A Tutorial on Program Development and Optimization using the Ciao Preprocessor

Abstract

We present in a tutorial fashion CiaoPP, the preprocessor of the Ciao multi-paradigm programming system, which implements 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 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).

1 Introduction

We describe in a tutorial fashion CiaoPP, an implementation of 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.

CiaoPP is the preprocessor of the Ciao program development system [3]. 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., [37, 10, 20, 4, 12, 23, 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, 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 is a standalone preprocessor to the standard clause-level compiler. It performs source-to-source transformations. The input to CiaoPP are logic programs (optionally with assertions and syntactic extensions). The output are error/warning messages plus the transformed logic program, with:

- Results of analysis (as assertions).
- Results of static checking of assertions.
- Assertion run-time checking code.
- Optimizations (specialization, parallelization, etc.)

By design, CiaoPP is a generic tool that can be easily customized to different programming systems and dialects and allows 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 [39].

This tutorial is organized as follows: Section 2 gives the “getting started” basics, Section 3 presents CiaoPP at work for program transformation and optimization, while Section 4 does the same for program debugging and validation, and Section 5 shows how CiaoPP performs program analysis.
2 Getting Started

A CiaoPP session consists in the preprocessing of a file. The session is governed by a menu, where you can choose the kind of preprocessing you want to be done to your file among several analyses and program transformations available. Clicking on the icon 📜 in the buffer containing the file to be preprocessed displays the menu, which will look (depending on the options available in the current CiaoPP version) something like the “Preprocessor Option Browser” shown in Figure 1.

```
module(app,[app/3],[assertions]).
entry app(A,B,C) : (list(A), list(B), var(C)).
app([X|Xs],Ys,[X|Zs]) :-
    app(Xs,Ys,Zs).
```

Figure 1: Starting menu for browsing CiaoPP options.

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

### 3 Source Program Optimization

We first turn our attention to the program optimizations that are available in CiaoPP. These include abstract specialization, multiple program specialization, integration of abstract interpretation and partial evaluation, and parallelization (including granularity control). 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 (See Section 5 for a tutorial on program analysis with CiaoPP).

#### 3.1 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 following definition of the property `sorted_num_list/1`:

```prolog
:- 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]).
```

and assume that regular type analysis infers that `sorted_num_list/1` will always be called with its argument bound to a list of integers. Abstract specialization can use this information to optimize the code into:

```prolog
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 [27].
3.2 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 abstract 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 [43] has the advantage that it can be incorporated with little or no modification of some existing abstract interpreters, provided they are multivariant (the abstract interpreter that CiaoPP uses, called PLAI [37, 5], has this property, see Section 5 for details).

This specialization can be used for example to improve automatic parallelization in those cases where run-time tests are included in the resulting program (see Section 3.6 for a tutorial on parallelization). 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 sort example used in Section 3.6. 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:

```prolog
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 \texttt{multiply} use the optimized version \texttt{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 [43]. 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 \texttt{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.

### 3.3 Basic Partial Evaluation:

The main purpose of \textit{partial evaluation} (see [28] for a general text on the area) is to specialize a given program w.r.t. part of its input data—hence it is also known as \textit{program specialization}. Essentially, partial evaluators are non-standard interpreters which evaluate expressions while enough information is available and residualize them (i.e. leave them in the resulting program) otherwise. The partial evaluation of logic programs is usually known as \textit{partial deduction} [30, 19]. Informally, the partial deduction algorithm proceeds as follows. Given an input program and a set of atoms, the first step consists in applying an \textit{unfolding rule} to compute finite (possibly incomplete) SLD trees for these atoms. This step returns a set of \textit{resultants} (or residual rules), i.e., a program, associated to the root-to-leaf derivations of these trees. Then, an \textit{abstraction operator} is applied to properly add the atoms in the right-hand sides of resultants to the set of atoms to be partially evaluated. The abstraction phase yields a new set of atoms, some of which may in turn need further evaluation and, thus, the process is iteratively repeated while new atoms are introduced.

We show a simple example where Partial Evaluation is used to specialize a program w.r.t. known input data. In this case, the entry declaration states that calls to append will be performed with a list starting by the prefix \([1, 2, 3]\) always. The user program will look as
follows:

```prolog
:- module( app, [append/3], [assertions] ).
:- entry append([1,2,3|L],L1,Cs).

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

The default options for optimization can be used to successfully specialize the pro-

gram (Figure 2 shows the default optimization menu).

![Default Optimization Menu](image-url)

**Figure 2:** Default menu options for optimization.

The following resulting partially evaluated program has specialized the third argument
by propagating the first three known values. There is an auxiliary predicate append_2 used
to concatenate the remaining elements of the first and second lists.
3.4 Nonleftmost Unfolding in Partial Evaluation of Prolog Programs:

It is well-known that non-leftmost unfolding is essential in partial evaluation in some cases for the satisfactory propagation of static information (see, e.g., [29]). Let us describe this feature by means of the following program, which implements an exponentiation procedure with accumulating parameter:

```prolog
:- module(exponential_ac, [exp/3], [assertions]).
:- entry exp(Base,3,_): int(Base).

exp(Base,Exp,Res):-
    exp_ac(Exp,Base,1,Res).
exp_ac(0,_Res,Res).
exp_ac(Exp,Base,Tmp,Res):-
    Exp > 0,
    Exp1 is Exp - 1,
    NTmp is Tmp * Base,
    exp_ac(Exp1,Base,NTmp,Res).
```

The default options for partial evaluation produce the following non-optimal residual program where only leftmost unfolding have been used:
where the calls to the builtin “is” cannot be executed and hence they have been residualized. This prevents the atoms to the right of the calls to “is” from being unfolded and intermediate rules have to be created.

In order to improve the specialization some specific options of the system must be set. We proceed by first selecting the expert mode of the optimization menu (by toggling the second option of the menu in Figure 2). An overview of the selected options is depicted in Figure 3. The computation rule no_sideffects allows us to jump over the residual builtins as long as nonleftmost unfolding is “safe” [1] – in the sense that calls to builtins are pure and hence the runtime behavior of the specialized program is preserved. We also select the option mono for abstract specialization so that a post-processing of unfolding is carried out.

The resulting specialized program is further improved:

```
:- module( _exponential_ac, [exp/3], [assertions] ).
:- entry exp(Base,3,-1) : int(Base).

exp(A,3,B) :-
    C is 1*A,
    exp_ac_1(B,C,A).

exp_ac_1(C,B,A) :-
    D is B*A,
    exp_ac_2(C,D,A).

exp_ac_2(C,B,A) :-
    C is B*A.
```
exp_ac(Exp, Base, 1, Res).

exp_ac(Exp, Base, Tmp, Res).

exp_ac(0, _, Res, Res).
exp_ac(Exp, Base, Tmp, Res) :-
   Exp > 0,
   Exp1 is Exp - 1,
   Tmp is Tmp * Base.

Figure 3: Extended menu options for nonleftmost unfolding in partial evaluation.
3.5 Integration of Abstract Interpretation and Partial Evaluation:

Abstract multiple specialization, abstract interpretation, and partial evaluation techniques are integrated into CiaoPP and their relationship is exploited in order to achieve greater levels of optimizations than those obtained by using these techniques alone.

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. Herein, we illustrate the CiaoPP's specialization framework [38] which integrates the better information propagation of abstract interpretation with the powerful program transformations performed by partial evaluation. We will use the challenge program of Figure 4.

It is a simple Ciao program which uses Peano's arithmetic. The entry declaration is used to inform that all calls to the only exported predicate (i.e., main/2) will always be of the form main(s(s(s(N))),R) with N a natural number in Peano's representation and R a variable. The predicate main/2 performs two calls to predicate formula/2. A call formula(X,W) performs mode tests ground(X) and var(W) on its input arguments and returns W = (X - 2) \times 2. Predicate two/1 returns s(s(0)), i.e., the natural number 2. A call minus(A,B,C) returns C = A - B. However, if the result becomes a negative number, C is left as a free variable. This indicates that the result is not valid. In turn, a call twice(A,B) returns B = A \times 2. Prior to computing the result, this predicate checks whether A is valid, i.e., not a variable, and simply returns a variable otherwise.

Figure 5 shows the extended option values needed in the optimization menu to produce the specialized code shown in Figure 6 using integrated abstract interpretation and partial evaluation (rules are renamed apart).

We can see that calls to predicates ground/1 and var/1 in predicate formula/2 have been removed. For this, we need to select the shfr abstract domain in the menu. The abstract information obtained from (groundness and freeness) analysis states that such calls will definitely succeed for initial queries satisfying the entry declaration (and thus, can be replaced by true). Also, the code for predicates twice/2 and tw/2 has been merged into one predicate: tw_1/2. This is also because the inferred abstract information states that the call to ground/1 in predicate twice/2 will definitely succeed (and thus can be removed). Also, the call to predicate var/1 in the first clause of predicate twice/2 will always fail.
:- module(_,[main/2],[assertions]).
:- entry main(N, R) : (gt_two_nat(N), var(R) ).
:- regtype gt_two_nat/1.
gt_two_nat(s(s(s(N))))> nat(N).

:- regtype nat/1
nat(0).
nat(s(N)) :- nat(N).

main(In,Out):-
    formula(In,Tmp),
    formula(Tmp,Out),
    nonvar(Out).

formula(X,W):-
    ground(X),
    var(W),
    two(T),
    minus(X,T,X2),
    twice(X2,W).

two(s(s(0))).

minus(X,0,X).
minus(s(Y),s(X),R):- minus(Y,X,R).
minus(0,s(X),X).

twice(X,Y):- var(X).
twice(X,Y):- ground(X), tw(X,Y).

tw(0,0).
tw(s(X),s(s(NX))):- tw(X,NX).

Figure 4: A simple Peano’s arithmetic program.
Figure 5: Extended menu options for integration of abstract interpretation and partial evaluation.
:- module( _example_sd, [main/2], [assertions , regtypes , nativeprops] ).
:- entry main(N,R): ( gt_two_nat(N), var(R) ).

main(s(s(s(B))),A) :-
  tw_1(B,C),
  formula_1(A,C).

tw_1(0,0).

tw_1(s(A),s(s(B))) :-
  tw_1(A,B).

formula_1(0,0).

formula_1(s(s(B)),s(A)) :-
  tw_1(A,B).

Figure 6: Optimized Peano's arithmetic program with abstract interpretation and partial evaluation integrated.

(and thus, this clause can be removed). These optimizations can be selected in CiaoPP by choosing the option value spec for Select Optimize and the option value all for Abstract Spec Definitions in the menu (See Figure 5). These points illustrate hence the benefits of exploiting abstract information in order to abstractly execute certain atoms which may, in turn, allow unfolding of other atoms.

However, the use of an abstract domain which captures groundness and freeness information will in general not be sufficient to determine that, in the second execution of formula/2 in predicate main/2, the tests ground(X) and var(W) will also succeed. The reason is that, on success of minus(T,X,X2), X2 cannot be guaranteed to be ground since minus/3 succeeds with a free variable in its third argument position. It can be observed, however, that for all calls to minus/3 in the executions described by the entry declaration, the third clause for minus/3 is useless. It will never contribute to a success of minus/3 since such predicate is always called with a value greater than zero in its first argument. Unfolding can make this explicit by fully unfolding calls to minus/3 since they are sufficiently instantiated, and as a result, the “dangerous” third clause is disregarded. This unfolding allows concluding that in our particular context all calls to minus/3 succeed with a ground third argument. This can be selected in CiaoPP by choosing the values for local and global control within the optimization menu shown in Figure 5. This illustrates the importance of performing unfolding steps in order to prune away useless branches, and that
3.6 Parallelization:

An example of a non-trivial program optimization performed using abstract interpretation in CiaoPP is program parallelization [5]. 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 [35], exploit parallelism under certain independence conditions, which allow guaranteeing interesting correctness and no-slowdown properties for the parallelized programs [26, 14]. This process is complicated by the presence of shared variables and pointers among data structures at run-time.

Consider the program of Figure 7 (the module and entry directives will be explained later).

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

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

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

Figure 7: A qsort program.
A possible parallelization obtained with the selected options in the menu depicted in Figure 8 is:

\[
\text{qsort}([X|L], R) :- \\
\text{partition}(L, X, L1, L2), \\
( \text{indep}([[L1,L2]]) \rightarrow \text{qsort}(L2, R2) \& \text{qsort}(L1, R1) \\
; \text{qsort}(L2, R2), \text{qsort}(L1, R1) ), \\
\text{append}(R1, [X|R2], R).
\]

which indicates that, provided that \(L1\) and \(L2\) do not have variables in common (at execution time), then the recursive calls to \text{qsort} can be run in parallel.

Figure 8: Menu options for parallelization with no analysis information.

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 \text{partition} (and therefore do not share variables), the independence test and the conditional can be simplified via abstract executability and the annotator yields instead:
which is much more efficient since it has no run-time test. This test simplification process is described in detail in [5] where the impact of abstract interpretation in the effectiveness of the resulting parallel expressions is also studied. The selected menu options needed to produce this output are depicted in Figure 9.

Figure 9: Menu options for parallelization with analysis information.

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 [9], constraint-based independence [14], 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.¹

### 3.7 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 [33] 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 threshold² 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 7. We use CiaoPP to perform a transformation for granularity control. An overview of the selected menu options to achieve this is depicted in Figure 10.

In the resulting optimized code, 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

---

¹For example, stream and-parallelism can be seen as independent and-parallelism if the independence of “bindings” rather than goals is considered.

²This threshold can be determined experimentally for each parallel system, by taking the average value resulting from several runs.
Figure 10: Menu options for parallelization with granularity control.
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.\(^3\) Predicate partition_o3_4/6:

\[
\begin{align*}
\text{partition_o3_4}([],_B, [], [], 0, 0). \\
\text{partition_o3_4}([E|R], C, [E|\text{Left1}], \text{Right}, _1, _2) :- \\
\quad E < C, \text{partition_o3_4}(R, C, \text{Left1}, \text{Right}, _3, _2), _1 \text{ is } _3 + 1. \\
\text{partition_o3_4}([E|R], C, \text{Left}, [E|\text{Right1}], _1, _2) :- \\
\quad E \geq C, \text{partition_o3_4}(R, C, \text{Left}, \text{Right1}, _1, _3), _2 \text{ is } _3 + 1.
\end{align*}
\]

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) [32].

## 4 Program Debugging and Assertion Validation

\(\text{CiaoPP}\) 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 [24, 39] which involves several of the tools which comprise \(\text{CiaoPP}\). Figure 11 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 inferring properties of the program via abstract interpretation-based static analysis and comparing this information against (partial) specifications written in terms of assertions (see [27] for a detailed description of the sufficient conditions used for achieving this \(\text{CiaoPP}\) functionality).

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 Assertions and Properties:

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`. 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 Figure 12, we could describe the mode of the (now missing) `geq` and `lt` predicates to the analyzer for example as follows:

```prolog
:- 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 later in this section. Interestingly, this very general concept of assertions is also particularly useful for generating documentation automatically (see [21] 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 12 when information is requested at this level:

---

3This size threshold will obviously be different if the cost function is.
:- 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|Leftl],Right):-
    lt(E,C), partition(R,C,Leftl,Right).
partition([E|R],C,Left,[E|Rightl]) :-
    geq(E,C), partition(R,C,Left,Right1).

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

Figure 12: A modular qsort program.

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:

```prolog
:- 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 [46, 11, 20, 18, 45], 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.:

```prolog
:- use.module(library(lists), [list/1]).
```

The idea of using analysis information for debugging comes naturally after observing analysis outputs for erroneous programs. Consider the program in Figure 13.

The result of regular type analysis for this program includes the following code:

```prolog
:- true pred qsort(A,B)  
   : ( term(A), term(B) )  
   => ( list(A,tll3), list(B,~x) ).
:- regtype tll3/l.
tll3(A) :- arithexpression(A).
tll3([]).
tll3([A|B]) :- arithexpression(A), list(B,tll3).
tll3(e).
```

where arithexpression is a library property which describes arithmetic expressions and list(B, ~x) means “a list of x’s.” A new name (tll3) 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, the preprocessor can detect inconsistencies in the program and check the assertions present in other modules used by the program.

For example, we can turn on compile-time error checking and selecting type and mode analysis for our tentative qsort program in Figure 13, by selecting the action `check.assertions`
as shown in Figure 14. By default, the option Perform Compile-Time Checks is set to auto, which means that the system will automatically detect the analyses to be performed in order to check the program, depending on the information available in the program assertions (in the example in Figure 13, the entry assertion informs how the predicate qsort/2 will be called using types and modes information only). Using the default options, and setting Report Non-Verified Assrts to error, we obtain the following messages (and the system highlights the line which produces the first of them, as shown in Figure 15):

WARNING (preproc_errors): (Ins 3-7) goal partition(L,L1,X,L2) at literal 1 does not succeed!
WARNING (ctchecks_messages): (Ins 11-12) the head of clause `partition/4/2` is incompatible with its call type
  Head: partition([e|R],C,[E|Left1],Right)
  Call Type: partition(list(num),term,num,term)
ERROR (ctchecks_messages): (Ins 13-14) at literal 1 false calls assertion:
  :- calls >=(A,B) : [[ground(A),ground(B)]]
because on call of >=(A,B) : mshare([[B],[A]]),var(B)
WARNING (preproc_errors): (Ins 13-14) goal >=(E,C) at literal 1
Figure 14: Static compile-time checking of assertions in system libraries.

does not succeed!

where the first message refers to the lines of the first clause of `qsort/2`, the second one to the second clause of `partition/4`, and the last two messages correspond to the third clause `partition/4`.

First and last messages warn that all calls to `partition` and `>=/2` will fail, something normally not intended (e.g., in our case). The error 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 the second argument to the call to `>=/2` is a variable, with the assertion:

```
:- check calls A>=B : (ground(A), ground(B)).
```

which is present in the default builtins module, and which implies that the two arguments to `>=/2` should be ground when this arithmetic predicate is called. The message signals a compile-time, or abstract, incorrectness symptom [7], 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
module(_, [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([],[],[],[]).
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).

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 continues producing the following message:

WARNING: Clause 'partition/4/2' is incompatible with its call type
Head: qsort2:partition([e|R],C,[E|Left1],Right)
Call Type: qsort2:partition(basic_props:list(num),term,num,term)

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:

\[
\begin{align*}
\text{:- calls qsort}(A, B) & : \text{list}(A, \text{num}). & \text{A1} \\
\text{:- success qsort}(A, B) & \Rightarrow (\text{ground}(B), \text{sorted_num_list}(B)). & \text{A2} \\
\text{:- calls partition}(A, B, C, D) & : (\text{ground}(A), \text{ground}(B)). & \text{A3} \\
\text{:- success partition}(A, B, C, D) & \Rightarrow (\text{list}(C, \text{num}), \text{ground}(D)). & \text{A4} \\
\text{:- calls append}(A, B, C) & : (\text{list}(A, \text{num}), \text{list}(B, \text{num})). & \text{A5}
\end{align*}
\]

\footnote{The check prefix is assumed when no prefix is given, as in the example shown.}
:- 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 restricted to a subset of the calls using “:”. More details on the assertion language can be found in [40].

CiaoPP can perform compile-time checking of the assertions above, by comparing them with the assertions inferred by analysis (see [27, 7, 41] for details), producing as output the following assertions (refer also to Figure 11, output of the comparator):

:- checked calls qsort(A,B) : list(A,num).
:- checked success qsort(A,B) => sorted_num_list(B).
:- checked calls partition(A,B,C,D) : (ground(A),ground(B)).
:- checked success partition(A,B,C,D) => (list(C,num),ground(D)).
:- false calls append(A,B,C) : (list(A,num), list(B,num)).
:- checked comp partition/4 + not_fails.
:- checked comp partition/4 + is_det.
:- checked comp partition/4 + terminates.

In order to produce this output, the CiaoPP check-assertions menu must be set to the same options as those used in Figure 14 for checking assertions in system libraries. Since the auto mode has been used for the option Perform Compile-Time Checks, CiaoPP has automatically detected that the program must be analyzed not only for types and modes domains, but also to check non-failure, determinism, and upper-bound cost. 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 [2]. 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 | Rl], R) in qsort (x instead of X). Assertions A1, A3, A4, A6, A7, and A8 have been detected to hold. Note that though the predicate partition may fail in general, in the context of the current program it can be proved not to fail (assertion A6). However, it was not possible to prove statically assertion A2, which has remained with check status. 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.5

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 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 11). 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):

5Note 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.
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 \texttt{qsort}, since \([2,1]\) is not the result of ordering \([1,2]\) in ascending order. This is a (now, run-time, or \textit{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 \(R_1\) and \(R_2\) have been swapped) is the cause of the error and that the right definition of predicate \texttt{qsort} is the one in Figure 7.

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 naïve reverse program in Figure 17.

Suppose that the programmer thinks that the cost of \texttt{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 \texttt{nrev} to be quadratic. We will show that CiaoPP can be used to inform the programmer about this false idea about the cost of \texttt{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, nativeprops]).
:- entry nrev(A,B) : (ground(A), list(A), var(B)).

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

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

Figure 17: The naive reverse program.

In order to check cost assertions, we have to set specific analysis options to CiaoPP. The way to do it is to set Customize Analysis Flags to on when Select Action Group is set to check assertions in the menu (see Figure 18).
The extended menu with the appropriate options is shown in Figure 19.

With these options, 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 $0.5 \times (\text{length}(A))^2 + 1.5 \times \text{length}(A) + 1$ 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, it holds when 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.

CiaoPP can also verify or falsify cost assertions expressing worst case computational complexity orders (this is specially useful if the programmer does not want or does not know which particular cost function should be checked). For example, suppose now that the programmer adds the following “check” assertion:

  :- check comp nrev(A,B) + steps_o(length(A)).
module(reverse, [nrev], [assertions]).
:= use_module(library('assertions/native_props')).
:= entry nrev(A,B) : (ground(A), list(A, term), var(B)).
:= check comp nrev(A,B) + steps_ub(length(A)+1).
nrev([],[]).
nrev([H|L],R) :-
nrev(L,R1),
append(R1,[H],R).

Figure 18: Access to analysis flags from the \textit{check\_assertions} menu.
module(reverse, [nrev/2], [assertions]).
:- use_module(library('assertions/native_props')).
:- entry nrev(A,B) :- (ground(A), list(A, term), var(B)).
:- check comp nrev(A,B) + steps_ub(length(A)+1).
nrev([],[]).
nrev([H|L],R) :-
    nrev(L,R1),
    append(R1,[H],R).

Figure 19: Extra analysis options for performance debugging.
In this case, we get the following error message:

ERROR: false comp assertion:

```
:- comp nrev(A,B) : true => steps_o(length(A))
```

because in the computation the following holds:

```
steps_lb(0.5*exp(length(A),2)+1.5*length(A)+1)
```

This message states that \( n\text{rev} \) will take at least \( 0.5 \cdot (\text{length}(A))^2 + 1.5 \cdot \text{length}(A) + 1 \) resolution steps (which is the cost analysis output, as in the previous example), while the assertion requires that the worst case cost of \( n\text{rev} \) be linear on \( \text{length}(A) \) (the size of the input argument).

If the programmer adds now the following “check” assertion:

```
:- check comp nrev(A,B) + steps_o(exp(length(A),2)).
```

which states that the worst case cost of \( n\text{rev} \) is quadratic, i.e. is in \( O(n^2) \), where \( n \) is the length of the first list (represented as \( \text{length}(A) \)). Then the assertion is validated and the following “checked” assertion is included in the output produced by CiaoPP:

```
:- checked comp nrev(A,_,l) + steps_o( exp(length(A), 2) ).
```

Thanks to this functionality, CiaoPP can certify programs with resource consumption assurances and also efficiently check such certificates [22].

### 4.6 Abstraction-Carrying Code:

CiaoPP also allows to generate program certificates based on abstract interpretation, in order to provide the basis for abstraction-carrying code.

Let us consider again a program for the naive reversal of a list, in this case using functional notation, part of the Ciao system. We have added a set of assertions which specify the intended safety policy. The idea is that, if the assertions can be verified, then we know that the safety policy is entailed from them and the program. The program code is as follows:

```
:- module(_, [nrev/2], [assertions,functions,regtypes,nativeprops]).
:- function(arith(false)).
:- entry nrev/2 : {list, ground} * var. 

:- check pred nrev(A,B) : list(A) => list(B).
:- check comp nrev(_,_) + ( not_fails, is_det ).
:- check comp nrev(A,_) + steps_o( exp(length(A),2) ).
```
nrev([ ] ) := [ ] .
nrev([H|L] ) := ~conc( ~nrev(L), [H] ) .

:- check comp conc(_,_,_ ) + ( terminates, is_det ) .
:- check comp conc(A,_,_ ) + steps_o( length(A) ) .

conc( [ ] , L ) := L .
conc([H|L], K) := [ H | ~conc(L,K) ] .

For generating the certificate, the menu check-assertions will be used. Since the certificate will require specific values for some advanced options, the expert menu must be used, selecting expert for the Select Menu Level option. The complete set of values are
module(_, [nrev/2], [assertions, functions, regtypes, nativeprops]).
:- function(arith(false)).
extent nrev/2 : {list, ground} * var.

:- check pred nrev(A,B) : list(A) => list(B).
:- check comp nrev(_,_) + ( not_fails, is_det ).
:- check comp nrev(A,_) + steps_o( exp(length(A),2) ).

:- checked comp nrev(_1,_2)
   + ( not_fails, is_det ).

:- checked comp nrev(A,_1)
   + steps_o( exp(length(A),2) ).

:- checked calls nrev(A,B)
   : list(A).

:- checked success nrev(A,B)
   : list(A)
   => list(B).

:- checked comp conc(_1,_2,_3)
   + ( terminates, is_det ).
The consumer will receive the untrusted code and the certificate package generated with the options in Figure 20. It proceeds to check that the certificate is valid for the program and that the safety policy is entailed from it. To do this, we select the option check_certificate from the Action Group, as shown in Figure 21. It can be seen in Figure 22 that with this option CiaoPreprocessor successfully validates the certificate and assertions. Hence, the program can be trusted by this consumer.

5 Static Analysis and Program Assertions

The fundamental functionality behind CiaoPreprocessor is static global program analysis, based on abstract interpretation. For this task CiaoPreprocessor uses the PLAI abstract interpreter [37, 5], including extensions for, e.g., incrementality [25, 42], modularity [4, 44, 6], analysis of constraints [13], and analysis of concurrency [34].

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: essen-
tially, 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, metaprogramming, higher-order, etc.) and as a result can for example deal safely with arbitrary ISO-Prolog programs [4]. A unified language of assertions [4, 40] is used to express the results of analysis, to provide input to the analyzer, and, as we have seen in Section 4, to provide program specifications for debugging and validation, as well as the results of the comparisons performed against the specifications.

5.1 Module-aware 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 12, 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” [36]) of the module using CiaoPP yields the following results:

\[
\begin{align*}
\text{- true pred qsort}(A,B) & : \text{mshare}([[A],[A,B],[B]]) \\
& \implies \text{mshare}([[A,B]]). \\
\text{- true pred partition}(A,B,C,D) & : (\text{var}(C), \text{var}(D), \text{mshare}([[A],[A,B],[B],[C],[D]])) \\
& \implies (\text{ground}(A), \text{ground}(C), \text{ground}(D), \text{mshare}([[B]])). \\
\text{- true pred append}(A,B,C) & : (\text{ground}(A), \text{mshare}([[B],[B,C],[C]])) \\
& \implies (\text{ground}(A), \text{mshare}([[B,C]])).
\end{align*}
\]

\[\text{6}\]In the “sharing+freeness” domain \text{var} denotes variables that do not point yet to any data structure, \text{mshare} denotes pointer sharing patterns between variables. Derived properties \text{ground} and \text{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.
These assertions express, for example, that the third and fourth arguments of \texttt{partition} have “output mode”: when \texttt{partition} is called (:) they are free unaliased variables and they are ground on success (=>). Also, \texttt{append} is used in a mode in which the first argument is input (i.e., ground on call). Also, upon success the arguments of \texttt{qsort} will share all variables (if any).

5.2 Type Analysis:

CiaoPP can infer (parametric) types for programs both at the predicate level and at the literal level [20, 18, 45]. The output for Figure 12 at the predicate level, assuming that we have imported the \texttt{lists} library, is:

\begin{verbatim}
:- true pred qsort(A,B) 
   : ( term(A), term(B) )
\end{verbatim}
Figure 24: Type analysis without entry assertions.

\[
\Rightarrow (\text{list}(A), \text{list}(B)).
\]

\[- \text{true pred partition}(A, B, C, D) \]
\[\begin{align*}
&: (\text{term}(A), \text{term}(B), \text{term}(C), \text{term}(D)) \\
&\Rightarrow (\text{list}(A), \text{term}(B), \text{list}(C), \text{list}(D)).
\end{align*}
\]

\[- \text{true pred append}(A, B, C) \]
\[\begin{align*}
&: (\text{list}(A), \text{listl}(B, \text{term}), \text{term}(C)) \\
&\Rightarrow (\text{list}(A), \text{listl}(B, \text{term}), \text{listl}(C, \text{term})).
\end{align*}
\]

where \text{term} is any term and \text{prop listl} is defined in library(lists) as:

\[- \text{regtype listl}(L, T) \# "@var{L} is a list of at least one @var{T}'s." \]
\[
\begin{align*}
\text{listl}([X|R], T) &\colon T(X), \text{list}(R, T). \\
\text{listl}([]_T) &\colon \text{on}.
\end{align*}
\]

\[- \text{regtype list}(L, T) \# "@var{L} is a list of @var{T}'s." \]
\[
\begin{align*}
\text{list}([],_T). \\
\text{list}([X|L], T) &\colon T(X), \text{list}(L).
\end{align*}
\]
We can use entry assertions [4] to specify a restricted class of calls to the module entry points as acceptable:

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

This informs the analyzer that in all external calls to \texttt{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 \texttt{var}, \texttt{num}, \texttt{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:

```prolog
:- true pred qsort(A,B) :
    : ( list(A,num), term(B) )
=> ( list(A,num), list(B,num) ).
```
5.3 Non-failure and Determinacy Analysis:

CiaoPP includes a non-failure analysis, based on [15] 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 [31],

```prolog
:- 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) ).
```
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 12 includes the following assertion:

```prolog
:- 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.) This result has been obtained using the analyze menu options depicted in Figure 26.
5.4 Size, Cost, and Termination Analysis:

CiaoPP can also infer lower and upper bounds on the sizes of terms and the computational cost of predicates [16, 17]. 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:
Note that in this example the size measure used is list length. The sentence
\( \text{size}_{ub}(C, \text{length}(B) + \text{length}(A)) \) 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 options that must be set in the analyze menu of CiaoPP are shown in Figure 27.

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),
    \text{size}_{lb}(A, \text{length}(A)), \text{size}_{lb}(B, \text{length}(B)),
    \text{size}_{lb}(C, \text{length}(B) + \text{length}(A)) )
+ ( \text{notfails, covered, steps}_{lb}(\text{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. The menu options are the same as in Figure 27, but selecting steps_{lb} for the cost analysis option.

### 5.5 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 \textit{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 \textit{incorrect}: the properties stated in inferred assertions do always hold of the program.

\textbf{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


[31] P. López-Garcia, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th Inter-


