Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor)

Manuel V. Hermenegildo[a,b,*], Germán Puebla[a], Francisco Bueno[a], Pedro López-García[a]

[a] *Department of Computer Science, Technical University of Madrid (UPM), Spain*
[b] *Departments of Computer Science and Electrical and Computer Engineering, University of New Mexico, United States*

Abstract

The technique of Abstract Interpretation has allowed the development of very sophisticated global program analyses which are at the same time provably correct and practical. We present in a tutorial fashion a novel program development framework which uses abstract interpretation as a fundamental tool. The framework uses modular, incremental abstract interpretation to obtain information about the program. This information is 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 and simplify run-time tests, and to perform high-level program transformations such as multiple abstract specialization, parallelization, and resource usage control, all in a provably correct way. In the case of validation and debugging, the assertions can refer to a variety of program points such as procedure entry, procedure exit, points within procedures, or global computations. The system can reason with much richer information than, for example, traditional types. This includes data structure

[*] Corresponding author at: School of Computer Science, Technical University of Madrid, Campus de Montegancedo, 28660 Boadilla del Monte, Madrid, Spain. Tel.: +34 91 336 7435; fax: +34 91 352 4819.

E-mail addresses: herme@fi.upm.es, herme@unm.edu (M.V. Hermenegildo), german@fi.upm.es (G. Puebla), bueno@fi.upm.es (F. Bueno), pedro.lopez@fi.upm.es (P. López-García).

URLs: http://www.clip.dia.fi.upm.es/~herme, http://www.unm.edu/~herme (M.V. Hermenegildo), http://www.clip.dia.fi.upm.es/~german (G. Puebla), http://www.clip.dia.fi.upm.es/~bueno (F. Bueno), http://www.clip.dia.fi.upm.es/~pedro (P. López-García).

shape (including pointer sharing), 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). CiaoPP, the preprocessor of the Ciao multi-paradigm programming system, which implements the described functionality, will be used to illustrate the fundamental ideas.

1. Introduction

The technique of Abstract Interpretation [13] 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 [3].[1] 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., [40,31,19,4,21,27] and their references). These techniques and systems 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,

[1] The 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.

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 2) and then illustrate it by presenting what is arguably the first and most complete implementation of this idea: CiaoPP [41,27].[2] 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 3 presents CiaoPP at work performing program analysis, while Section 4 does the same for program debugging and validation, and Section 5 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.

2. 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 $[\![P]\!]$) is defined as the least fixpoint of the S_P operator, i.e., $[\![P]\!] = \text{lfp}(S_P)$. 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.

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) [13].

[2] In 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 [41].

Table 1
Set theoretic formulation of verification problems

Property	Definition
P is partially correct w.r.t. I	$[\![P]\!] \subseteq I$
P is complete w.r.t. I	$I \subseteq [\![P]\!]$
P is incorrect w.r.t. I	$[\![P]\!] \not\subseteq I$
P is incomplete w.r.t. I	$I \not\subseteq [\![P]\!]$

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 $[\![P]\!]$, 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 $[\![P]\!]_\alpha$.

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

2.1. Abstract verification and debugging

Both program verification and debugging compare the *actual semantics* of the program, i.e., $[\![P]\!]$, 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 $[\![P]\!]$ 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 [5,28,43] is to use the abstract approximation $[\![P]\!]_\alpha$ 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 [33]. Abstract interpretation for debugging of imperative programs has been studied by Bourdoncle [2], by Comini et al. for the particular case of algorithmic debugging of logic programs [11] (making use of partial specifications) [10], and very recently by Cousot [12].

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

Table 2
Validation problems using approximations

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	$I_\alpha \not\subseteq \alpha(\llbracket P \rrbracket)$	$I_\alpha \not\subseteq \llbracket P \rrbracket_{\alpha^+}$

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* [42]), 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, 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 a 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 with program points, it is often possible to use the this information directly and/or the analysis graph itself to locate the earliest program point where the symptom occurs (see Section 4). 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 4.

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 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 [5,28,43,41].

2.2. 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* [23,46]. 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 $TS(L, P) = \{\theta|_L : L\theta$ succeeds exactly once in P 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 $FF(L, P) = \{\theta|_L : L\theta$ 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
Abstract executability

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

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 $A_{TS}(\overline{L}, D_\alpha)$ and $A_{FF}(\overline{L}, D_\alpha)$ where $\overline{L}$ 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 $\overline{L}$ trivially succeeds (resp. finitely fails). For soundness it is required that $\forall\lambda \in A_{TS}(\overline{L}, D_\alpha)\; \gamma(\lambda) \subseteq TS(\overline{L}, P)$ and $\forall\lambda \in A_{FF}(\overline{L}, D_\alpha)\; \gamma(\lambda) \subseteq FF(\overline{L}, P)$.

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 [47] (Section 5) and the optimization of delay conditions in logic programs with dynamic procedure call scheduling order [44].

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 [47,49,14,32]. 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.

3. 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 [40,6], including extensions for, e.g., incrementality [29,45], modularity [4,48,7], analysis of constraints [21], and analysis of concurrency [37].

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 [4]. A unified language of assertions [4,42] 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.

3.1. 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 Fig. 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" [39]) of the module using CiaoPP yields the following results[3]:

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

[3] In 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).
```

Fig. 1. A modular qsort program.

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).

3.2. 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 [4]. For example, if we commented out the `use_module/2` declaration in Fig. 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 4. Interestingly, this very general concept of assertions is also particularly useful for generating documentation automatically (see [24] 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 Fig. 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* [51,15,19,20,50], 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]).
```

3.3. Type analysis

CiaoPP can infer (parametric) types for programs both at the predicate level and at the literal level [19,20,50]. The output for Fig. 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 [4] 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 to 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) ).
```

3.4. Non-failure and determinacy analysis

CiaoPP includes a non-failure analysis, based on [16] and [8], 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 [34], 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 Fig. 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.)

3.5. Size, cost, and termination analysis

CiaoPP can also infer lower and upper bounds on the sizes of terms and the computational cost of predicates [17,18]. 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.

3.6. 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.*

4. 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 [28,41] which involves several of the tools which comprise CiaoPP. Fig. 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.

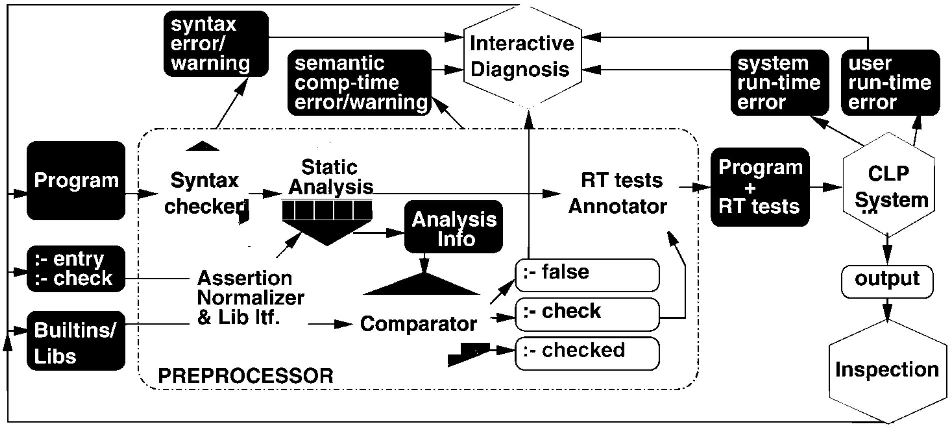

Fig. 2. Architecture of the preprocessor.

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

4.1. Static debugging

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

```
:- 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.

4.2. 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,

```
:- 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).
```

Fig. 3. A tentative qsort program.

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 Fig. 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`.[4]

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 [5], 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)`.

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

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)
```

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.

4.3. 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[5]:

```
:- 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

[5] The `check` prefix is assumed when no prefix is given, as in the example shown.

run-time checking. They can also be restricted to a subset of the calls using “:”. More details on the assertion language can be found in [42].

CiaoPP can perform compile-time checking of the assertions above, by comparing them with the assertions inferred by analysis (see Table 2 and [5,43] for details), producing as output the following assertions (refer also to Fig. 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 [1]. 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.[6]

4.4. 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

[6] Note 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 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.

checks" option. CiaoPP will then introduce run-time tests in the program for those `calls` and `success` assertions which have not been proved or disproved during compile-time (see again Fig. 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 Fig. 1.

4.5. 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 Fig. 4. Assume also that the predicate `append` is defined as in Fig. 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).
:- 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).
```

Fig. 4. The naive reverse program.

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 $0.5\ (\mathit{length}(A))^2 + 1.5\ \mathit{length}(A) + 1$ resolution steps (which is the cost analysis output), while the assertion requires that it take at most $\mathit{length}(A) + 1$ resolution steps. The cost function in the user-provided assertion is compared with the lower-bound cost assertion inferred by analysis. This allows detection of the inconsistency and proof 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 a such case, if the upper bound computed by analysis is lower than or equal to 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 [25].

5. 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.

5.1. 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 2.2.

CiaoPP can also apply abstract specialization to the optimization of programs with dynamic scheduling (e.g., using `delay` declarations) [44]. 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 [26].

5.2. Parallelization

An example of a non-trivial program optimization performed using abstract interpretation in CiaoPP is program parallelization [6]. 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 [38], exploit parallelism under certain *independence* conditions, which allow guaranteeing interesting correctness and no-slowdown properties for the parallelized programs [30,22]. This process is complicated by the presence of shared variables and pointers among data structures at run-time.

We consider again the program of Fig. 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 3), 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 [6] 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 parametrized on the notion of independence. Different tests can be used for different independence notions: non-strict independence [9], constraint-based independence [22], 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.[7]

5.3. 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 [36] 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 threshold[8] 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 Fig. 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

[7] For example, stream and-parallelism can be seen as independent and-parallelism if the independence of "bindings" rather than goals is considered.

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

length of its input list is not 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.[9] 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) [35].

5.4. 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 [47] 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 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

[9] This size threshold will obviously be different if the cost function is.

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 [47]. 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.

5.5. 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 [49] 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 [50] for their use for partial evaluation.

More info: For more information, full versions of papers and technical reports, and/or to download Ciao and other related systems please access: http://www.cliplab.org/.

References

[1] E. Albert, G. Puebla, M. Hermenegildo, An abstract interpretation-based approach to mobile code safety, in: Proc. of Compiler Optimization meets Compiler Verification, COCV'04, April 2004.

[2] F. Bourdoncle, Abstract debugging of higher-order imperative languages, in: Programming Languages Design and Implementation'93, 1993, pp. 46–55.

[3] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, 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), 1997. System and on-line version of the manual available at http://clip.dia.fi.upm.es/Software/Ciao/.

[4] F. Bueno, D. Cabeza, M. Hermenegildo, G. Puebla, Global analysis of standard prolog programs, in: European Symposium on Programming, April 1996, Sweden, in: LNCS, vol. 1058, Springer-Verlag, 1996, pp. 108–124.

[5] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, 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, May 1997, Linköping, Sweden, U. of Linköping Press, 1997, pp. 155–170.

[6] F. Bueno, M. García de la Banda, M. Hermenegildo, Effectiveness of abstract interpretation in automatic parallelization: A case study in logic programming, ACM Transactions on Programming Languages and Systems 21 (2) (1999) 189–238.

[7] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, P. Stuckey, A model for inter-module analysis and optimizing compilation, in: Logic-based Program Synthesis and Transformation, March 2001, in: LNCS, vol. 2042, Springer-Verlag, 2001, pp. 86–102.

[8] F. Bueno, P. López-García, M. Hermenegildo, Multivariant non-failure analysis via standard abstract interpretation, in: 7th International Symposium on Functional and Logic Programming, FLOPS 2004, April 2004, Heidelberg, Germany, in: LNCS, vol. 2998, Springer-Verlag, 2004, pp. 100–116.

[9] D. Cabeza, M. Hermenegildo, Extracting non-strict independent and-parallelism using sharing and freeness information, in: 1994 International Static Analysis Symposium, September 1994, Namur, Belgium, in: LNCS, vol. 864, Springer-Verlag, 1994, pp. 297–313.

[10] M. Comini, G. Levi, M.C. Meo, G. Vitiello, Abstract diagnosis, Journal of Logic Programming 39 (1–3) (1999) 43–93.

[11] M. Comini, G. Levi, G. Vitiello, Declarative diagnosis revisited, in: 1995 International Logic Programming Symposium, December 1995, Portland, Oregon, MIT Press, Cambridge, MA, 1995, pp. 275–287.

[12] P. Cousot, Automatic verification by abstract interpretation, invited tutorial, in: Fourth International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI, January 2003, in: LNCS, vol. 2575, Springer, 2003, pp. 20–24.

[13] P. Cousot, 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, 1977, pp. 238–252.

[14] P. Cousot, R. Cousot, Systematic design of program transformation frameworks by abstract interpretation, in: 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'02, January 2002, Portland, Oregon, ACM, 2002, pp. 178–190.

[15] P.W. Dart, J. Zobel, A regular type language for logic programs, in: Types in Logic Programming, MIT Press, 1992, pp. 157–187.

[16] S.K. Debray, P. López-García, M. Hermenegildo, Non-failure analysis for logic programs, in: 1997 International Conference on Logic Programming, June 1997, Cambridge, MA, MIT Press, Cambridge, MA, 1997, pp. 48–62.

[17] S.K. Debray, P. López-García, M. Hermenegildo, N.-W. Lin, Estimating the computational cost of logic programs, in: Static Analysis Symposium, SAS'94, September 1994, Namur, Belgium, in: LNCS, vol. 864, Springer-Verlag, 1994, pp. 255–265.

[18] S.K. Debray, P. López-García, M. Hermenegildo, N.-W. Lin, Lower bound cost estimation for logic programs, in: 1997 International Logic Programming Symposium, October 1997, Cambridge, MA, MIT Press, 1997, pp. 291–305.

[19] J.P. Gallagher, D.A. de Waal, Fast and precise regular approximations of logic programs, in: P.V. Hentenryck (Ed.), Proc. of the 11th International Conference on Logic Programming, MIT Press, 1994, pp. 599–613.

[20] J. Gallagher, 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, January 2002, in: LNCS, vol. 2257, Springer-Verlag, 2002, pp. 243–261.

[21] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, W. Simoens, Global analysis of constraint logic programs, ACM Transactions on Programming Languages and Systems 18 (5) (1996) 564–615.

[22] M. García de la Banda, M. Hermenegildo, K. Marriott, Independence in CLP languages, ACM Transactions on Programming Languages and Systems 22 (2) (2000) 269–339.

[23] F. Giannotti, M. Hermenegildo, A technique for tecursive invariance detection and selective program specialization, in: Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming, August 1991, in: LNCS, vol. 528, Springer-Verlag, 1991, pp. 323–335.

[24] M. Hermenegildo, A documentation generator for (C)LP systems, in: International Conference on Computational Logic, CL2000, July 2000, in: LNAI, vol. 1861, Springer-Verlag, 2000, pp. 1345–1361.

[25] M. Hermenegildo, E. Albert, P. López-García, G. Puebla, Some techniques for automated, resource-aware distributed and mobile computing in a multi-paradigm programming system, in: Proc. of EURO–PAR 2004, August 2004, in: LNCS, vol. 3149, Springer-Verlag, 2004, pp. 21–37.

[26] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, 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, Commack, NY, USA, April 1999, Nova Science, 1999, pp. 65–85.

[27] M. Hermenegildo, F. Bueno, G. Puebla, P. López-García, Program analysis, debugging and optimization using the Ciao system preprocessor, in: 1999 Int'l Conference on Logic Programming, November 1999, Cambridge, MA, MIT Press, 1999, pp. 52–66.

[28] M. Hermenegildo, G. Puebla, 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, D.S. Warren (Eds.), The Logic Programming Paradigm: a 25-Year Perspective, Springer-Verlag, 1999, pp. 161–192.

[29] M. Hermenegildo, G. Puebla, K. Marriott, P. Stuckey, Incremental analysis of constraint logic programs, ACM Transactions on Programming Languages and Systems 22 (2) (2000) 187–223.

[30] M. Hermenegildo, F. Rossi, Strict and non-strict independent and-parallelism in logic programs: correctness, efficiency, and compile-time conditions, Journal of Logic Programming 22 (1) (1995) 1–45.

[31] B. Le Charlier, P. Van Hentenryck, Experimental evaluation of a generic abstract interpretation algorithm for prolog, ACM Transactions on Programming Languages and Systems 16 (1) (1994) 35–101.

[32] M. Leuschel, Program specialisation and abstract interpretation reconciled, in: Joint International Conference and Symposium on Logic Programming, June 1998.

[33] Y. Lichtenstein, E.Y. Shapiro, Abstract algorithmic debugging, in: R.A. Kowalski, K.A. Bowen (Eds.), Fifth International Conference and Symposium on Logic Programming, August 1988, Seattle, WA, MIT, 1988, pp. 512–531.

[34] P. López-García, F. Bueno, 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, Springer-Verlag, August 2004 (in press).

[35] P. López-García, M. Hermenegildo, Efficient term size computation for granularity control, in: International Conference on Logic Programming, June 1995, Cambridge, MA, MIT Press, Cambridge, MA, 1995, pp. 647–661.

[36] P. López-García, M. Hermenegildo, S.K. Debray, A methodology for granularity based control of parallelism in logic programs, in: Parallel Symbolic Computation, Journal of Symbolic Computation 22 (1996) 715–734 (special issue).

[37] K. Marriott, M. García de la Banda, M. Hermenegildo, Analyzing logic programs with dynamic scheduling, in: 20th Annual ACM Conf. on Principles of Programming Languages, January 1994, ACM, 1994, pp. 240–254.

[38] K. Muthukumar, F. Bueno, M. García de la Banda, M. Hermenegildo, Automatic compile-time parallelization of logic programs for restricted, goal-level, independent and-parallelism, Journal of Logic Programming 38 (2) (1999) 165–218.

[39] K. Muthukumar, M. Hermenegildo, Combined determination of sharing and freeness of program variables through abstract interpretation, in: 1991 International Conference on Logic Programming, June 1991, MIT Press, 1991, pp. 49–63.

[40] K. Muthukumar, M. Hermenegildo, Compile-time derivation of variable dependency using abstract interpretation, Journal of Logic Programming 13 (2–3) (1992) 315–347.

[41] G. Puebla, F. Bueno, M. Hermenegildo, A generic preprocessor for program validation and debugging, in: P. Deransart, M. Hermenegildo, J. Maluszynski (Eds.), Analysis and Visualization Tools for Constraint Programming, September 2000, in: LNCS, vol. 1870, Springer-Verlag, 2000, pp. 63–107.

[42] G. Puebla, F. Bueno, M. Hermenegildo, An assertion language for constraint logic programs, in: P. Deransart, M. Hermenegildo, J. Maluszynski (Eds.), Analysis and Visualization Tools for Constraint Programming, September 2000, in: LNCS, vol. 1870, Springer-Verlag, 2000, pp. 23–61.

[43] G. Puebla, F. Bueno, M. Hermenegildo, Combined static and dynamic assertion-based debugging of constraint logic programs, in: Logic-based Program Synthesis and Transformation, LOPSTR'99, in: LNCS, vol. 1817, Springer-Verlag, 2000, pp. 273–292.

[44] G. Puebla, M. García de la Banda, K. Marriott, P. Stuckey, Optimization of logic programs with dynamic scheduling, in: 1997 International Conference on Logic Programming, June 1997, Cambridge, MA, MIT Press, 1997, pp. 93–107.

[45] G. Puebla, M. Hermenegildo, Optimized algorithms for the incremental analysis of logic programs, in: International Static Analysis Symposium, September 1996, in: LNCS, vol. 1145, Springer-Verlag, 1996, pp. 270–284.

[46] G. Puebla, M. Hermenegildo, Abstract specialization and its application to program parallelization, in: J. Gallagher (Ed.), Logic Program Synthesis and Transformation, in: LNCS, vol. 1207, Springer-Verlag, 1997, pp. 169–186.

[47] G. Puebla, M. Hermenegildo, Abstract multiple specialization and its application to program parallelization, in: Synthesis, Transformation and Analysis of Logic Programs, Journal of Logic Programming 41 (2–3) (1999) 279–316 (special issue).

[48] G. Puebla, M. Hermenegildo, Some issues in analysis and specialization of modular Ciao-prolog programs, in: Optimization and Implementation of Declarative Programming Languages, in: Electronic Notes in Theoretical Computer Science, vol. 30, Elsevier, North-Holland, 2000 (special issue).

[49] G. Puebla, M. Hermenegildo, J. Gallagher, An integration of partial evaluation in a generic abstract interpretation framework, in: O. Danvy (Ed.), ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'99, January 1999, in: BRISC Series, vol. NS-99-1, University of Aarhus, Denmark, 1999, pp. 75–85.

[50] C. Vaucheret, F. Bueno, More precise yet efficient type inference for logic programs, in: International Static Analysis Symposium, September 2002, in: LNCS, vol. 2477, Springer-Verlag, 2002, pp. 102–116.

[51] E. Yardeni, E. Shapiro, A type system for logic programs. Concurrent Prolog: Collected Papers, 1987, pp. 211–244.