, it is usual to use a *collecting semantics* as concrete semantics for abstract interpretation so that analysis computes for each procedure call L in the program an abstract substitution λ_L which is a safe approximation of $RT(L, P)$, i.e. $\forall L \in P \ RT(L, P) \subseteq \gamma(\lambda_L)$.

Also, under certain conditions we can compute either automatically or by hand sets of abstract values $A_{TS}(\bar{L}, D_\alpha)$ and $A_{FF}(\bar{L}, D_\alpha)$ where \bar{L} stands for the *base form* of L , i.e., where all the arguments of L contain distinct free variables. Intuitively they contain abstract values in domain D_α which guarantee that the execution of \bar{L} trivially succeeds (resp. finitely fails). For soundness it is required that $\forall \lambda \in A_{TS}(\bar{L}, D_\alpha) \ \gamma(\lambda) \subseteq TS(\bar{L}, P)$ and $\forall \lambda \in A_{FF}(\bar{L}, D_\alpha) \ \gamma(\lambda) \subseteq FF(\bar{L}, P)$.

Even though the simple optimizations illustrated above may seem of narrow applicability, in fact for many builtin procedures such as those that check basic types or which inspect the structure of data, even these simple optimizations are indeed very relevant. Two non-trivial examples of this are their application to simplifying independence tests in program parallelization [PH99] (Section 7) and the optimization of delay conditions in logic programs with dynamic procedure call scheduling order [PdIBMS97].

These and other more powerful abstract executability rules are embedded in the multivariant abstract interpreter in our program development framework. The resulting system performs essentially all high- and low-level program optimizations and transformations during program development and in compilation. In fact, the combination of the concept of abstract executability and multivariant abstract interpretation has been shown to be a very powerful program transformation and optimization tool, capable of performing essentially all the transformations traditionally done via partial evaluation [PH99, PHG99, CC02, Leu98]. Also, the class of optimizations which can be performed can be made to cover traditional lower-level optimizations as well, provided the lower-level code to be optimized is “reflected” at the source level or if the abstract interpretation is performed directly at the object level.

5 Static Analysis and Program Assertions

The fundamental functionality behind CiaoPP is static global program analysis, based on abstract interpretation. For this task CiaoPP uses the PLAI abstract interpreter [MH92, BdIBH99], including extensions for, e.g., incrementality [HPMS00, PH96], modularity [BCHP96, PH00,

BdlBH⁺01], analysis of constraints [dlBHB⁺96b], and analysis of concurrency [MdlBH94].

The system includes several abstract analysis domains developed by several groups in the LP and CLP communities and can infer information on variable-level properties such as moded types, definiteness, freeness, independence, and grounding dependencies: essentially, precise data structure shape and pointer sharing. It can also infer bounds on data structure sizes, as well as procedure-level properties such as determinacy, termination, non-failure, and bounds on resource consumption (time or space cost). CiaoPP implements several techniques for dealing with “difficult” language features (such as side-effects, meta-programming, higher-order, etc.) and as a result can for example deal safely with arbitrary ISO-Prolog programs [BCHP96]. A unified language of assertions [BCHP96, PBH00b] is used to express the results of analysis, to provide input to the analyzer, and, as we will see later, to provide program specifications for debugging and validation, as well as the results of the comparisons performed against the specifications.

5.1 Modular Static Analysis Basics:

As mentioned before, CiaoPP takes advantage of modular program structure to perform more precise and efficient, incremental analysis. Consider the program in Figure 1, defining a module which exports the `qsort` predicate and imports predicates `geq` and `lt` from module `compare`. During the analysis of this program, CiaoPP will take advantage of the fact that the only predicate that can be called from outside is the *exported* predicate `qsort`. This allows CiaoPP to infer more precise information than if it had to consider that all predicates may be called in any possible way (as would be true had this been a simple “user” file instead of a module). Also, assume that the `compare` module has already been analyzed. This allows CiaoPP to be more efficient and/or precise, since it will use the information obtained for `geq` and `lt` during analysis of `compare` instead of either (re-)analyzing `compare` or assuming topmost substitutions for them. Assuming that `geq` and `lt` have a similar binding behavior as the standard comparison predicates, a mode and independence analysis (“sharing+freeness” [MH91]) of the module using CiaoPP yields the following results:⁴

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

⁴In the “sharing+freeness” domain `var` denotes variables that do not point yet to any data structure, `mshare` denotes pointer sharing patterns between variables. Derived properties `ground` and `indep` denote respectively variables which point to data structures which contain no pointers, and pairs of variables which point to data structures which do not share any pointers.

```

:- module(qsort, [qsort/2], [assertions]).
:- use_module(compare,[geq/2,lt/2]).

qsort([X|L],R) :-
    partition(L,X,L1,L2),
    qsort(L2,R2), qsort(L1,R1),
    append(R1,[X|R2],R).
qsort([],[]).

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
    lt(E,C), partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
    geq(E,C), partition(R,C,Left,Right1).

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

```

Figure 1: A modular qsort program.

```

:- true pred partition(A,B,C,D)
    : ( var(C), var(D), mshare([[A],[A,B],[B],[C],[D]]) )
    => ( ground(A), ground(C), ground(D), mshare([[B]]) ).
:- true pred append(A,B,C)
    : ( ground(A), mshare([[B],[B,C],[C]]) )
    => ( ground(A), mshare([[B,C]]) ).

```

These *assertions* express, for example, that the third and fourth arguments of `partition` have “output mode”: when `partition` is called (`:`) they are free unaliased variables and they are ground on success (`=>`). Also, `append` is used in a mode in which the first argument is input (i.e., ground on call). Also, upon success the arguments of `qsort` will share all variables (if any).

5.2 Assertions and Properties:

The above output is given in the form of CiaoPP *assertions*. These assertions are a means of specifying *properties* which are (or should be) true of a given predicate, predicate argument, and/or *program point*. If an assertion has been proved to be true it has a prefix `true` –like the ones

above. Assertions can also be used to provide information to the analyzer in order to increase its precision or to describe predicates which have not been coded yet during program development. These assertions have a `trust` prefix [BCHP96]. For example, if we commented out the `use_module/2` declaration in Figure 1, we could describe the mode of the (now missing) `geq` and `lt` predicates to the analyzer for example as follows:

```
:- trust pred geq(X,Y) => ( ground(X), ground(Y) ).
:- trust pred lt(X,Y)  => ( ground(X), ground(Y) ).
```

The same approach can be used if the predicates are written in, e.g., an external language such as, e.g., C or Java. Finally, assertions with a `check` prefix are the ones used to specify the *intended* semantics of the program, which can then be used in debugging and/or validation, as we will see in Section 6. Interestingly, this very general concept of assertions is also particularly useful for generating documentation automatically (see [Her00] for a description of their use by the Ciao auto-documenter).

Assertions refer to certain program points. The `true pred` assertions above specify in a combined way properties of both the entry (i.e., upon calling) and exit (i.e., upon success) points of *all calls* to the predicate. It is also possible to express properties which hold at points between clause literals. As an example of this, the following is a fragment of the output produced by CiaoPP for the program in Figure 1 when information is requested at this level:

```
qsort([X|L],R) :-
  true((ground(X),ground(L),var(R),var(L1),var(L2),var(R2), ...
  partition(L,X,L1,L2),
  true((ground(X),ground(L),ground(L1),ground(L2),var(R),var(R2), ...
  qsort(L2,R2), ...
```

In CiaoPP properties are just predicates, which may be builtin or user defined. For example, the property `var` used in the above examples is the standard builtin predicate to check for a free variable. The same applies to `ground` and `mshare`. The properties used by an analysis in its output (such as `var`, `ground`, and `mshare` for the previous mode analysis) are said to be *native* for that particular analysis. The system requires that properties be marked as such with a `prop` declaration which must be visible to the module in which the property is used. In addition, properties which are to be used in run-time checking (see later) should be defined by a (logic) program or system builtin, and also visible. Properties declared and/or defined in a module can be exported as any other predicate. For example:

```
:- prop list/1.
list([]).
list([_|L]) :- list(L).
```

or, using the functional syntax package, more compactly as:

```
:- prop list/1. list := [] | [_|list].
```

defines the property “list”. A list is an instance of a very useful class of user-defined properties called *regular types* [YS87, DZ92, GdW94, GP02b, VB02], which herein are simply a syntactically restricted class of logic programs. We can mark this fact by stating “:- regtype list/1.” instead of “:- prop list/1.” (this can be done automatically). The definition above can be included in a user program or, alternatively, it can be imported from a system library, e.g.:

```
:- use_module(library(lists),[list/1]).
```

5.3 Type Analysis:

CiaoPP can infer (parametric) types for programs both at the predicate level and at the literal level [GdW94, GP02b, VB02]. The output for Figure 1 at the predicate level, assuming that we have imported the `lists` library, is:

```
:- true pred qsort(A,B)
      : ( term(A), term(B) )
      => ( list(A), list(B) ).
:- true pred partition(A,B,C,D)
      : ( term(A), term(B), term(C), term(D) )
      => ( list(A), term(B), list(C), list(D) ).
:- true pred append(A,B,C)
      : ( list(A), list1(B,term), term(C) )
      => ( list(A), list1(B,term), list1(C,term) ).
```

where `term` is any term and `prop list1` is defined in `library(lists)` as:

```
:- regtype list1(L,T) # "@var{L} is a list of at least one @var{T}'s."
list1([X|R],T) :- T(X), list(R,T).
:- regtype list(L,T) # "@var{L} is a list of @var{T}'s."
list([],_T).
list([X|L],T) :- T(X), list(L).
```

We can use entry assertions [BCHP96] to specify a restricted class of calls to the module entry points as acceptable:

```
:- entry qsort(A,B) : (list(A, num), var(B)).
```

This informs the analyzer that in all external calls to `qsort`, the first argument will be a list of numbers and the second a free variable. Note the use of builtin properties (i.e., defined in modules which are loaded by default, such as `var`, `num`, `list`, etc.). Note also that properties natively understood by different analysis domains can be combined in the same assertion. This assertion will aid goal-dependent analyses obtain more accurate information. For example, it allows the type analysis to obtain the following, more precise information:

```
:- true pred qsort(A,B)
    : ( list(A,num), term(B) )
    => ( list(A,num), list(B,num) ).
:- true pred partition(A,B,C,D)
    : ( list(A,num), num(B), term(C), term(D) )
    => ( list(A,num), num(B), list(C,num), list(D,num) ).
:- true pred append(A,B,C)
    : ( list(A,num), list1(B,num), term(C) )
    => ( list(A,num), list1(B,num), list1(C,num) ).
```

5.4 Non-failure and Determinacy Analysis:

CiaoPP includes a non-failure analysis, based on [DLGH97] and [BLGH04], which can detect procedures and goals that can be guaranteed not to fail, i.e., to produce at least one solution or not terminate. It also can detect predicates that are “covered”, i.e., such that for any input (included in the calling type of the predicate), there is at least one clause whose “test” (head unification and body builtins) succeeds. CiaoPP also includes a determinacy analysis based on [LGBH04], which can detect predicates which produce at most one solution, or predicates whose clause tests are mutually exclusive, even if they are not deterministic (because they call other predicates that can produce more than one solution). For example, the result of these analyses for Figure 1 includes the following assertion:

```
:- true pred qsort(A,B)
    : ( list(A,num), var(B) ) => ( list(A,num), list(B,num) )
    + ( not_fails, covered, is_det, mut_exclusive ).
```

(The `+` field in `pred` assertions can contain a conjunction of global properties of the *computation* of the predicate.)

5.5 Size, Cost, and Termination Analysis:

CiaoPP can also infer lower and upper bounds on the sizes of terms and the computational cost of predicates [DLGHL94, DLGHL97]. The cost bounds are expressed as functions on the sizes

of the input arguments and yield the number of resolution steps. Various measures are used for the “size” of an input, such as list-length, term-size, term-depth, integer-value, etc. Note that obtaining a non-infinite upper bound on cost also implies proving *termination* of the predicate.

As an example, the following assertion is part of the output of the upper bounds analysis:

```
:- true pred append(A,B,C)
   : ( list(A,num), list1(B,num), var(C) )
  => ( list(A,num), list1(B,num), list1(C,num),
       size_ub(A,length(A)), size_ub(B,length(B)),
       size_ub(C,length(B)+length(A)) )
     + steps_ub(length(A)+1).
```

Note that in this example the size measure used is list length. The assertion `size_ub(C,length(B)+length(A))` means that an (upper) bound on the size of the third argument of `append/3` is the sum of the sizes of the first and second arguments. The inferred upper bound on computational steps is the length of the first argument of `append/3`.

The following is the output of the lower-bounds analysis:

```
:- true pred append(A,B,C)
   : ( list(A,num), list1(B,num), var(C) )
  => ( list(A,num), list1(B,num), list1(C,num),
       size_lb(A,length(A)), size_lb(B,length(B)),
       size_lb(C,length(B)+length(A)) )
     + ( not_fails, covered, steps_lb(length(A)+1) ).
```

The lower-bounds analysis uses information from the non-failure analysis, without which a trivial lower bound of 0 would be derived.

5.6 Decidability, Approximations, and Safety:

As a final note on the analyses, it should be pointed out that since most of the properties being inferred are in general undecidable at compile-time, the inference technique used, abstract interpretation, is necessarily *approximate*, i.e., possibly imprecise. On the other hand, such approximations are also always guaranteed to be safe, in the sense that (modulo bugs, of course) they are never *incorrect*.

6 Program Debugging and Assertion Validation

CiaoPP is also capable of combined static and dynamic validation, and debugging using the ideas outlined so far. To this end, it implements the framework described in [HPB99, PBH00a] which

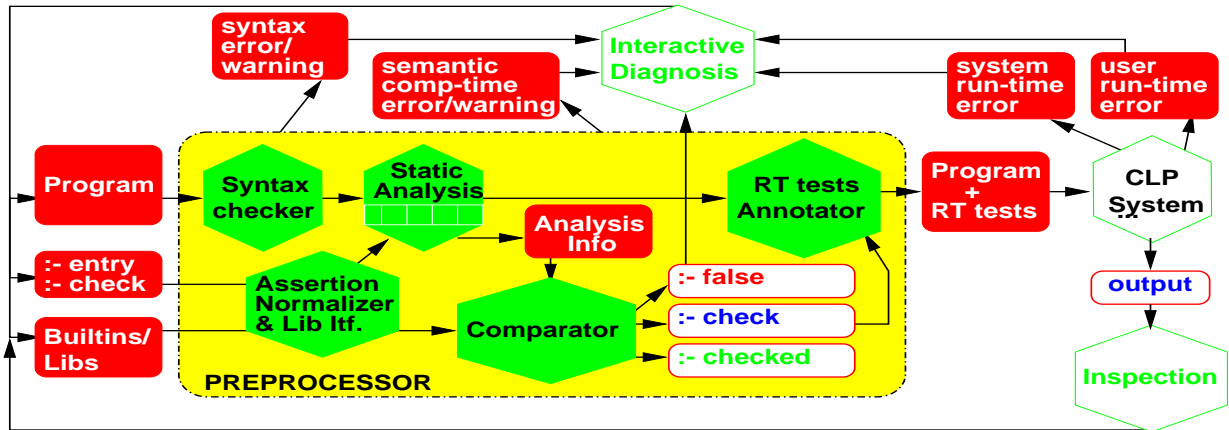


Figure 2: Architecture of the Preprocessor

involves several of the tools which comprise CiaoPP. Figure 2 depicts the overall architecture. Hexagons represent the different tools involved and arrows indicate the communication paths among them.

Program verification and detection of errors is first performed at compile-time by using the sufficient conditions shown in Table 2, i.e., by inferring properties of the program via abstract interpretation-based static analysis and comparing this information against (partial) specifications I_α written in terms of assertions.

Both the static and the dynamic checking are provably *safe* in the sense that all errors flagged are definite violations of the specifications.

6.1 Static Debugging:

The idea of using analysis information for debugging comes naturally after observing analysis outputs for erroneous programs. Consider the program in Figure 3. The result of regular type analysis for this program includes the following code:

```

:- module(qsort, [qsort/2], [assertions]).
:- entry qsort(A,B) : (list(A, num), var(B)).

qsort([X|L],R) :-
    partition(L,L1,X,L2),
    qsort(L2,R2), qsort(L1,R1),
    append(R2,[x|R1],R).
qsort([],[]).

partition([],_B,[],[]).
partition([e|R],C,[E|Left1],Right):-
    E < C, !, partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
    E >= C, partition(R,C,Left,Right1).

append([],X,X).
append([H|X],Y,[H|Z]):- append(X,Y,Z).

```

Figure 3: A tentative qsort program.

```

:- true pred qsort(A,B)
    : ( term(A), term(B) )
    => ( list(A,t113), list(B,^x) ).

:- regtype t113/1.
t113(A) :- arithexpression(A).
t113([]).
t113([A|B]) :- arithexpression(A), list(B,t113).
t113(e).

```

where `arithexpression` is a library property which describes arithmetic expressions and `list(B, ^x)` means “a list of x’s.” A new name (`t113`) is given to one of the inferred types, and its definition included, because no definition of this type was found visible to the module. In any case, the information inferred does not seem compatible with a correct definition of `qsort`, which clearly points to a bug in the program.

6.2 Static Checking of Assertions in System Libraries:

In addition to manual inspection of the analyzer output, CiaoPP includes a number of automated facilities to help in the debugging task. For example, CiaoPP can find incompatibilities between the ways in which library predicates are called and their intended mode of use, expressed in the form of assertions in the libraries themselves. Also, the preprocessor can detect inconsistencies in the program and check the assertions present in other modules used by the program.

For example, turning on compile-time error checking and selecting type and mode analysis for our tentative `qsort` program in Figure 3 we obtain the following messages:

```
WARNING: Literal partition(L,L1,X,L2) at qsort/2/1/1 does not succeed!
ERROR: Predicate E>=C at partition/4/3/1 is not called as expected:
      Called:   num>=var
      Expected: arithexpression>=arithexpression
```

where `qsort/2/1/1` stands for the first literal in the first clause of `qsort` and `partition/4/3/1` stands for the first literal in the third clause of `partition`.⁵

The first message warns that all calls to `partition` will fail, something normally not intended (e.g., in our case). The second message indicates a wrong call to a builtin predicate, which is an obvious error. This error has been detected by comparing the mode information obtained by global analysis, which at the corresponding program point indicates that `E` is a free variable, with the assertion:

```
:- check calls A<B (arithexpression(A), arithexpression(B)).
```

which is present in the default `builtins` module, and which implies that the two arguments to `</2` should be ground. The message signals a compile-time, or *abstract*, incorrectness symptom [BDD⁺97], indicating that the program does not satisfy the specification given (that of the builtin predicates, in this case). Checking the indicated call to `partition` and inspecting its arguments we detect that in the definition of `qsort`, `partition` is called with the second and third arguments in reversed order – the correct call is `partition(L, X, L1, L2)`.

After correcting this bug, we proceed to perform another round of compile-time checking, which produces the following message:

```
WARNING: Clause 'partition/4/2' is incompatible with its call type
      Head:   partition([e|R],C,[E|Left1],Right)
      Call Type: partition(list(num),num,var,var)
```

⁵In the actual system line numbers and automated location of errors in source files are provided.

This time the error is in the second clause of `partition`. Checking this clause we see that in the first argument of the head there is an `e` which should be `E` instead. Compile-time checking of the program with this bug corrected does not produce any further warning or error messages.

6.3 Static Checking of User Assertions and Program Validation:

Though, as seen above, it is often possible to detect error without adding assertions to user programs, if the program is not correct, the more assertions are present in the program the more likely it is for errors to be automatically detected. Thus, for those parts of the program which are potentially buggy or for parts whose correctness is crucial, the programmer may decide to invest more time in writing assertions than for other parts of the program which are more stable. In order to be more confident about our program, we add to it the following check assertions:⁶

```
:- calls qsort(A,B) : list(A, num). % A1
:- success qsort(A,B) => (ground(B), sorted_num_list(B)). % A2
:- calls partition(A,B,C,D) : (ground(A), ground(B)). % A3
:- success partition(A,B,C,D) => (list(C, num),ground(D)). % A4
:- calls append(A,B,C) : (list(A,num),list(B,num)). % A5
:- comp partition/4 + not_fails. % A6
:- comp partition/4 + is_det. % A7
:- comp partition(A,B,C,D) + terminates. % A8

:- prop sorted_num_list/1.
sorted_num_list([]).
sorted_num_list([X]):- number(X).
sorted_num_list([X,Y|Z]):-
    number(X), number(Y), X=<Y, sorted_num_list([Y|Z]).
```

where we also use a new property, `sorted_num_list`, defined in the module itself. These assertions provide a partial specification of the program. They can be seen as integrity constraints: if their properties do not hold at the corresponding program points (procedure call, procedure exit, etc.), the program is incorrect. `Calls` assertions specify properties of all calls to a predicate, while `success` assertions specify properties of exit points for all calls to a predicate. Properties of successes can be restricted to apply only to calls satisfying certain properties upon entry by adding a “:” field to `success` assertions. Finally, `Comp` assertions specify *global* properties of the execution of a predicate. These include complex properties such as determinacy or termination and are in general not amenable to run-time checking. They can also be

⁶The check prefix is assumed when no prefix is given, as in the example shown.

restricted to a subset of the calls using “:”. More details on the assertion language can be found in [PBH00b].

CiaoPP can perform compile-time checking of the assertions above, by comparing them with the assertions inferred by analysis (see Table 2 and [BDD⁺97, PBH00c] for details), producing as output the following assertions (refer also to Figure 2, output of the comparator):

```
:- checked calls qsort(A,B) : list(A,num). % A1
:- check success qsort(A,B) => sorted_num_list(B). % A2
:- checked calls partition(A,B,C,D) : (ground(A),ground(B)). % A3
:- checked success partition(A,B,C,D) => (list(C,num),ground(D) ).% A4
:- false calls append(A,B,C) : ( list(A,num), list(B,num) ). % A5
:- checked comp partition/4 + not_fails. % A6
:- checked comp partition/4 + is_det. % A7
:- checked comp partition/4 + terminates. % A8
```

Note that a number of initial assertions have been marked as *checked*, i.e., they have been *validated*. If all assertions had been moved to this *checked* status, the program would have been *verified*. In these cases CiaoPP is capable of generating certificates which can be checked efficiently for, e.g., mobile code applications [APH04]. However, in our case assertion A5 has been detected to be false. This indicates a violation of the specification given, which is also flagged by CiaoPP as follows:

```
ERROR: (lns 22-23) false calls assertion:
  :- calls append(A,B,C) : list(A,num),list(B,num)
     Called append(list(^x),[^x|list(^x)],var)
```

The error is now in the call `append(R2, [x|R1],R)` in `qsort` (`x` instead of `X`). Assertions A1, A3, A4, A6, A7, and A8 have been detected to hold, but it was not possible to prove statically assertion A2, which has remained with `check` status. Note that though the predicate `partition` may fail in general, in the context of the current program it can be proved not to fail. Note also that A2 has been simplified, and this is because the mode analysis has determined that on success the second argument of `qsort` is ground, and thus this does not have to be checked at run-time. On the other hand the analyses used in our session (types, modes, non-failure, determinism, and upper-bound cost analysis) do not provide enough information to prove that the output of `qsort` is a *sorted* list of numbers, since this is not a native property of the analyses being used. While this property could be captured by including a more refined domain (such as constrained types), it is interesting to see what happens with the analyses selected for the example.⁷

⁷Note that while property `sorted_num_list` cannot be proved with only (over approximations) of mode and regular type information, it may be possible to prove that it does *not* hold (an example of how properties which are

6.4 Dynamic Debugging with Run-time Checks:

Assuming that we stay with the analyses selected previously, the following step in the development process is to compile the program obtained above with the “generate run-time checks” option. CiaoPP will then introduce run-time tests in the program for those `calls` and `success` assertions which have not been proved nor disproved during compile-time (see again Figure 2). In our case, the program with run-time checks will call the definition of `sorted_num_list` at the appropriate times. In the current implementation of CiaoPP we obtain the following code for predicate `qsort` (the code for `partition` and `append` remain the same as there is no other assertion left to check):

```
qsort(A,B) :-
    new_qsort(A,B),
    postc([ qsort(C,D) : true => sorted(D) ], qsort(A,B)).

new_qsort([X|L],R) :-
    partition(L,X,L1,L2),
    qsort(L2,R2), qsort(L1,R1),
    append(R2,[X|R1],R).
new_qsort([],[]).
```

where `postc` is the library predicate in charge of checking postconditions of predicates. If we now run the program with run-time checks in order to sort, say, the list `[1,2]`, the Ciao system generates the following error message:

```
?- qsort([1,2],L).
ERROR: for Goal qsort([1,2],[2,1])
Precondition: true holds, but
Postcondition: sorted_num_list([2,1]) does not.

L = [2,1] ?
```

Clearly, there is a problem with `qsort`, since `[2,1]` is not the result of ordering `[1,2]` in ascending order. This is a (now, run-time, or *concrete*) incorrectness symptom, which can be used as the starting point of diagnosis. The result of such diagnosis should indicate that the call to `append` (where `R1` and `R2` have been swapped) is the cause of the error and that the right definition of predicate `qsort` is the one in Figure 1.

not natively understood by the analysis can also be useful for detecting bugs at compile-time): while the regular type analysis cannot capture perfectly the property `sorted_num_list`, it can still approximate it (by analyzing the definition) as `list(B, num)`. If type analysis for the program were to generate a type for `B` not compatible with `list(B, num)`, then a definite error symptom would be detected.

```

:- module(reverse, [nrev/2], [assertions]).
:- use_module(library('assertions/native_props')).
:- entry nrev(A,B) : (ground(A), list(A, term), var(B)).

nrev([],[]).
nrev([H|L],R) :-
    nrev(L,R1),
    append(R1,[H],R).

```

Figure 4: The naive reverse program.

6.5 Performance Debugging and Validation:

Another very interesting feature of CiaoPP is the possibility of stating assertions about the efficiency of the program which the system will try to verify or falsify. This is done by stating lower and/or upper bounds on the computational cost of predicates (given in number of execution steps). Consider for example the naive reverse program in Figure 4. Assume also that the predicate `append` is defined as in Figure 1.

Suppose that the programmer thinks that the cost of `nrev` is given by a linear function on the size (list-length) of its first argument, maybe because he has not taken into account the cost of the `append` call). Since `append` is linear, it causes `nrev` to be quadratic. We will show that CiaoPP can be used to inform the programmer about this false idea about the cost of `nrev`. For example, suppose that the programmer adds the following “check” assertion:

```
:- check comp nrev(A,B) + steps_ub(length(A)+1).
```

With compile-time error checking turned on, and mode, type, non-failure and lower-bound cost analysis selected, we get the following error message:

```

ERROR: false comp assertion:
      :- comp nrev(A,B) : true => steps_ub(length(A)+1)
      because in the computation the following holds:
      steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1)

```

This message states that `nrev` will take at least $0.5 (\text{length}(A))^2 + 1.5 \text{length}(A) + 1$ resolution steps (which is the cost analysis output), while the assertion requires that it take at most $\text{length}(A) + 1$ resolution steps. The cost function in the user-provided assertion is compared with the lower-bound cost assertion inferred by analysis. This allows detecting the inconsistency

and proving that the program does not satisfy the efficiency requirements imposed. Upper-bound cost assertions can also be proved to hold, i.e., can be *checked*, by using upper-bound cost analysis rather than lower-bound cost analysis. In such case, if the upper-bound computed by analysis is lower or equal than the upper-bound stated by the user in the assertion. The converse holds for lower-bound cost assertions. Thanks to this functionality, CiaoPP can certify programs with resource consumption assurances and also efficiently check such certificates [HALGP04].

7 Source Program Optimization

We now turn our attention to the program optimizations that are available in CiaoPP. These include abstract specialization, parallelization (including granularity control), multiple program specialization, and integration of abstract interpretation and partial evaluation. All of them are performed as source to source transformations of the program. In most of them static analysis is instrumental, or, at least, beneficial.

7.1 Abstract Specialization:

Program specialization optimizes programs for known values (substitutions) of the input. It is often the case that the set of possible input values is unknown, or this set is infinite. However, a form of specialization can still be performed in such cases by means of abstract interpretation, specialization then being with respect to abstract values, rather than concrete ones. Such abstract values represent a (possibly infinite) set of concrete values. For example, consider the definition of the property `sorted_num_list/1`, and assume that regular type analysis has produced:

```
:- true pred sorted_num_list(A) : list(A,num) => list(A,num).
```

Abstract specialization can use this information to optimize the code into:

```
sorted_num_list([]).
sorted_num_list([_]).
sorted_num_list([X,Y|Z]):- X=<Y, sorted_num_list([Y|Z]).
```

which is clearly more efficient because no `number` tests are executed. The optimization above is based on abstractly executing the `number` literals to the value `true`, as discussed in Section 4.2.

CiaoPP can also apply abstract specialization to the optimization of programs with dynamic scheduling (e.g., using `delay` declarations) [PdIBMS97]. The transformations simplify the conditions on the *delay declarations* and also move delayed literals later in the rule body, leading to substantial performance improvement. This is used by CiaoPP, for example, when supporting complex computation models, such as Andorra-style execution [HBC⁺99].

7.2 Parallelization:

An example of a non-trivial program optimization performed using abstract interpretation in CiaoPP is program parallelization [BdlBH99]. It is also performed as a source-to-source transformation, in which the input program is *annotated* with parallel expressions. The parallelization algorithms, or annotators [MBdlBH99], exploit parallelism under certain *independence* conditions, which allow guaranteeing interesting correctness and no-slowdown properties for the parallelized programs [HR95, dlBHM00]. This process is complicated by the presence of shared variables and pointers among data structures at run-time.

We consider again the program of Figure 1. A possible parallelization (obtained in this case with the “MEL” annotator) is:

```
qsort([X|L],R) :-  
    partition(L,X,L1,L2),  
    ( indep([[L1,L2]]) -> qsort(L2,R2) & qsort(L1,R1)  
      ; qsort(L2,R2), qsort(L1,R1) ),  
    append(R1,[X|R2],R).
```

which indicates that, provided that L1 and L2 do not have variables in common (at execution time), then the recursive calls to `qsort` can be run in parallel. Given the information inferred by the abstract interpreter using, e.g., the mode and independence analysis (see Section 5), which determines that L1 and L2 are ground after `partition` (and therefore do not share variables), the independence test and the conditional can be simplified via abstract executability and the annotator yields instead:

```
qsort([X|L],R) :-  
    partition(L,X,L1,L2),  
    qsort(L2,R2) & qsort(L1,R1),  
    append(R1,[X|R2],R).
```

which is much more efficient since it has no run-time test. This test simplification process is described in detail in [BdlBH99] where the impact of abstract interpretation in the effectiveness of the resulting parallel expressions is also studied.

The tests in the above example aim at *strict* independent and-parallelism. However, the annotators are parameterized on the notion of independence. Different tests can be used for different independence notions: non-strict independence [CH94], constraint-based independence [dlBHM00], etc. Moreover, all forms of and-parallelism in logic programs can be seen as independent and-parallelism, provided the definition of independence is applied at the appropriate granularity

level.⁸

7.3 Resource and Granularity Control:

Another application of the information produced by the CiaoPP analyzers, in this case cost analysis, is to perform combined compile-time/run-time resource control. An example of this is task granularity control [LGHD96] of parallelized code. Such parallel code can be the output of the process mentioned above or code parallelized manually.

In general, this run-time granularity control process involves computing sizes of terms involved in granularity control, evaluating cost functions, and comparing the result with a threshold⁹ to decide for parallel or sequential execution. Optimizations to this general process include cost function simplification and improved term size computation, both of which are illustrated in the following example.

Consider again the `qsort` program in Figure 1. We use CiaoPP to perform a transformation for granularity control, using the analysis information of type, sharing+freeness, and upper bound cost analysis, and taking as input the parallelized code obtained in the previous section. CiaoPP adds a clause:

“`qsort(_1,_2) :- g_qsort(_1,_2).`” (to preserve the original entry point) and produces `g_qsort/2`, the version of `qsort/2` that performs granularity control (`s_qsort/2` is the sequential version):

```
g_qsort([X|L],R) :-
    partition_o3_4(L,X,L1,L2,_1,_2),
    ( _2>7 -> (_1>7 -> g_qsort(L2,R2) & g_qsort(L1,R1)
                ; g_qsort(L2,R2), s_qsort(L1,R1))
      ; (_1>7 -> s_qsort(L2,R2), g_qsort(L1,R1)
                ; s_qsort(L2,R2), s_qsort(L1,R1))),
    append(R1,[X|R2],R).
g_qsort([],[]).
```

Note that if the lengths of the two input lists to the `qsort` program are greater than a threshold (a list length of 7 in this case) then versions which continue performing granularity control are executed in parallel. Otherwise, the two recursive calls are executed sequentially. The executed version of each of such calls depends on its grain size: if the length of its input list is not

⁸For example, stream and-parallelism can be seen as independent and-parallelism if the independence of “bindings” rather than goals is considered.

⁹This threshold can be determined experimentally for each parallel system, by taking the average value resulting from several runs.

greater than the threshold then a sequential version which does not perform granularity control is executed. This is based on the detection of a recursive invariant: in subsequent recursions this goal will not produce tasks with input sizes greater than the threshold, and thus, for all of them, execution should be performed sequentially and, obviously, no granularity control is needed.

In general, the evaluation of the condition to decide which predicate versions are executed will require the computation of cost functions and a comparison with a cost threshold (measured in units of computation). However, in this example a test simplification has been performed, so that the input size is simply compared against a size threshold, and thus the cost function for `qsort` does not need to be evaluated.¹⁰ Predicate `partition_o3_4/6`:

```
partition_o3_4([],_B,[],[],0,0).
partition_o3_4([E|R],C,[E|Left1],Right,_1,_2) :-
    E<C, partition_o3_4(R,C,Left1,Right,_3,_2), _1 is _3+1.
partition_o3_4([E|R],C,Left,[E|Right1],_1,_2) :-
    E>=C, partition_o3_4(R,C,Left,Right1,_1,_3), _2 is _3+1.
```

is the transformed version of `partition/4`, which “on the fly” computes the sizes of its third and fourth arguments (the automatically generated variables `_1` and `_2` represent these sizes respectively) [LGH95].

7.4 Multiple Specialization:

Sometimes a procedure has different uses within a program, i.e. it is called from different places in the program with different (abstract) input values. In principle, (abstract) program specialization is then allowable only if the optimization is applicable to all uses of the predicate. However, it is possible that in several different uses the input values allow different and incompatible optimizations and then none of them can take place. In CiaoPP this problem is overcome by means of “multiple program specialization” where different versions of the predicate are generated for each use. Each version is then optimized for the particular subset of input values with which it is to be used. The abstract multiple specialization technique used in CiaoPP [PH99] has the advantage that it can be incorporated with little or no modification of some existing abstract interpreters, provided they are *multivariant* (PLAI and similar frameworks have this property).

This specialization can be used for example to improve automatic parallelization in those cases where run-time tests are included in the resulting program. In such cases, a good number of run-time tests may be eliminated and invariants extracted automatically from loops, resulting generally in lower overheads and in several cases in increased speedups. We consider automatic

¹⁰This size threshold will obviously be different if the cost function is.

parallelization of a program for matrix multiplication using the same analysis and parallelization algorithms as the `qsort` example used before. This program is automatically parallelized without tests if we provide the analyzer (by means of an `entry` declaration) with accurate information on the expected modes of use of the program. However, in the interesting case in which the user does not provide such declaration, the code generated contains a large number of run-time tests. We include below the code for predicate `multiply` which multiplies a matrix by a vector:

```
multiply([],_,[]).
multiply([V0|Rest],V1,[Result|Others]) :-
    (ground(V1),
     indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
     vmul(V0,V1,Result) & multiply(Rest,V1,Others)
    ; vmul(V0,V1,Result), multiply(Rest,V1,Others)).
```

Four independence tests and one groundness test have to be executed prior to executing in parallel the calls in the body of the recursive clause of `multiply` (these tests essentially check that the arrays do not contain pointers that point in such a way that would make the `vmul` and `multiply` calls be dependent). However, abstract multiple specialization generates four versions of the predicate `multiply` which correspond to the different ways this predicate may be called (basically, depending on whether the tests succeed or not). Of these four variants, the most optimized one is:

```
multiply3([],_,[]).
multiply3([V0|Rest],V1,[Result|Others]) :-
    (indep([[Result,Others]]) ->
     vmul(V0,V1,Result) & multiply3(Rest,V1,Others)
    ; vmul(V0,V1,Result), multiply3(Rest,V1,Others)).
```

where the groundness test and three out of the four independence tests have been eliminated. Note also that the recursive calls to `multiply` use the optimized version `multiply3`. Thus, execution of matrix multiplication with the expected mode (the only one which will succeed in Prolog) will be quickly directed to the optimized versions of the predicates and iterate on them. This is because the specializer has been able to detect this optimization as an invariant of the loop. The complete code for this example can be found in [PH99]. The multiple specialization implemented incorporates a minimization algorithm which keeps in the final program as few versions as possible while not losing opportunities for optimization. For example, eight versions of predicate `vmul` (for vector multiplication) would be generated if no minimizations were performed. However, as multiple versions do not allow further optimization, only one version is present in the final program.

7.5 Integration of Abstract Interpretation and Partial Evaluation:

In the context of CiaoPP we have also studied the relationship between abstract multiple specialization, abstract interpretation, and partial evaluation. Abstract specialization exploits the information obtained by multivariant abstract interpretation where information about values of variables is propagated by simulating program execution and performing fixpoint computations for recursive calls. In contrast, traditional partial evaluators (mainly) use unfolding for both propagating values of variables and transforming the program. It is known that abstract interpretation is a better technique for propagating success values than unfolding. However, the program transformations induced by unfolding may lead to important optimizations which are not directly achievable in the existing frameworks for multiple specialization based on abstract interpretation. In [PHG99] we present a specialization framework which integrates the better information propagation of abstract interpretation with the powerful program transformations performed by partial evaluation.

We are currently investigating the use of abstract domains based on improvements of regular types [VB02] for their use for partial evaluation.

Part II

A Tutorial Overview of the PyLogen System

The PYLOGEN system is an implemented tool for specialising Prolog programs. The specialisation engine is written in SICStus Prolog and the interface is a mixture of Python and Tk. This section will explain the basic functionality through a simple tutorial.

8 Starting PYLOGEN

Follow the online instructions for installing PYLOGEN . To start PYLOGEN :

- OS X:
`[~] pythonw logen.py`
- Windows and Linux:
`[~] python logen.py`

Regular Expression Example

For this tutorial we use a simple regular expression parser (Listing 1). The interpreter takes a basic regular expression and a string (represented by a list of atoms) and succeeds if the string matches the regular expression (Listing 2). The empty pattern, ϵ , is represented by the special constant `eps`.

Listing 1: An interpreter for regular expressions

```
match(Regexp, String) :- regexp(Regexp, String, []).
```

```
regexp(eps, T, T).
```

```
regexp(X, [X|T], T) :- atomic(X).
```

```
regexp(+(_A, B), Str, DStr) :- regexp(B, Str, DStr).
```

```
regexp(+A, Str, DStr) :- regexp(A, Str, DStr).
```

```
regexp(. (A, B), Str, DStr) :- regexp(A, Str, I), regexp(B, I, DStr).
```

```
regexp(*(A), S, DS) :- regexp(. (A, *(A)), S, DS).
```

```
regexp(*(A), S, S).
```

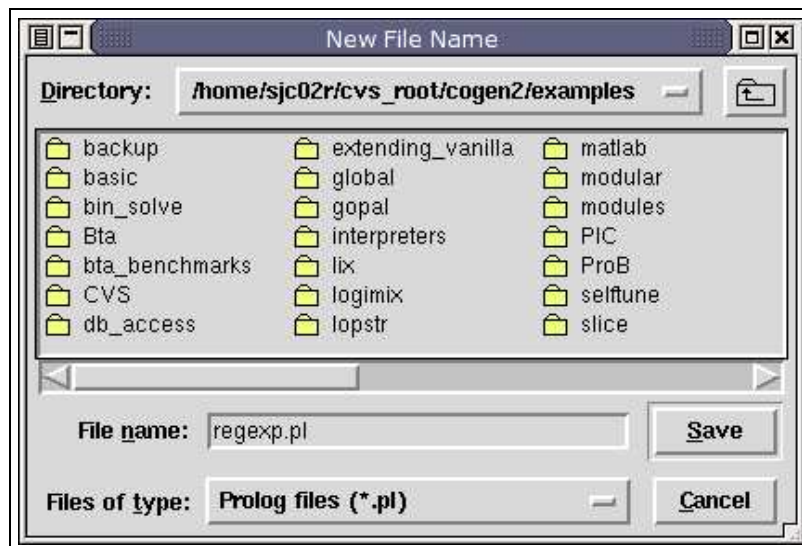
Listing 2: Using the regular expression interpreter

```
| ? - match (.(*(a),b) , [ a , a , a , b ] ).  
yes  
% source_info  
| ? - match (.(*(a),b) , [ a , a , a , b , c ] ).  
no
```

9 Specialising the Regular Expression Interpreter

Create a new file

Click on the **new** icon or select **new** from from the **File** menu. In the dialogbox select a location for the new file and call it *regexp.pl*.



Edit the new file

The default mode in PYLOGEN edits the annotations associated with the current source code. The top left pane contains the sourcecode, the top right pane contains the filter declarations and the lower pane displays the different output modes. To actually edit the sourcecode we must first enter **sourcecode mode**. Click on the **edit** icon or select **sourcecode mode** from the **Edit** menu.

Once in **sourcecode mode** add the sourcecode from Listing 1 into the top left pane. When you have finished typing entering the sourcecode click the **save** icon or select **annotation mode** from

the **Edit** menu. If there is a parse error you will be notified by an error message, if everything is correct the source code will be reloaded and annotated using the **unknown** annotation.

```

Source
/* Created by Pylogen */
match(Regexp,String) :- regexp(Regexp,String,[]).

regexp(eps,T,T).
regexp(X,[X|_],T) :- atomic(X).
regexp(+_A_B,Str,DStr) :- regexp(A,Str,DStr).
regexp(+_A_B,Str,DStr) :- regexp(B,Str,DStr).
regexp(.(A,B),Str,DStr) :- regexp(A,Str,I), regexp(B,I,DStr).
regexp(*_A,S,DS) :- regexp(.(A,*_A),S,DS).
regexp(*_A,S,S).

```

Annotate the new file

The **unknown** annotation is used to identify unannotated calls in the program. To specialise the regular expression interpreter we must first properly annotate the program. We assume that the regular expression will be known at specialisation time, **static**, but the string to match against will be **dynamic**.

The predicate `match/2` is an entry point into the regular expression interpreter, it simply calls `regexp/3` with an empty list as the third argument. The third argument contains the "left over" part of the string, so `match/2` only succeeds on an exact match. We choose to **unfold** the call to `regexp/3`, clicking on the call will display the annotation menu. Select **unfold** from the menu to annotate this call.

```

Source
/* Created by Pylogen */
match(Regexp,String) :- regexp(Regexp,String,[]).

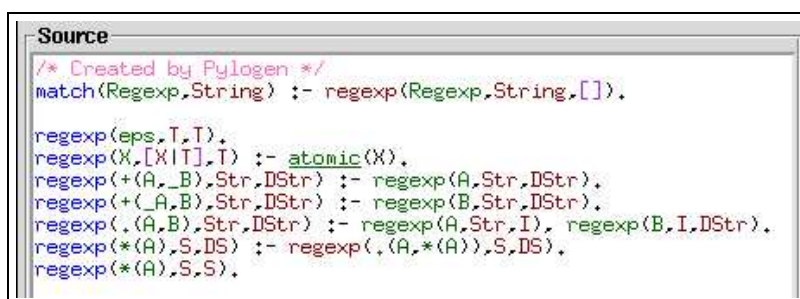
regexp(eps,T,T).
regexp(X,[X|_],T) :- atomic(X).
regexp(+_A_B,Str,DStr) :- regexp(A,Str,DStr).
regexp(+_A_B,Str,DStr) :- regexp(B,Str,DStr).
regexp(.(A,B),Str,DStr) :- regexp(A,Str,I), regexp(B,I,DStr).
regexp(*_A,S,DS) :- regexp(.(A,*_A),S,DS).
regexp(*_A,S,S).

```

Now we move onto annotate the `regexp/3` predicate. The first call is to the built-in predicate `atomic/1`. If we have an atomic item in our pattern then we simply look for that item in the input string. As the first argument is **static** (it was passed directly from `match/2`), we

can safely make this call at specialisation time. Mark the call to `atomic/1` as `call`, again by clicking on the call and selecting `call`.

The remaining calls are all recursive calls to the `regexp/3` predicate. The annotations in a program ensure it will terminate at specialisation time. When annotating a program by hand it is important to keep in mind which calls are safe to **unfold** and which must be marked **memo**. In the case of the regular expression interpreter we know the pattern is **static**, so as long as we are decreasing the pattern each call we are going to eventually terminate. Inspecting the clauses shows that the only unsafe call is in handling of the `*(Pattern)`, this allows an unbounded number of matches against `Pattern`. As we do not have the string to match against we must mark the recursive call to `regexp(.(A,*(A)),S,DS)` as **memo**. The rest of the calls can be marked **unfold**.



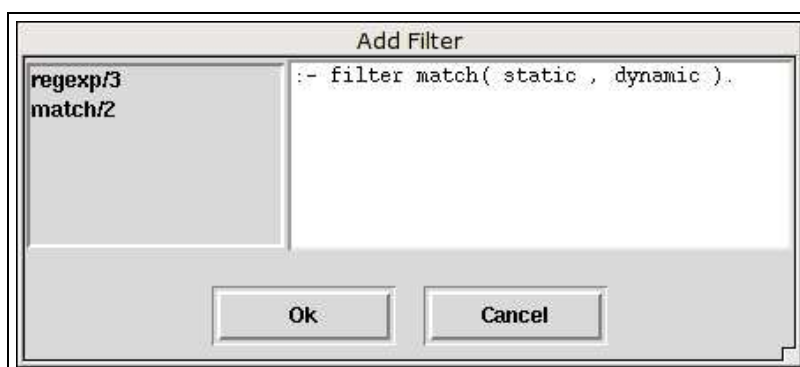
```
Source
/* Created by Pylogen */
match(Regexp,String) :- regexp(Regexp,String,[]).

regexp(eps,T,T).
regexp(X,[X|_],T) :- atomic(X).
regexp(+(_A,_B),Str,DStr) :- regexp(A,Str,DStr).
regexp(+(_A,_B),Str,DStr) :- regexp(B,Str,DStr).
regexp(.(A,B),Str,DStr) :- regexp(A,Str,I), regexp(B,I,DStr).
regexp(*(A),S,DS) :- regexp(.(A,*(A)),S,DS),
regexp(*(A),S,S).

```

Add an entry point

We have now annotated all of the clauses in the regular expression program. Now we must tell the specialiser something about the entry point of the program. We intend to call `match/2` with a **static** first argument and a **dynamic** second argument. Click the **insert filter** icon or select **insert filter** from the **Edit** menu.



The left hand side contains a list of predicates appearing in the source program. Double

click on `match/2` to create an empty filter declaration. Change the declaration to make the first argument **static**.

```
:- filter match(static , dynamic).
```

Filter Propagation

As a call to `regexp/3` is marked as **memo** we will also need to provide a filter declaration for `regexp/3`. This can be done manually, inferring that `regexp/3` is **static, dynamic, dynamic** from the initial call in `match/2`. We can also use the filter propagation discussed in Lopstr 2004, BTA paper. Save the file and select **propagate filters** from the **BTA** menu.



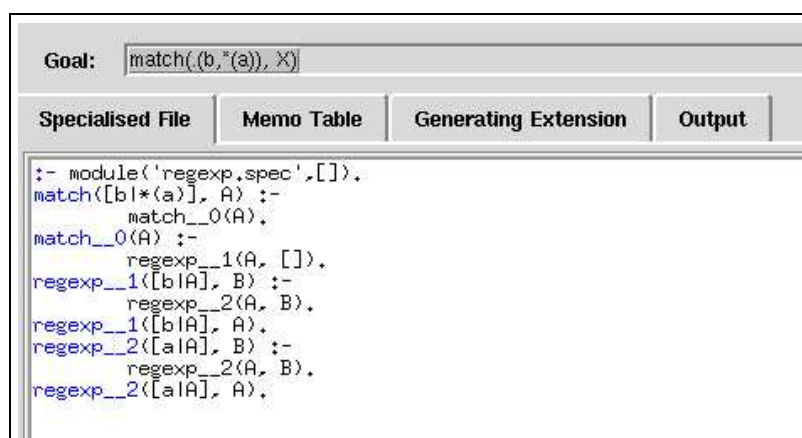
```
Declarations
/* Filter Declarations */
:- filter
    match(static, dynamic).
:- filter
    regexp(static, dynamic, dynamic).
```

Specialising the regular expression interpreter

Now we have annotated the interpreter we can specialise it for different regular expressions. Save the file and enter a specialisation query in the **Goal** entry box.

```
match(. ( b , *( a ) ) , X)
```

This will specialise the interpreter for matching a string beginning with a `b` followed by zero or more `a`'s. Click **Specialise** or press return to specialise the program.



```
Goal: match((b,*(a)), X)

Specialised File Memo Table Generating Extension Output

:- module('regexp.spec', []).
match([b]*(a), A) :-
    match__0(A).
match__0(A) :-
    regexp__1(A, []).
regexp__1([b]A, B) :-
    regexp__2(A, B).
regexp__1([a]A, A).
regexp__2([a]A, B) :-
    regexp__2(A, B).
regexp__2([a]A, A).
```


The specialise code contains an entry point `match([b|*(a)], A)` which will call the corresponding specialised predicate. The overhead of interpreting the regular expression has been removed and only the string matcher remains.

The memo table maintains the list of specialised predicates and their original call patterns. It is used internally during specialisation and is saved to a file when specialisation is complete. Selecting the **Memo Table** tab displays the table.

```

Specialised File  Memo Table  Generating Extension  Output
-----
gensym(3),
table(match([b|*(a)],A), match__0(A), [crossmodule]),
table(regex([b|*(a)],A,B), regex__1(A,B), []),
table(regex([a|*(a)],A,B), regex__2(A,B), []).

```

The two entries for `regex/3` correspond to the two specialised versions of `regex/3` generated during specialisation, called `regex__1` and `regex__2`. `regex__1` is specialised for a `b` followed by some `a`'s, and `regex__2` is specialised for an `a` followed by some more `a`'s. Hovering over a call in the specialised file displays the original mapping from the memo table in a balloon window.

```

regex__2([a|A], A),
Stat regex([a|*(a)],A,B) --> regex__2(A,B)

```

A *cogen* specialiser first creates a generating extension, a specialised specialiser, which is then used to specialise a file for a particular query. Clicking on the **generating extension** tab will display this file. The generating extension only needs to be regenerated if the annotations change, it can be reused for different specialisation queries.

10 Using the Automatic Binding-time Analysis

In the last section we annotated the file by hand, manually checking each annotation. The LOP-STR'04 BTA paper introduces the automatic binding-time analysis (bta). The bta automatically annotates a file with a correct set of annotations. From the **BTA** menu select **unfold all**, this will reset the file, annotating it to perform *all* of the operations at specialisation time. Now add an entry point for the bta, this is done using a filter declaration.

```
:- filter match(static, dynamic).
```

Specialised File	Memo Table	Generating Extension	Output
<pre> match_u(A, B, C) :- regexp_request(A, B, [], internal, C), regexp_u(eps, A, A, true), regexp_u(A, [A B], B, true) :- atomic(A), regexp_u(A+_, B, C, D) :- regexp_u(A, B, C, D), regexp_u(_+A, B, C, D) :- regexp_u(A, B, C, D), regexp_u([A B], C, D, (E,F)) :- regexp_u(A, C, G, E), regexp_u(B, G, D, F), regexp_u(*(A), B, C, D) :- </pre>			
Status:			

The regular expression interpreter manipulates terms as it parses the regular expression. Select **List Norm** from the **BTA** menu. Save the file and then select **Auto bta** from the **BTA** menu.

The bta should provide the same annotations we selected manually. Only the recursive call to `regexp/3` handling the `*` will be marked as **memo**.

```

Source
/* Created by Pylogen */
match(Regexp,String) :- regexp(Regexp,String,[]),

regexp(eps,T,T),
regexp(X,[X|T],T) :- atomic(X),
regexp(+_A_B,Str,DStr) :- regexp(A,Str,DStr),
regexp(+_A_B,Str,DStr) :- regexp(B,Str,DStr),
regexp(_(A,B),Str,DStr) :- regexp(A,Str,I), regexp(B,I,DStr),
regexp(*(A),S,DS) :- regexp(_(A,*(A)),S,DS),
regexp(*(A),S,S),

```

The filter declarations should be correctly propagated throughout the program.

```

Declarations
/* Filter Declarations */
:- filter
    match(static, dynamic),
:- filter
    regexp(static, dynamic, dynamic),

```

References

- [APH04] E. Albert, G. Puebla, and M. Hermenegildo. An Abstract Interpretation-based Approach to Mobile Code Safety. In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, April 2004.
- [BCC⁺97] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series—TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and on-line version of the manual available at <http://clip.dia.fi.upm.es/Software/Ciao/> <http://clip.dia.fi.upm.es/Software/Ciao/>.
- [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.
- [BDD⁺97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging—AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.
- [BdlBH99] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.
- [BdlBH⁺01] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.
- [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.
- [Bou93] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.

- [CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
- [CC02] P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *POPL'02: 29ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, January 2002. ACM.
- [CGLH04] S.J. Craig, John P. Gallagher, M. Leuschel, and Kim S. Henriksen. Fully automatic binding time analysis for Prolog. In Sandro Etalle, editor, *Pre-Proceedings, 14th International Workshop on Logic-Based Program Synthesis and Transformation, LOPSTR 2004, Verona, August 2004*, pages 61–70, 2004.
- [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.
- [CLMV99] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93, 1999.
- [CLV95] M. Comini, G. Levi, and G. Vitiello. Declarative diagnosis revisited. In *1995 International Logic Programming Symposium*, pages 275–287, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.
- [Cou03] P. Cousot. Automatic Verification by Abstract Interpretation, Invited Tutorial. In *Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, number 2575 in LNCS, pages 20–24. Springer, January 2003.
- [CV94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.
- [dlBHB⁺96a] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, September 1996.

- [dlBHB⁺96b] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Trans. on Programming Languages and Systems*, 18(5):564–615, 1996.
- [dlBHM00] 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.
- [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.
- [DLGHL94] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.
- [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.
- [DZ92] P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
- [Gal03] J. P. Gallagher. A Program Transformation for Backwards Analysis of Logic Programs. In M. Bruynooghe, editor, *Proceedings of the International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR 2003)*, volume 3018 of LNCS, pages 92–105, 2003.
- [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.
- [GH91] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.
- [GH04] J. P. Gallagher and K. S. Henriksen. Abstract domains based on regular types. In V. Lifschitz and B. Demoen, editors, *Proceedings of the International Conference*

on *Logic Programming (ICLP'2004)*, volume 3132 of *Springer-Verlag Lecture Notes in Computer Science*, pages 27–42, 2004.

- [GP02a] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Fourth International Symposium on Practical Aspects of Declarative Languages (PADL'02)*, LNCS, pages 243–261, January 2002.
- [GP02b] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, number 2257 in LNCS, pages 243–261. Springer-Verlag, January 2002.
- [HALGP04] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Some Techniques for Automated, Resource-Aware Distributed and Mobile Computing in a Multi-Paradigm Programming System. In *Proc. of EURO-PAR 2004*, number 3149 in LNCS, pages 21–37. Springer-Verlag, August 2004.
- [HBC⁺99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.
- [HBPLG99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 Int'l. Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
- [Her00] M. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.
- [HPB99] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

- [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.
- [Leu98] M. Leuschel. Program Specialisation and Abstract Interpretation Reconciled. In *Joint International Conference and Symposium on Logic Programming*, June 1998.
- [LGBH04] P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, LNCS. Springer-Verlag, August 2004.
- [LGH95] P. López-García and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Press, Cambridge, MA.
- [LGHD96] 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*, 22:715–734, 1996.
- [LS88] Y. Lichtenstein and E. Y. Shapiro. Abstract algorithmic debugging. In R. A. Kowalski and K. A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 512–531, Seattle, Washington, August 1988. MIT.
- [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.
- [MdlBH94] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

- [MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, 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.
- [PBH00a] G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.
- [PBH00b] 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.
- [PBH00c] G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, 2000.
- [PdIBMS97] G. Puebla, M. García de la Banda, K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on Logic Programming*, pages 93–107, Cambridge, MA, June 1997. MIT Press.
- [PH96] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.
- [PH97] G. Puebla and M. Hermenegildo. Abstract Specialization and its Application to Program Parallelization. In J. Gallagher, editor, *Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.
- [PH99] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

- [PH00] G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
- [PHG99] G. Puebla, M. Hermenegildo, and J. Gallagher. An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework. In O Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, number NS-99-1 in BRISC Series, pages 75–85. University of Aarhus, Denmark, January 1999.
- [VB02] C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.
- [YS87] E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Concurrent Prolog: Collected Papers*, pages 211–244, 1987.

The Ciao Prolog Preprocessor

A Program Analysis, Verification, Debugging, and Source-to-Source Transformation Tool

The Ciao System Documentation Series

Technical Report CLIP 1/04 (first version 8/95).

Draft printed on: 16 March 2005

Version 1.0#1011 (2005/3/15, 13:0:46 CET)

Edited by: F. Bueno, P. López-García, G. Puebla, M. Hermenegildo

`clip@clip.dia.fi.upm.es`

`http://www.cliplab.org/`

Facultad de Informática

Universidad Politécnica de Madrid

Table of Contents

Summary	1
1 Introduction.....	3
1.1 How to use this manual	3
1.2 Note	3
1.3 Installation	3
1.4 Getting started	4
1.5 Usage and interface (<code>ciaopp</code>).....	5
1.6 Documentation on exports (<code>ciaopp</code>)	6
<code>current_pp_flag/2</code> (pred)	6
<code>set_pp_flag/2</code> (pred)	6
<code>push_pp_flag/2</code> (pred)	6
<code>pop_pp_flag/1</code> (pred).....	6
<code>pp_flag/1</code> (pred).....	6
<code>valid_flag_value/2</code> (prop)	7
<code>remove_action/1</code> (udreexp)	8
<code>add_action/1</code> (udreexp)	8
<code>transform/1</code> (pred)	8
<code>module/1</code> (pred)	8
<code>acheck/0</code> (pred)	8
<code>analyze/1</code> (pred)	9
<code>output/1</code> (pred).....	9
<code>output/0</code> (pred).....	9
1.7 Documentation on internals (<code>ciaopp</code>)	9
<code>analysis/1</code> (prop).....	9
<code>transformation/1</code> (prop)	12
<code>help/0</code> (pred)	14
1.8 Other information (<code>ciaopp</code>).....	14
1.8.1 Analysis with PLAI.....	14
1.8.2 Inter-modular Analysis.....	14
1.8.3 Abstract Partial Deduction	16
2 CiaoPP user menu interface.....	17
2.1 Usage and interface (<code>auto_interface</code>).....	18
2.2 Documentation on exports (<code>auto_interface</code>)	18
<code>auto_analyze/1</code> (pred)	18
<code>auto_optimize/1</code> (pred)	18
<code>auto_check_assert/1</code> (pred)	18
<code>customize/1</code> (pred)	18
<code>customize_and_exec/1</code> (pred)	19
<code>again/0</code> (pred)	19
<code>get_menu_configs/1</code> (pred).....	19
<code>save_menu_config/1</code> (pred)	19
<code>remove_menu_config/1</code> (pred)	19
<code>restore_menu_config/1</code> (pred)	19
<code>show_menu_configs/0</code> (pred)	19
<code>show_menu_config/1</code> (pred).....	20

3 Using Assertions for Preprocessing Programs.. 21

3.1	Assertions	21
3.1.1	Properties of Success States	21
3.1.2	Restricting Assertions to a Subset of Calls	22
3.1.3	Properties of Call States	22
3.1.4	Properties of the Computation	22
3.1.5	Compound Assertions	22
3.1.6	Examples	23
3.2	Properties	23
3.3	Preprocessing Units	24
3.4	Foreign Code	25
3.4.1	Examples	26
3.5	Dynamic Predicates	26
3.6	Entry Points	27
3.6.1	Examples	28
3.7	Modules	28
3.8	Dynamic Calls	29
3.8.1	Examples	29
3.9	Summary	30

4 The Ciao assertion package..... 31

4.1	More info	31
4.2	Some attention points	31
4.3	Usage and interface (<code>assertions</code>)	32
4.4	Documentation on new declarations (<code>assertions</code>)	32
	<code>pred/1</code> (decl)	32
	<code>pred/2</code> (decl)	33
	<code>calls/1</code> (decl)	33
	<code>calls/2</code> (decl)	33
	<code>success/1</code> (decl)	33
	<code>success/2</code> (decl)	34
	<code>comp/1</code> (decl)	34
	<code>comp/2</code> (decl)	34
	<code>prop/1</code> (decl)	34
	<code>prop/2</code> (decl)	35
	<code>entry/1</code> (decl)	35
	<code>modedef/1</code> (decl)	35
	<code>decl/1</code> (decl)	36
	<code>decl/2</code> (decl)	36
	<code>comment/2</code> (decl)	36
	<code>exit/1</code> (decl)	36
	<code>exit/2</code> (decl)	36
4.5	Documentation on exports (<code>assertions</code>)	37
	<code>check/1</code> (pred)	37
	<code>trust/1</code> (pred)	37
	<code>true/1</code> (pred)	37
	<code>false/1</code> (pred)	38

5	Types and properties related to assertions	39
5.1	Usage and interface (<code>assertions_props</code>)	39
5.2	Documentation on exports (<code>assertions_props</code>)	39
	<code>assrt_body/1</code> (regtype)	39
	<code>head_pattern/1</code> (prop)	40
	<code>complex_arg_property/1</code> (regtype)	41
	<code>property_conjunction/1</code> (regtype)	41
	<code>property_starterm/1</code> (regtype)	41
	<code>complex_goal_property/1</code> (regtype)	42
	<code>nabody/1</code> (prop)	42
	<code>dictionary/1</code> (regtype)	42
	<code>c_assrt_body/1</code> (regtype)	42
	<code>s_assrt_body/1</code> (regtype)	43
	<code>g_assrt_body/1</code> (regtype)	43
	<code>assrt_status/1</code> (regtype)	44
	<code>assrt_type/1</code> (regtype)	44
	<code>predfunctor/1</code> (regtype)	44
	<code>propfunctor/1</code> (regtype)	44
	<code>docstring/1</code> (prop)	44
6	Declaring regular types	47
6.1	Defining properties	47
6.2	Usage and interface (<code>regtypes</code>)	50
6.3	Documentation on new declarations (<code>regtypes</code>)	50
	<code>regtype/1</code> (decl)	50
	<code>regtype/2</code> (decl)	51
7	Basic data types and properties	53
7.1	Usage and interface (<code>basic_props</code>)	53
7.2	Documentation on exports (<code>basic_props</code>)	53
	<code>term/1</code> (regtype)	53
	<code>int/1</code> (regtype)	54
	<code>nnegint/1</code> (regtype)	54
	<code>flt/1</code> (regtype)	54
	<code>num/1</code> (regtype)	55
	<code>atm/1</code> (regtype)	55
	<code>struct/1</code> (regtype)	56
	<code>gnd/1</code> (regtype)	56
	<code>constant/1</code> (regtype)	57
	<code>callable/1</code> (regtype)	57
	<code>operator_specifier/1</code> (regtype)	57
	<code>list/1</code> (regtype)	58
	<code>list/2</code> (regtype)	59
	<code>member/2</code> (prop)	59
	<code>sequence/2</code> (regtype)	60
	<code>sequence_or_list/2</code> (regtype)	60
	<code>character_code/1</code> (regtype)	61
	<code>string/1</code> (regtype)	61
	<code>predname/1</code> (regtype)	61
	<code>atm_or_atm_list/1</code> (regtype)	62
	<code>compat/2</code> (prop)	62
	<code>inst/2</code> (prop)	63
	<code>iso/1</code> (prop)	63
	<code>not_further_inst/2</code> (prop)	63
	<code>sideff/2</code> (prop)	63

	regtype/1 (prop)	64
	native/1 (prop)	64
	native/2 (prop)	64
	eval/1 (prop)	64
	equiv/2 (prop)	65
8	Properties which are native to analyzers	67
8.1	Usage and interface (<code>native_props</code>)	67
8.2	Documentation on exports (<code>native_props</code>)	67
	covered/2 (prop)	67
	linear/1 (prop)	67
	mshare/1 (prop)	68
	nonground/1 (prop)	68
	fails/1 (prop)	68
	not_fails/1 (prop)	68
	possibly_fails/1 (prop)	69
	covered/1 (prop)	69
	not_covered/1 (prop)	69
	is_det/1 (prop)	69
	non_det/1 (prop)	69
	possibly_nondet/1 (prop)	69
	mut_exclusive/1 (prop)	69
	not_mut_exclusive/1 (prop)	70
	size_lb/2 (prop)	70
	size_ub/2 (prop)	70
	size/2 (prop)	70
	size_o/2 (prop)	70
	steps_lb/2 (prop)	70
	steps_ub/2 (prop)	71
	steps/2 (prop)	71
	steps_o/2 (prop)	71
	finite_solutions/1 (prop)	71
	terminates/1 (prop)	71
	indep/1 (prop)	71
	indep/2 (prop)	72
	instance/2 (prop)	72
9	Run-time checking of assertions	73
9.1	Usage and interface (<code>rtchecks</code>)	73
9.2	Documentation on exports (<code>rtchecks</code>)	73
	check/1 (pred)	73
	References	75
	Predicate/Method Definition Index	79
	Regular Type Definition Index	81
	Concept Definition Index	83
	Global Index	85

Summary

CiaoPP is the precompiler of the Ciao Prolog 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, 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, 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 information generated by analysis, the assertions in the system libraries, and the assertions optionally included in user programs as 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.

This documentation corresponds to version 1.0#1011 (2005/3/15, 13:0:46 CET).

1 Introduction

CiaoPP is the precompiler of the Ciao Prolog 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, 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, 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 information generated by analysis, the assertions in the system libraries, and the assertions optionally included in user programs as 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, and lists all commands, flags, and options necessary to use its program transformation, debugging, and analysis functionality. 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 [\[A tutorial overview of CiaoPP\]](#), page [in the form of a tutorial on CiaoPP](#).

1.2 Note

We are in the process of merging all CiaoPP 0.8 functionality into the 1.0 version. In the meantime, 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

The distribution of CiaoPP consists of its source files, written in Ciao. Thus, you need to have Ciao (version 1.11 or higher) installed.

Once you have the source files in an installation directory, please, edit first file `CIAOPPSSETTINGS.pl` and change the required options at the top of that file. Then, run:

```
lpmake install
```

(lpmake is an utility that comes with the Ciao distribution). This will:

1. Create an executable `ciaopp` that you can run under Unix.
2. Create an executable `ciaopp.bat` that you can run under Windows.
3. Set up things so that you can use CiaoPP as a library module from, e.g., the Ciao shell. In order to do this, please follow the (short) instructions that `lpmake install` prints out at the end when you run it.

1.4 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. Many of these make use of several flags to modify their behaviour. The available flags are described later in this chapter. The options available at the menu are described to some extent also later in this chapter. What follows is an introductory overview of the menu. Commands to manipulate the menu are described in Chapter 2 [CiaoPP user menu interface], page 17.

The execution of command `customize_and_exec(FileName)`, which takes a file name as argument, at the CiaoPP shell prompt displays the menu, which will look (depending on the options available in the current CiaoPP version) something like:

```
?- customize_and_exec( myfile ).
```

```
(Press h for help)
```

```
Use Saved Menu Configuration: [stored_cfg1] (none) ?
Select Menu Level:          [naive, expert] (naive) ?
Select Action Group:        [analyze, check_assertions, optimize]
                             (analyze) ?
Select Cost Analysis:        [none, steps_ub, steps_lb, steps_ualb,
                             steps_o] (none) ?
Select Mode Analysis:        [none, pd, pdb, def, gr, share, shareson,
                             shfr, shfrson, shfrnv, son, aeq, depth,
                             path, diffllsign, fr, frdef, lsign]
                             (shfr) ?
Select Type Analysis:        [none, eterms, ptypes, svterms, terms]
                             (eterms) ?
Select Type Output:          [defined, all] (all) ?
Perform Non-Failure Analysis: [none, nf, nfg] (none) ?
Perform Determinism Analysis: [none, det] (none) ?
Print Program Point Info:    [off, on] (off) ?
Collapse AI Info:           [off, on] (on) ?
Note: Current Saved Menu Configurations: [stored_cfg1]
Menu Configuration Name:      (none) ?
```

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 program optimization (`optimize`), and you can later combine the three kinds of preprocessing. The relevant options for the action group selected are then shown, together with the relevant flags. See `analysis/1` and `transformation/1` later in this chapter for a description of the values for each option. See `pp_flag/1` later in this chapter for a description of the values of each flag.

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 6 [Declaring regular types], page 47. Non-failure and Determinacy Analysis 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 it 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 3 [Using Assertions for Preprocessing Programs], page 21, 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).

1.5 Usage and interface (ciaopp)

- **Library usage:**

The `ciaopp` executable starts a shell at which prompt you can issue any of the commands described below and in the next chapter as exports.

- **Other modules used:**

- *Application modules:*

```
ciaopp(driver),    ciaopp(preprocess_flags),    ciaopp(printer),    auto_
interface(auto_interface), auto_interface(auto_help), typeslib(typeslib),
program(p_asr), infer(infer).
```

- *System library modules:*

```
messages, system.
```

- *Internal (engine) modules:*

```
term_basic, arithmetic, atomic_basic, attributes, mattr_global, basic_props,
basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags,
streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_
support.
```

1.6 Documentation on exports (ciaopp)

- current_pp_flag/2:** PREDICATE
 Usage: `current_pp_flag(Name, Value)`
- *Description:* Preprocess flag `Name` has the value `Value`.
 - *The following properties should hold at call time:*
 - `Value` is a free variable. (var/1)
 - `Name` is a valid preprocessor flag. (pp_flag/1)
 - *The following properties should hold upon exit:*
 - `Value` is a valid value for `Name` preprocessor flag. (valid_flag_value/2)
- set_pp_flag/2:** PREDICATE
 Usage: `set_pp_flag(Name, Value)`
- *Description:* Sets the `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 `Name` preprocessor flag. (valid_flag_value/2)
- push_pp_flag/2:** PREDICATE
 Usage: `push_pp_flag(Flag, Value)`
- *Description:* Sets the `Value` for `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 `Flag` preprocessor flag. (valid_flag_value/2)
- pop_pp_flag/1:** PREDICATE
 Usage: `pop_pp_flag(Flag)`
- *Description:* Restores the value of `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.
 - `initial_guess` (`botfirst,botbest,botall,bottom`) The policy for obtaining initial guess when computing the analysis of a predicate from the current module.
 - `entry_policy` (`all,top_level,force`) The policy for obtaining entry call patterns for exported predicates during inter-modular analysis.
 - `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,local_builtin,local_emb,jump_builtin`) 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,safe,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.

- valid_flag_value/2:** PROPERTY
Usage: `valid_flag_value(Name, Value)`
 – *Description:* `Value` is a valid value for `Name` preprocessor flag.
 – *If the following properties should hold at call time:*
 `Name` is a valid preprocessor flag. (pp_flag/1)
 `flag_value(Value)` (undefined property)
- remove_action/1:** (UNDOC_REEXPORT)
 Imported from `driver` (see the corresponding documentation for details).
- add_action/1:** (UNDOC_REEXPORT)
 Imported from `driver` (see the corresponding documentation for details).
- transform/1:** PREDICATE
Usage 1: `transform(Trans)`
 – *Description:* Returns on backtracking all available program transformations.
 – *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)`
 – *Description:* 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(File)`
 – *Description:* Reads `File` and sets it as the current module.
 – *The following properties should hold at call time:*
 `File` is currently a term which is not a free variable. (nonvar/1)
- Usage 2:** `module(FileList)`
 – *Description:* Reads the list of files `FileList` and sets the set of them as the current module.
 – *The following properties should hold at call time:*
 `FileList` is currently a term which is not a free variable. (nonvar/1)
 `FileList` is a list. (list/1)

acheck/0:	PREDICATE
Usage:	
– <i>Description:</i> Checks assertions w.r.t. analysis information.	
analyze/1:	PREDICATE
Usage 1: analyze(Analysis)	
– <i>Description:</i> Returns on backtracking all available analyses.	
– <i>The following properties should hold at call time:</i>	
Analysis is a free variable.	(var/1)
– <i>The following properties should hold upon exit:</i>	
Analysis is a valid analysis identifier.	(analysis/1)
Usage 2: analyze(Analysis)	
– <i>Description:</i> Analyzes the current module with Analysis.	
– <i>The following properties should hold at call time:</i>	
Analysis is currently a term which is not a free variable.	(nonvar/1)
Analysis is a valid analysis identifier.	(analysis/1)
output/1:	PREDICATE
Usage: output(Output)	
– <i>Description:</i> Outputs the current module preprocessing state to a file Output.	
– <i>The following properties should hold at call time:</i>	
Output is currently a term which is not a free variable.	(nonvar/1)
output/0:	PREDICATE
Usage:	
– <i>Description:</i> Outputs the current Module preprocessing state to a file Module_opt.pl.	

1.7 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 and to have in the output only predefined types, 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 [VHCLC95]).
 - **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)**). Whether unfolding should take place (thus, possibly creating new patterns) or not is governed by flag **global_control**:

 - **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 [BibRef: homeoemb].

Only **hom_emb** guarantees termination. However, **id** and **inst** are more efficient, and terminating in many practical cases.

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).
 - **diffsign** 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)`

- *Description:* `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 explore 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;
 - `local_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.

Unfolding is performed continuously on the already unfolded clauses, until a condition for stopping the process is satisfied. This condition is established 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.
- `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,dLBHM00] 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].
- `cr1p` 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)`

- *Description:* `Transformation` is a valid transformation identifier.

help/0:

PREDICATE

No further documentation available for this predicate.

1.8 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).

1.8.1 Analysis with PLAI

Most of the analyses of CiaoPP are performed with the PLAI (Programming in Logic with Abstract Interpretation) framework [BdlBH94]. 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` .

1.8.2 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`. Also, flag `fixpoint` should be set to `di`. 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 `auto_analyze/2` with the main module of the program, until inter-modular fixed point.
- `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.

1.8.3 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.

2 CiaoPP user menu interface

Author(s): David Trallero Mena.

Version: 1.0#1011 (2005/3/15, 13:0:46 CET)

Version of last change: 1.0#1008 (2005/3/13, 23:23:7 CET)

This module defines a simplified user-level interface for CiaoPP. It complements the more expert oriented interface defined by modules `driver` and `printer`. 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.

The basic way of using it is as follows:

- In general, the default setting should be adequate for most basic tasks. Thus:
 - To **check a program** simply call `auto_check_assertions/1` with the file name as argument. In `emacs` this can be done most easily by clicking on the corresponding button in the toolbar or in the CiaoPP menus.
 - To **optimize (transform) a program** simply call `auto_optimize/1` with the file name as argument. In `emacs` this can be done most easily by clicking on the corresponding button in the toolbar or in the CiaoPP menus.
 - To **analyze a program** simply call `auto_analyze/1` with the file name as argument. In `emacs` this can be done most easily by clicking on the corresponding button in the toolbar or in the CiaoPP menus.
 - To **customize the actions performed by the above operations** call `auto_optimize/1` with the file name as argument. This will prompt (with help) for the value of the different options and flags.
- Alternatively, one can change what the above commands do by **customizing** each of them. To this end, call `customize_and_exec/1` with the file name as argument. In `emacs` this can be done 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, customize/1,`
`customize_and_exec/1, again/0.`
- **Other modules used:**
 - *Application modules:*
`ciaopp(preprocess_flags), ciaopp(driver), ciaopp(printer), ciaopp(menu_`
`generator).`
 - *System library modules:*
`lists, aggregates, messages.`
 - *Internal (engine) modules:*
`term_basic, arithmetic, atomic_basic, attributes, mattr_global, basic_props,`
`basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags,`
`streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_`
`support.`

2.2 Documentation on exports (auto_interface)

auto_analyze/1: PREDICATE

Usage: `auto_analyze(F)`

- *Description:* Analyze the module `F` with the default analysis options (use `customize(analyze)` to change these options).

auto_optimize/1: PREDICATE

Usage: `auto_optimize(F)`

- *Description:* Optimize file `F` with default options (use `customize(optimize)` to change these options).

auto_check_assert/1: PREDICATE

Usage: `auto_check_assert(F)`

- *Description:* Check the assertions in file `F` giving errors if assertions are violated (use `customize(check_assertions)` to change these options).

customize/1: PREDICATE

Usage: `customize(X)`

- *Description:* `customize` is used for change 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_exec/1:** PREDICATE
Usage: `customize_and_exec(File)`
 – *Description:* It is like doing `customize(all)`, `auto_analyze(File)`. Consider `auto_optimize/1` or `auto_check_assertions/1` in proper cases.
- again/0:** PREDICATE
Usage:
 – *Description:* Performs the last actions done by `customize_and_exec/1`, on the last file previously analyzed, checked, or optimized
- get_menu_configs/1:** PREDICATE
Usage: `get_menu_configs(X)`
 – *Description:* 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 atoms. (list/2)
- save_menu_config/1:** PREDICATE
Usage: `save_menu_config(Name)`
 – *Description:* 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)`
 – *Description:* 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)`
 – *Description:* 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:
 – *Description:* Show all stored configurations.

show_menu_config/1:

PREDICATE

Usage: show_menu_config(C)

- *Description:* 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)

3 Using Assertions for Preprocessing Programs

Author(s): F. Bueno.

Version: 1.0#1011 (2005/3/15, 13:0:46 CET)

Version of last change: 0.7#33 (2000/3/28, 10:54:38 CEST)

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 4 [The Ciao assertion package], page 31, a complete reference description of assertions is provided. More detailed explanations of the language can be found in [PBH97].

This chapter also introduces and uses properties, and among them (regular) types. See Chapter 7 [Basic data types and properties], page 53, for a concrete reference of (some of) the Ciao properties. See Chapter 6 [Declaring regular types], page 47, 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.

3.1 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 qualified—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.

3.1.1 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, in (C)LP calls to a predicate may either succeed or fail. The postcondition stated in a `success` assertion only refers to successful executions.

3.1.2 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”.

3.1.3 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`”.

3.1.4 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.

3.1.5 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.

3.1.6 Examples

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.

3.2 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 CLP 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.

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 6 [Declaring regular types], page 47) are documented in Chapter 7 [Basic data types and properties], page 53.

3.3 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 consist 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

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

queries, calls from other parts of the program to the unit code, calls to the unit code from unit code which does not appear explicitly 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 explicitly 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

3.4 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

² When the language supports a module system, entry points are implicitly 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].

are called by it, builtin predicates⁴ 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

- 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 3.1 [Assertions], page 21.

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.

3.4.1 Examples

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.

3.5 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.

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.

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.

3.6 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 explicitly 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 3.1 [Assertions], page 21.

3.6.1 Examples

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 CLP 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).

3.7 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.

3.8 Dynamic Calls

In addition to entry points there are other calls that may occur from within a piece of code which do not explicitly 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.

3.8.1 Examples

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:

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

```
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).
```

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.

3.9 Summary

To process CLP 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.

4 The Ciao assertion package

Author(s): Manuel Hermenegildo, Francisco Bueno, German Puebla.

Version: 1.11#309 (2005/3/16, 16:41:12 CET)

Version of last change: 1.5#8 (1999/12/9, 21:1:11 MET)

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 be formal specifications (such as preconditions and post-conditions) or 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.

4.1 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 the document “An Assertion Language for Debugging of Constraint Logic Programs (Technical Report CLIP2/97.1)”. 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.

4.2 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.

4.3 Usage and interface (assertions)

- **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]).
:- include(library(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], 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`, `calls/1`, `calls/2`, `success/1`, `success/2`, `comp/1`, `comp/2`, `prop/1`,
`prop/2`, `entry/1`, `modedef/1`, `decl/1`, `decl/2`, `comment/2`, `exit/1`, `exit/2`.

- **Other modules used:**

- *System library modules:*
`assertions/assertions_props`.
- *Internal (engine) modules:*
`term_basic`, `arithmetic`, `atomic_basic`, `attributes`, `matrr_global`, `basic_props`,
`basiccontrol`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`,
`streams_basic`, `system_info`, `term_compare`, `term_typing`, `hiord_rt`, `debugger_`
`support`.

4.4 Documentation on new declarations (assertions)

`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 explicitly qualified. Non-qualified `pred/1` assertions are assumed the qualifier `check`.

Usage: :- pred(AssertionStatus, AssertionBody).

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

AssertionStatus is an acceptable status for an assertion. (assrt_status/1)

AssertionBody is an assertion body. (assrt_body/1)

calls/1: DECLARATION

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 explicitly qualified. Non-qualified `calls/1` assertions are assumed the qualifier `check`.

Usage: :- calls(AssertionStatus, AssertionBody).

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

AssertionStatus is an acceptable status for an assertion. (assrt_status/1)

AssertionBody is a call assertion body. (c_assrt_body/1)

success/1: DECLARATION

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

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

Usage: `:- success(AssertionStatus, AssertionBody).`

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

AssertionStatus is an acceptable status for an assertion. (assrt_status/1)

AssertionBody is a predicate assertion body. (s_assrt_body/1)

comp/1: DECLARATION

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 explicitly qualified. Non-qualified `comp/1` assertions are assumed the qualifier `check`.

Usage: `:- comp(AssertionStatus, AssertionBody).`

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

AssertionStatus is an acceptable status for an assertion. (assrt_status/1)

AssertionBody is a comp assertion body. (g_assrt_body/1)

prop/1: DECLARATION

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 6 [Declaring regular types], page 47 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 explicitly qualified. Non-qualified `prop/1` assertions are assumed the qualifier `check`.

Usage: `:- prop(AssertionStatus, AssertionBody).`

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

`AssertionStatus` is an acceptable status for an assertion. (assrt_status/1)

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

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)

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 explicitly qualified. Non-qualified `decl/1` assertions are assumed the qualifier `check`.

Usage: `:- decl(AssertionStatus, AssertionBody).`

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

AssertionStatus is an acceptable status for an assertion. (assrt_status/1)

AssertionBody is an assertion body. (assrt_body/1)

comment/2: DECLARATION

Usage: `:- comment(Pred, Comment).`

– *Description:* This assertion gives 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)

exit/1: DECLARATION

No further documentation available for this predicate.

exit/2: DECLARATION

No further documentation available for this predicate.

4.5 Documentation on exports (assertions)

check/1: PREDICATE

Usage: `check(PropertyConjunction)`

- *Description:* 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 9 [Run-time checking of assertions], page 73.
- *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)`

- *Description:* 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)`

- *Description:* 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)

- *Description:* 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)

5 Types and properties related to assertions

Author(s): Manuel Hermenegildo.

Version: 1.11#309 (2005/3/16, 16:41:12 CET)

Version of last change: 1.7#156 (2001/11/24, 13:23:30 CET)

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.

5.1 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.`
- **Other modules used:**
 - *System library modules:*
`dcg_expansion.`
 - *Internal (engine) modules:*
`term_basic, arithmetic, atomic_basic, attributes, mattr_global, basic_props, basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_support.`

5.2 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:

$$\text{Pr} [:: \text{DP}] [:\text{ CP}] [=> \text{AP}] [+ \text{GP}] [\# \text{CO}]$$

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 to 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)`

- *Description:* 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

$p(\text{Input}, \text{Parameter}, \text{Output})$ and stating that the property `list(Parameter, int)` holds for the calls of the predicate.

Usage: `head_pattern(Pr)`

- *Description:* `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_starterterm/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)`

- *Description:* `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)`

- *Description:* `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_starterterm/1:

REGTYPE

This type defines a second, compact format in which properties can be expressed in the bodies of assertions. A `property_starterterm/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_starterterm/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_starterterm(Props)`

- *Description:* **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 implicitly 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)**

- *Description:* **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)**

- *Description:* **ABody** is a normalized assertion body.

dictionary/1: REGTYPE

Usage: **dictionary(D)**

- *Description:* **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)`

- *Description:* `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)`

- *Description:* `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)`

- *Description:* `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)`

- *Description:* `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(modedef).
```

Usage: `assrt_type(X)`

- *Description:* `X` is an admissible kind of assertion.

predfunctor/1: REGTYPE

Usage: `predfunctor(X)`

- *Description:* `X` is a type of assertion which defines a predicate.

propfunctor/1: REGTYPE

Usage: `propfunctor(X)`

- *Description:* `X` is a type of assertion which defines a *property*.

docstring/1:

PROPERTY

Usage: `docstring(String)`

- *Description:* `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 `lpadoc` manual for documentation on comments.

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 9 [Run-time checking of assertions], page 73) 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.

6.2 Usage and interface (regtypes)

- **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.`
- **Other modules used:**
 - *System library modules:*
`assertions/assertions_props.`
 - *Internal (engine) modules:*
`term_basic.`

6.3 Documentation on new declarations (regtypes)

regtype/1:

DECLARATION

This assertion is similar to a pred assertion but it flags that the predicate being documented is also a “regular type.” This allows for example checking whether it is in the class of types supported by the type checking and inference modules. Currently, types are properties whose definitions are *regular programs*.

A regular program is defined by a set of clauses, each of the form:

$$p(x, v_1, \dots, v_n) \text{ :- } body_1, \dots, body_k.$$

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. $n \geq 0$ and p/n is a *parametric type functor* (whereas the predicate defined by the clauses is $p/n+1$).
4. v_1, \dots, 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. $q(y, t_1, \dots, t_m)$ where $m \geq 0$, q/m is a *parametric type functor*, not in the set of functors `=/2, ^/2, ./3`.
 t_1, \dots, 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 7 [Basic data types and properties], page 53.

The set of 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 explicitly qualified. Non-qualified `regtype/1` assertions are assumed the qualifier `check`. Note that checking regular type definitions should be done with the `ciaopp` preprocessor.

Usage: `:- regtype(AssertionStatus, AssertionBody).`

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

AssertionStatus is an acceptable status for an assertion. (assrt_status/1)

AssertionBody is an assertion body. (assrt_body/1)

7 Basic data types and properties

Author(s): Daniel Cabeza, Manuel Hermenegildo.

Version: 1.11#309 (2005/3/16, 16:41:12 CET)

Version of last change: 1.11#307 (2005/3/10, 13:35:50 CET)

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.

7.1 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, not_further_inst/2, sideff/2, regtype/1, native/1, native/2, eval/1, equiv/2.
 - *Regular Types:*
term/1, int/1, nnegint/1, flt/1, num/1, atm/1, struct/1, gnd/1, constant/1, callable/1, operator_specifier/1, list/1, list/2, sequence/2, sequence_or_list/2, character_code/1, string/1, predname/1, atm_or_atm_list/1.
- **Other modules used:**
 - *System library modules:*
terms_check.
 - *Internal (engine) modules:*
term_basic, arithmetic, atomic_basic, attributes, mattr_global, basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_support.

7.2 Documentation on exports (basic_props)

- term/1:** REGTYPE
The most general type (includes all possible terms).
- General properties:** term(X)
- *The following properties hold globally:*
term(X) is side-effect free. (sideff/2)
 - term(X)
– *The following properties hold globally:*
term(X) is evaluable at compile-time. (eval/1)
 - term(X)
– *The following properties hold globally:*
term(X) is equivalent to true. (equiv/2)
- Usage:** term(X)

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

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.

General properties: `int(T)`

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

`int(T)`

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

Usage: `int(T)`

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

nnegint/1: REGTYPE

The type of non-negative integers, i.e., natural numbers.

General properties: `nnegint(T)`

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

`nnegint(T)`

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

`nnegint(T)`

- *The following properties hold upon exit:*
T is currently ground (it contains no variables). (ground/1)

Usage: `nnegint(T)`

- *Description:* T is a non-negative integer.
- *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (native/1)

- flt/1:** REGTYPE
- The type of floating-point numbers. The range of floats is the one provided by the C `double` type, typically $[4.9\text{e-}324, 1.8\text{e+}308]$ (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`.
- General properties: flt(T)**
- *The following properties hold globally:*
`flt(T)` is side-effect free. (sideff/2)
- `flt(T)`
- *If the following properties hold at call time:*
`T` is currently a term which is not a free variable. (nonvar/1)
then the following properties hold globally:
`flt(T)` is evaluable at compile-time. (eval/1)
- `flt(T)`
- *The following properties hold upon exit:*
`T` is currently ground (it contains no variables). (ground/1)
- Usage: flt(T)**
- *Description:* `T` is a float.
 - *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (native/1)
-
- num/1:** REGTYPE
- The type of numbers, that is, integer or floating-point.
- General properties: num(T)**
- *The following properties hold globally:*
`num(T)` is side-effect free. (sideff/2)
- `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)
- Usage: num(T)**
- *Description:* `T` is a number.
 - *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (native/1)
-
- atm/1:** REGTYPE
- The type of atoms, or non-numeric constants. The size of atoms is unbound.
- General properties: atm(T)**
- *The following properties hold globally:*
`atm(T)` is side-effect free. (sideff/2)

atm(T)

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

T is currently a term which is not a free variable.

(nonvar/1)

then the following properties hold globally:

atm(T) is evaluable at compile-time.

(eval/1)

atm(T)

- *The following properties hold upon exit:*

T is currently ground (it contains no variables).

(ground/1)

Usage: atm(T)

- *Description:* T is an atom.

- *The following properties hold globally:*

This predicate is understood natively by CiaoPP.

(native/1)

struct/1:

REGTYPE

The type of compound terms, or terms with non-zeroary functors. By now there is a limit of 255 arguments.

General properties: struct(T)

- *The following properties hold globally:*

struct(T) is side-effect free.

(sideff/2)

struct(T)

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

T is currently a term which is not a free variable.

(nonvar/1)

then the following properties hold globally:

struct(T) is evaluable at compile-time.

(eval/1)

struct(T)

- *The following properties hold upon exit:*

T is currently a term which is not a free variable.

(nonvar/1)

Usage: struct(T)

- *Description:* T is a compound term.

- *The following properties hold globally:*

This predicate is understood natively by CiaoPP.

(native/1)

gnd/1:

REGTYPE

The type of all terms without variables.

General properties: gnd(T)

- *The following properties hold globally:*

gnd(T) is side-effect free.

(sideff/2)

gnd(T)

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

T is currently ground (it contains no variables).

(ground/1)

then the following properties hold globally:

gnd(T) is evaluable at compile-time.

(eval/1)

`gnd(T)`

- *The following properties hold upon exit:*
T is currently ground (it contains no variables). (ground/1)

Usage: `gnd(T)`

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

constant/1:

REGTYPE

General properties: `constant(T)`

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

`constant(T)`

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

`constant(T)`

- *The following properties hold upon exit:*
T is currently ground (it contains no variables). (ground/1)

Usage: `constant(T)`

- *Description:* T is an atomic term (an atom or a number).

callable/1:

REGTYPE

General properties: `callable(T)`

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

`callable(T)`

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

`callable(T)`

- *The following properties hold upon exit:*
T is currently a term which is not a free variable. (nonvar/1)

Usage: `callable(T)`

- *Description:* T is a term which represents a goal, i.e., an atom or a structure.

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.

General properties: operator_specifier(X)

- *The following properties hold globally:*

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

operator_specifier(X)

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

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

then the following properties hold globally:

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

operator_specifier(T)

- *The following properties hold upon exit:*

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

Usage: operator_specifier(X)

- *Description:* X specifies the type and associativity of an operator.

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).
```

General properties: list(L)

- *The following properties hold globally:*

list(L) is side-effect free. (sideff/2)

list(L)

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

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

then the following properties hold globally:

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

list(T)

- *The following properties hold upon exit:*

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

Usage: list(L)

- *Description:* L is a list.

list/2:

REGTYPE

list(L, T)

L is a list, and for all its elements, T holds.

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

General properties: list(L, T)

- *The following properties hold globally:*

list(L,T) is side-effect free. (sideeff/2)

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)

list(X, T)

- *The following properties hold upon exit:*

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

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

Usage: list(L, T)

- *Description:* L is a list of Ts.

member/2:

PROPERTY

General properties: member(X, L)

- *The following properties hold globally:*

member(X,L) is side-effect free. (sideeff/2)

member(X, L)

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

L is a list. (list/1)

then the following properties hold globally:

member(X,L) is evaluable at compile-time. (eval/1)

member(_X, L)

– *The following properties hold upon exit:*

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

Usage: `member(X, L)`

– *Description:* X is an element of L.

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.

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

General properties: `sequence(S, T)`

– *The following properties hold globally:*

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

`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)

`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)

Usage: `sequence(S, T)`

– *Description:* S is a sequence of Ts.

sequence_or_list/2: REGTYPE

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

General properties: `sequence_or_list(S, T)`

– *The following properties hold globally:*

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

`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)

`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)

Usage: `sequence_or_list(S, T)`

– *Description:* S is a sequence or list of Ts.

character_code/1:	REGTYPE
General properties: <code>character_code(T)</code>	
– <i>The following properties hold upon exit:</i>	
T is an integer.	(int/1)
<code>character_code(T)</code>	
– <i>The following properties hold globally:</i>	
<code>character_code(T)</code> is side-effect free.	(sideff/2)
<code>character_code(T)</code>	
– <i>If the following properties hold at call time:</i>	
T is currently a term which is not a free variable.	(nonvar/1)
<i>then the following properties hold globally:</i>	
<code>character_code(T)</code> is evaluable at compile-time.	(eval/1)
<code>character_code(I)</code>	
– <i>The following properties hold upon exit:</i>	
I is currently ground (it contains no variables).	(ground/1)
Usage: <code>character_code(T)</code>	
– <i>Description:</i> T is an integer which is a character code.	
string/1:	REGTYPE
A string is a list of character codes. The usual syntax for strings "string" is allowed, which is equivalent to <code>[0's,0't,0'r,0'i,0'n,0'g]</code> or <code>[115,116,114,105,110,103]</code> . There is also a special Ciao syntax when the list is not complete: <code>"st" R</code> is equivalent to <code>[0's,0't R]</code> .	
General properties: <code>string(T)</code>	
– <i>The following properties hold upon exit:</i>	
T is a list of <code>character_codes</code> .	(list/2)
<code>string(T)</code>	
– <i>The following properties hold globally:</i>	
<code>string(T)</code> is side-effect free.	(sideff/2)
<code>string(T)</code>	
– <i>If the following properties hold at call time:</i>	
T is currently ground (it contains no variables).	(ground/1)
<i>then the following properties hold globally:</i>	
<code>string(T)</code> is evaluable at compile-time.	(eval/1)
<code>string(T)</code>	
– <i>The following properties hold upon exit:</i>	
T is currently a term which is not a free variable.	(nonvar/1)
Usage: <code>string(T)</code>	
– <i>Description:</i> T is a string (a list of character codes).	

- predname/1:** REGTYPE
- General properties:** `predname(P)`
- *The following properties hold globally:*
`predname(P)` is side-effect free. (sideeff/2)
- `predname(P)`
- *If the following properties hold at call time:*
`P` is currently ground (it contains no variables). (ground/1)
then the following properties hold globally:
`predname(P)` is evaluable at compile-time. (eval/1)
- `predname(P)`
- *The following properties hold upon exit:*
`P` is currently ground (it contains no variables). (ground/1)
- Usage:** `predname(P)`
- *Description:* `P` is a Name/Arity structure denoting a predicate name:

```

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

```
-
- atm_or_atm_list/1:** REGTYPE
- General properties:** `atm_or_atm_list(T)`
- *The following properties hold globally:*
`atm_or_atm_list(T)` is side-effect free. (sideeff/2)
- `atm_or_atm_list(T)`
- *If the following properties hold at call time:*
`T` is currently ground (it contains no variables). (ground/1)
then the following properties hold globally:
`atm_or_atm_list(T)` is evaluable at compile-time. (eval/1)
- `atm_or_atm_list(T)`
- *The following properties hold upon exit:*
`T` is currently ground (it contains no variables). (ground/1)
- Usage:** `atm_or_atm_list(T)`
- *Description:* `T` is an atom or a list of atoms.
-
- 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.
- Meta-predicate* with arguments: `compat(?,pred(1))`.
- General properties:** `compat(Term, Prop)`

- *If the following properties hold at call time:*
 - Term is currently ground (it contains no variables). (ground/1)
 - Prop is currently ground (it contains no variables). (ground/1)
 - then the following properties hold globally:*
 - compat(Term,Prop) is evaluable at compile-time. (eval/1)
- Usage:** compat(Term, Prop)
- *Description:* Term is compatible with Prop
-
- inst/2:** PROPERTY
- Meta-predicate with arguments: inst(?,pred(1)).*
- General properties:** inst(Term, Prop)
- *The following properties hold globally:*
 - inst(Term,Prop) is side-effect free. (sideff/2)
- inst(Term, Prop)
- *If the following properties hold at call time:*
 - Term is currently ground (it contains no variables). (ground/1)
 - Prop is currently ground (it contains no variables). (ground/1)
 - then the following properties hold globally:*
 - inst(Term,Prop) is evaluable at compile-time. (eval/1)
- Usage:** inst(Term, Prop)
- *Description:* Term is instantiated enough to satisfy Prop.
-
- iso/1:** PROPERTY
- General properties:** iso(G)
- *The following properties hold globally:*
 - iso(G) is side-effect free. (sideff/2)
- Usage:** iso(G)
- *Description:* Complies with the ISO-Prolog standard.
-
- not_further_inst/2:** PROPERTY
- General properties:** not_further_inst(G, V)
- *The following properties hold globally:*
 - not_further_inst(G,V) is side-effect free. (sideff/2)
- Usage:** not_further_inst(G, V)
- *Description:* V is not further instantiated.

sideff/2:	PROPERTY
<code>sideff(G, X)</code>	
Declares that <code>G</code> is side-effect free (if its execution has no observable result other than its success, its failure, or its abortion), <code>soft</code> (if its execution may have other observable results which, however, do not affect subsequent execution, e.g., input/output), or <code>hard</code> (e.g., <code>assert/retract</code>).	
<i>Meta-predicate</i> with arguments: <code>sideff(goal,?)</code> .	
General properties: <code>sideff(G, X)</code>	
– <i>The following properties hold globally:</i>	
This predicate is understood natively by CiaoPP.	(native/1)
<code>sideff(G,X)</code> is side-effect free.	(sideff/2)
Usage: <code>sideff(G, X)</code>	
– <i>Description:</i> <code>G</code> is side-effect <code>X</code> .	
– <i>If the following properties should hold at call time:</i>	
<code>X</code> is an element of [<code>free,soft,hard</code>].	(member/2)
regtype/1:	PROPERTY
<i>Meta-predicate</i> with arguments: <code>regtype(goal)</code> .	
General properties: <code>regtype(G)</code>	
– <i>The following properties hold globally:</i>	
<code>regtype(G)</code> is side-effect free.	(sideff/2)
Usage: <code>regtype(G)</code>	
– <i>Description:</i> Defines a regular type.	
native/1:	PROPERTY
<i>Meta-predicate</i> with arguments: <code>native(goal)</code> .	
General properties: <code>native(P)</code>	
– <i>The following properties hold globally:</i>	
<code>native(P)</code> is side-effect free.	(sideff/2)
Usage: <code>native(Pred)</code>	
– <i>Description:</i> This predicate is understood natively by CiaoPP.	
native/2:	PROPERTY
<i>Meta-predicate</i> with arguments: <code>native(goal,?)</code> .	
General properties: <code>native(P, K)</code>	
– <i>The following properties hold globally:</i>	
<code>native(P,K)</code> is side-effect free.	(sideff/2)
Usage: <code>native(Pred, Key)</code>	
– <i>Description:</i> This predicate is understood natively by CiaoPP as <code>Key</code> .	

eval/1:**Usage:** `eval(Prop)`

- *Description:* `Prop` is evaluable at compile-time.

PROPERTY

equiv/2:*Meta-predicate* with arguments: `equiv(?,goal)`.**Usage:** `equiv(Prop1, Prop2)`

- *Description:* `Prop1` is equivalent to `Prop2`.

PROPERTY

8 Properties which are native to analyzers

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

Version: 1.11#309 (2005/3/16, 16:41:12 CET)

Version of last change: 1.11#144 (2003/12/31, 19:2:9 CET)

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.

8.1 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:*
 - `covered/2, linear/1, mshare/1, nonground/1, fails/1, not_fails/1, possibly_fails/1, covered/1, not_covered/1, is_det/1, non_det/1, possibly_nondet/1, mut_exclusive/1, not_mut_exclusive/1, size_lb/2, size_ub/2, size/2, size_o/2, steps_lb/2, steps_ub/2, steps/2, steps_o/2, finite_solutions/1, terminates/1.`
- **Other modules used:**
 - *System library modules:*
 - `andprolog/andprolog_rt, terms_check, terms_vars, sort, lists.`
 - *Internal (engine) modules:*
 - `term_basic, arithmetic, atomic_basic, attributes, mattr_global, basic_props, basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags, streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_support.`

8.2 Documentation on exports (`native_props`)

- covered/2:** PROPERTY
`covered(X, Y)`
 All variables occurring in X occur also in Y.
Usage: `covered(X, Y)`
- *Description:* X is covered by Y.
 - *The following properties hold globally:*
 - This predicate is understood natively by CiaoPP. (native/1)
- linear/1:** PROPERTY
`linear(X)`

X is bound to a term which is linear, i.e., if it contains any variables, such variables appear only once in the term. For example, $[1,2,3]$ and $f(A,B)$ are linear terms, while $f(A,A)$ is not.

Usage: `linear(X)`

- *Description:* 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)`

- *Description:* The sharing pattern is X .
- *The following properties should hold globally:*
This predicate is understood natively by CiaoPP as `sharing(X)`.

(native/2)

nonground/1:

PROPERTY

Usage: `nonground(X)`

- *Description:* X is not ground.
- *The following properties should hold globally:*
This predicate is understood natively by CiaoPP as `not_ground(X)`.

(native/2)

fails/1:

PROPERTY

`fails(X)`

Calls of the form X fail.

Usage: `fails(X)`

- *Description:* Calls of the form X fail.
- *The following properties hold globally:*
This predicate is understood natively by CiaoPP.

(native/1)

not_fails/1:

PROPERTY

`not_fails(X)`

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

Usage: `not_fails(X)`

- *Description:* All the calls of the form X do not fail.
- *The following properties hold globally:*
This predicate is understood natively by CiaoPP.

(native/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)
 – *Description:* Non-failure is not ensured for calls of the form X.
- 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)
 – *Description:* All the calls of the form X are covered.
- not_covered/1:** PROPERTY
 not_covered(X)
 There is some call of the form X for which there is not any clause whose test succeeds [DLGH97].
Usage: not_covered(X)
 – *Description:* Not all of the calls of the form X are covered.
- is_det/1:** PROPERTY
 is_det(X)
 All calls of the form X are deterministic, i.e. produce at most one solution, or not terminate.
Usage: is_det(X)
 – *Description:* All calls of the form X are deterministic.
- non_det/1:** PROPERTY
 non_det(X)
 All calls of the form X are not deterministic, i.e., produce several solutions.
Usage: non_det(X)
 – *Description:* All calls of the form X are not deterministic.
- 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)
 – *Description:* Non-determinism is not ensured for calls of the form X.

- 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)
 – *Description:* For any call of the form X at most one clause succeeds.
- not_mut_exclusive/1:** PROPERTY
 not_mut_exclusive(X)
 Not for all calls of the form X at most one clause succeeds. I.e. clauses are not disjoint for some call.
Usage: not_mut_exclusive(X)
 – *Description:* Not for all calls of the form X at most one clause succeeds.
- size_lb/2:** PROPERTY
 size_lb(X, Y)
 The minimum size of the terms to which the argument Y is bound to 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].
Usage: size_lb(X, Y)
 – *Description:* Y is a lower bound on the size of argument X.
- size_ub/2:** PROPERTY
 size_ub(X, Y)
 The maximum size of the terms to which the argument Y is bound to 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].
Usage: size_ub(X, Y)
 – *Description:* Y is an upper bound on the size of argument X.
- size/2:** PROPERTY
Usage: size(X, Y)
 – *Description:* Y is the size of argument X.
- size_o/2:** PROPERTY
Usage: size_o(X, Y)
 – *Description:* The size of argument X is in the order of Y.

- 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)
 – *Description:* Y is a lower bound on the cost of any call of the form X.
- 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)
 – *Description:* Y is an upper bound on the cost of any call of the form X.
- 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)
 – *Description:* Y is the cost (number of resolution steps) of any call of the form X.
- steps_o/2:** PROPERTY
Usage: steps_o(X, Y)
 – *Description:* Y is the complexity order of the cost of any call of the form X.
- finite_solutions/1:** PROPERTY
 finite_solutions(X)
 Calls of the form X produce a finite number of solutions [DLGH97].
Usage: finite_solutions(X)
 – *Description:* All the calls of the form X have a finite number of solutions.
- terminates/1:** PROPERTY
 terminates(X)
 Calls of the form X always terminate [DLGH97].
Usage: terminates(X)
 – *Description:* All the calls of the form X terminate.
- indep/1:** PROPERTY
Usage: indep(X)
 – *Description:* 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
Usage: `indep(X, Y)`
– *Description:* 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/2:** PROPERTY
Usage: `instance(Term1, Term2)`
– *Description:* Term1 is an instance of Term2.
– *The following properties hold globally:*
This predicate is understood natively by CiaoPP. (native/1)

9 Run-time checking of assertions

Author(s): David Trallero Mena.

Version: 1.11#309 (2005/3/16, 16:41:12 CET)

This library package can be used to perform run-time checking of assertions. Properties are checked during execution of the program and errors found (when the property does not hold) are reported.

9.1 Usage and interface (rtchecks)

- **Library usage:**

```
:- use_package(rtchecks).
or
:- module(...,...,[rtchecks]).
```
- **Exports:**
 - *Predicates:*

```
check/1.
```
- **Other modules used:**
 - *System library modules:*

```
assertions/assertions_props.
```
 - *Internal (engine) modules:*

```
term_basic, arithmetic, atomic_basic, attributes, mattr_global, basic_props,
basiccontrol, data_facts, exceptions, io_aux, io_basic, prolog_flags,
streams_basic, system_info, term_compare, term_typing, hiord_rt, debugger_
support.
```

9.2 Documentation on exports (rtchecks)

check/1:

PREDICATE

See Chapter 4 [The Ciao assertion package], page 31.

Usage: `check(Prop)`

- *Description:* `Prop` is checked. If it fails, an exception is raised.
- *The following properties should hold at call time:*

`Prop` 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)

References

- [BCC04] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. L'opez-Garc'ia, and G. Puebla (Eds.).
The Ciao System. Reference Manual (v1.10).
Technical Report CLIP3/97.1.10(04), School of Computer Science (UPM), August 2004.
Available at <http://clip.dia.fi.upm.es/Software/Ciao/>.
- [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.
- [BdlBH94] F. Bueno, M. Garc'ia 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. L'opez-Garc'ia, 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.
- [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. Garc'ia 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.
- [dlBHM00] M. Garc'ia 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.
- [DLGH97] S.K. Debray, P. L'opez-Garc'ia, 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.
- [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.
- [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, 22:715–734, 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, 22:715–734, 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 *1991 International Conference on Logic Programming*, 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.
- [PBH97] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Debugging of Constraint Logic Programs.
In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997.
Available from ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_discipldeliv.ps.gz as technical report CLIP2/97.1.
- [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. Sondergaard.
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*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.

[VHCLC95]

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.

Predicate/Method Definition Index

A

acheck/0 8
 again/0 19
 analyze/1 9
 auto_analyze/1 18
 auto_check_assert/1 18
 auto_optimize/1 18

C

check/1 37, 73
 current_pp_flag/2 6
 customize/1 18
 customize_and_exec/1 19

F

false/1 38

G

get_menu_configs/1 19

H

help/0 14

M

module/1 8

O

output/0 9
 output/1 9

P

pop_pp_flag/1 6
 pp_flag/1 6
 push_pp_flag/2 6

R

remove_menu_config/1 19
 restore_menu_config/1 19

S

save_menu_config/1 19
 set_pp_flag/2 6
 show_menu_config/1 20
 show_menu_configs/0 19

T

transform/1 8
 true/1 37
 trust/1 37

Regular Type Definition Index

A

assrt_body/1.....	39
assrt_status/1.....	44
assrt_type/1.....	44
atm/1.....	55
atm_or_atm_list/1.....	62

C

c_assrt_body/1.....	42
callable/1.....	57
character_code/1.....	61
complex_arg_property/1.....	41
complex_goal_property/1.....	42
constant/1.....	57

D

dictionary/1.....	42
-------------------	----

F

flt/1.....	54
------------	----

G

g_assrt_body/1.....	43
gnd/1.....	56

I

int/1.....	54
------------	----

L

list/1.....	58
list/2.....	59

N

nnegint/1.....	54
num/1.....	55

O

operator_specifier/1.....	57
---------------------------	----

P

predfunctor/1.....	44
predname/1.....	61
property_conjunction/1.....	41
property_starterm/1.....	41
propfunctor/1.....	44

S

s_assrt_body/1.....	43
sequence/2.....	60
sequence_or_list/2.....	60
string/1.....	61
struct/1.....	56

T

term/1.....	53
-------------	----

Concept Definition Index

A

acceptable modes 40
 assertion body syntax 39, 42, 43
 assertion checking 5

C

calls assertion 33
 check assertion 37
 comment assertion 36
 comments, machine readable 31
 comp assertion 34
 compatibility properties 47

D

data declaration 27
 debugging 5
 decl assertion 36
 dynamic declaration 27

E

entry assertion 35
 entry declaration 27

F

false assertion 38
 formatting commands 31

I

instantiation properties 47
 ISO-Prolog 21

M

module declaration 28

P

parametric type functor 50
 pred assertion 32, 33
 program transformations 5
 prop assertion 34, 35
 properties of computations 47
 properties of execution states 47
 properties, basic 53
 properties, native 67

R

regtype assertion 50, 51
 regular type expression 50
 run-time tests 5

S

sharing sets 68
 specifications 5
 static debugging 5
 success assertion 33, 34

T

true assertion 37
 trust assertion 37
 trust assertions 26

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. Note that due to limitations of the `info` format unfortunately only the first reference will appear in online versions of the document.

,		
,',/2	60	
*		
*/2	41, 42	
:		
::/2	32	
=		
=>/2	32	
+		
+/1	40	
+/2	40	
A		
acceptable modes	40	
acheck/0	8	
add_action/1	8	
again/0	18, 19	
aggregates	18	
analysis/1	9	
analyze/1	9	
analyzer output	37, 38	
andprolog/andprolog_rt	67	
arithmetic	5, 18, 32, 33, 39, 53, 67, 73	
assertion body syntax	39, 42, 43	
assertion checking	5	
assertion language	1, 3	
assertions	31, 32, 39	
assertions/assertions_props	32, 50, 73	
assrt_body/1	32, 39	
assrt_status/1	39, 44	
assrt_type/1	39, 44	
atm/1	53, 55	
atm_or_atm_list/1	53, 62	
atomic_basic	5, 18, 32, 39, 53, 67, 73	
attributes	5, 18, 32, 39, 53, 67, 73	
auto_analyze/1	17, 18	
auto_check_assert/1	18	
auto_check_assertions/1	17, 19	
auto_interface(auto_help)	5	
auto_interface(auto_interface)	5	
auto_optimize/1	17, 18, 19	
B		
basic_props	5, 18, 32, 39, 67, 73	
basic_props:regtype/1	47	
basiccontrol	5, 18, 32, 39, 53, 67, 73	
C		
c_assrt_body/1	39, 42	
call/1	42	
callable/1	53, 57	
calls assertion	33	
calls/1	32, 33, 35	
calls/2	32, 33	
character string	31	
character_code/1	53, 61	
check assertion	37	
check/1	32, 37, 38, 73	
checking the assertions	1, 3	
ciaopp	5, 67	
ciaopp(driver)	5, 18	
ciaopp(menu_generator)	18	
ciaopp(preprocess_flags)	5, 18	
ciaopp(printer)	5, 18	
CIAOPPSETTINGS.pl	3	
comment assertion	36	
comment string	40, 42, 43, 44	
comment/2	32, 36	
comments, machine readable	31	
comp assertion	34	
comp/1	32, 34, 43	
comp/2	32, 34	
compat/2	53, 62	
compatibility properties	47	
compatible	40	
complex argument property	40, 41, 42, 43	
complex goal property	40, 42, 43	
complex_arg_property/1	39, 40, 41, 42, 43	
complex_goal_property/1	39, 40, 42, 43	
computational cost	1, 3	
constant/1	53, 57	

covered/1 67, 69
 covered/2 67
 current_pp_flag/2 6
 customize/1 18
 customize_and_exec/1 17, 18, 19

D

data declaration 27
 data_facts 5, 18, 32, 39, 53, 67, 73
 dcg_expansion 39
 debugger_support 5, 18, 32, 39, 53, 67, 73
 debugging 5
 decl assertion 36
 decl/1 32, 36, 39
 decl/2 32, 36
 determinacy 1, 3
 dictionary/1 39, 42
 docstring/1 31, 39, 40, 42, 43, 44
 driver 8, 17
 dynamic declaration 27

E

emacs 17
 entry assertion 35
 entry declaration 27
 entry/1 32, 35, 42
 equiv/2 53, 65
 eval/1 53, 64
 exceptions 5, 18, 32, 39, 53, 67, 73
 exit/1 32, 36
 exit/2 32, 36

F

fails/1 67, 68
 false assertion 38
 false/1 32, 38
 finite_solutions/1 67, 71
 flt/1 53, 54
 formatting commands 31
 func/1 43

G

g_assrt_body/1 39, 43
 get_menu_configs/1 19
 gnd/1 53, 56
 GNU general public license 1, 3
 granularity control 1, 3
 ground/1 68

H

head pattern 39, 40, 43
 head_pattern/1 39, 40, 43
 help/0 14
 hiord_rt 5, 18, 32, 39, 53, 67, 73

I

indep/1 68, 71
 indep/2 68, 72
 infer(infer) 5
 Inference of properties 1, 3
 inst/2 53, 63
 instance/2 72
 instantiation properties 47
 int/1 53, 54
 integer/1 41
 inter-modular analysis 14
 io_aux 5, 18, 32, 39, 53, 67, 73
 io_basic 5, 18, 32, 39, 53, 67, 73
 is_det/1 67, 69
 ISO-Prolog 21
 iso/1 53, 63

L

library(basicmodes) 40
 library(isomodes) 40
 linear/1 67
 list/1 53, 58, 62
 list/2 41, 53, 59
 lists 18, 32, 34, 67
 lpdoc 1, 3, 31, 36, 40, 45

M

mattr_global 5, 18, 32, 39, 53, 67, 73
 member/2 53, 59
 messages 5, 18
 mode 32, 40
 modedef/1 32, 35, 40
 modes 1, 3
 module declaration 28
 module/1 8
 mshare/1 67, 68
 mut_exclusive/1 67, 69

N

n_assrt_body/5 43, 44
 nabody/1 39, 42
 native/1 53, 64
 native/2 53, 64
 nnegint/1 53, 54
 non-failure 1, 3
 non_det/1 67, 69
 nonground/1 67, 68
 not_covered/1 67, 69
 not_fails/1 67, 68
 not_further_inst/1 42
 not_further_inst/2 53, 63
 not_mut_exclusive/1 67, 70
 num/1 53, 55

O

operator_specifier/1 53, 57
 output/0 9
 output/1 9

P

parametric type functor 50
 Partial deduction 16
 partial evaluation 1, 3, 16
 pop_pp_flag/1 6
 possibly_fails/1 67, 69
 possibly_nondet/1 67, 69
 pp_flag/1 6
 pred assertion 32, 33
 pred/1 32, 33, 34, 36, 39, 43

pred/2 32, 33
 predfunctor/1 39, 44
 predname/1 40, 53, 61
 printer 17
 program assertions 31
 program parallelization 1, 3
 program specialization 1, 3
 program transformations 1, 3, 5
 program(p_asr) 5
 prolog_flags 5, 18, 32, 39, 53, 67, 73
 prop assertion 34, 35
 prop/1 32, 34, 35
 prop/2 32, 35
 properties of computations 47
 properties of execution states 47
 properties, basic 53
 properties, native 67
 property 34
 property compatibility 62
 property_conjunction/1 39, 41
 property_starterm/1 39, 41
 propfunctor/1 39, 44
 providing information to the compiler 35, 37
 push_pp_flag/2 6

R

regtype assertion 50, 51
 regtype/1 50, 51, 53, 64
 regtype/2 50, 51
 regular type 50
 regular type definitions 47
 regular type expression 50
 regular types 47
 remove_action/1 8
 remove_menu_config/1 19
 restore_menu_config/1 19
 run-time checks 35
 run-time tests 1, 3, 5

S

s_assrt_body/1 39, 43
 save_menu_config/1 19, 20
 sequence/2 53, 60
 sequence_or_list/2 53, 60
 set_pp_flag/2 6
 sharing sets 68
 show_menu_config/1 20
 show_menu_configs/0 19
 sideff/2 53, 63
 size/2 67, 70
 size_lb/2 67, 70
 size_o/2 67, 70
 size_ub/2 67, 70
 sizes of terms 1, 3
 sort 67
 specifications 1, 3, 5, 31
 static debugging 1, 3, 5
 steps/2 67, 71
 steps_lb/2 67, 70
 steps_o/2 67, 71
 steps_ub/2 67, 71
 streams_basic 5, 18, 32, 39, 53, 67, 73
 string/1 53, 61
 stringcommand/1 36, 40, 42, 43, 44, 45
 struct/1 53, 56
 success assertion 33, 34
 success/1 32, 33, 34
 success/2 32, 34
 syntax of regular types 47
 system 5

system_info 5, 18, 32, 39, 53, 67, 73

T

term/1 53
 term_basic 5, 18, 32, 39, 50, 53, 67, 73
 term_compare 5, 18, 32, 39, 53, 67, 73
 term_typing 5, 18, 32, 39, 53, 67, 73
 terminates/1 67, 71
 terms_check 53, 67
 terms_vars 67
 transform/1 8
 transformation/1 12
 true assertion 37
 true/1 32, 37
 trust assertion 37
 trust assertions 26
 trust/1 32, 37
 types 1, 3
 typeslib(typeslib) 5

U

usage 32

V

valid_flag_value/2 7
 var/1 41
 variable instantiation 1, 3, 5
 variable names 31