Program Analysis, Debugging, and Optimization Using the Ciao System Preprocessor[1]

Manuel V. Hermenegildo
Francisco Bueno
Germán Puebla
Pedro López
{herme,bueno,german,pedro}@fi.upm.es
School of Computer Science, T.U. Madrid (UPM)

Abstract

We present a tutorial overview of Ciaopp, the Ciao system preprocessor. Ciao is a public-domain, next-generation logic programming system, which subsumes ISO-Prolog and is specifically designed to a) be highly extensible via libraries and b) support modular program analysis, debugging, and optimization. The latter tasks are performed in an integrated fashion by Ciaopp. Ciaopp uses modular, incremental abstract interpretation to infer properties of program predicates and literals, including types, variable instantiation properties (including modes), non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc. Using such analysis information, Ciaopp can find errors at compile-time in programs and/or perform partial verification. Ciaopp checks how programs call system libraries and also any assertions present in the program or in other modules used by the program. These assertions are also used to generate documentation automatically. Ciaopp also uses analysis information to perform program transformations and optimizations such as multiple abstract specialization, parallelization (including granularity control), and optimization of run-time tests for properties which cannot be checked completely at compile-time. We illustrate "hands-on" the use of Ciaopp in all these tasks. By design, Ciaopp is a generic tool, which can be easily tailored to perform these and other tasks for different LP and CLP dialects.

Keywords: Global Analysis, Debugging, Verification, Parallelization, Optimization, Partial Evaluation, Multiple Specialization, Abstract Interpretation.

[1] We would like to thank the members of the ICLP'99 Program Committee for their kind invitation to present this tutorial. The CIAO system is a collaborative effort of members of several institutions, including UPM, U.Melbourne, Monash U., U.Arizona, Linköping U., NMSU, K.U.Leuven, Bristol U., and Ben-Gurion U. The system documentation and related publications contain more specific credits. The development of Ciaopp has been funded in part by ESPRIT project "DiSCiPl" and CICYT project "ELLA".

1 The Ciao Program Development System

Ciao [14] is a public-domain,[2] next-generation logic programming environment. It is intended at the same time as a robust public-domain ISO-Prolog implementation supporting programming in the large and in the small, and as an experimentation workbench for new logic programming technology. The Ciao environment includes an enhanced version of the interactive shell found in most Prolog systems, a standalone compiler, a powerful preprocessor/debugger, a script interpreter, an automatic documentation generator, a rich interface to the emacs editor, and some program visualization tools.

The Ciao system has been specifically designed to be highly extensible and to support modular program analysis, debugging, and optimization. The language includes a simple kernel with a robust module system, on top of which extensions are added via libraries. These libraries are generally normal Ciao modules which provide run-time support predicates (including attributed variable handling code) and compile-time support such as operator declarations and macro expansions. The latter are all local to the modules which import the library. The Ciao libraries currently support the full ISO-Prolog standard, several constraint domains, functional and higher order programming, concurrent and distributed programming, Internet programming, objects, persistence, database access, rich interfaces to other languages (such as C, tcl/tk, and Java), etc.

The Ciao compilation process is conceptually divided into two levels. The *low-level compiler*, Ciaoc, itself a Ciao application, is in charge of producing object code for each module, and linking the object code into executables. This compilation is performed automatically and incrementally, in the sense that only necessary modules whose source code has changed are recompiled when a module is used. Ciaoc generates executables which are small and of performance which is competitive with state-of-the-art bytecoded systems.[3] At a higher level, the *preprocessor*, Ciaopp, performs modular, incremental global program analysis based on abstract interpretation [4] to infer information on the program. This information is applied in a novel way to aid the program development and debugging process, as well as in the more traditional areas of program transformation and optimization. By design, Ciaopp is a generic tool, which can be easily tailored to perform these and other tasks for different LP and CLP dialects.

In the following, we present an overview of Ciaopp at work. Our aim is to present not the techniques used by Ciaopp, but instead the main functionalities of the system in a tutorial way, by means of examples. However, we do provide references where the interested reader can find more details on the actual techniques used.[4] Section 2 presents Ciaopp at work performing

---

[2] The ciao system is available from `http://www.clip.dia.fi.upm.es`.

[3] In addition, the *script processor*, allows executing *scripts* written in Prolog [13].

[4] Space limitations 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

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

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

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

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

Figure 1: A modular qsort program.

program analysis, while Section 3 does the same for program debugging and validation and Section 4 for program transformation and optimization.

2 Static Analysis and Program Assertions

The fundamental functionality behind Ciaopp is static program analysis. For this task Ciaopp uses the PLAI abstract interpreter [18, 2], its CLP [11] and incremental [15, 22] versions, and adaptations of Gallagher's regular type analysis [8]. The system includes several abstract domains and can infer information on basic properties such as moded types, definiteness, freeness, and grounding dependencies, as well as on more complex properties such as determinacy, non-failure, bounds on term sizes, and bounds on computational 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 [1]. A unified language of assertions [1, 19] 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.

Modular Static Analysis Basics: Ciaopp takes advantage of modular program structure to perform more precise and efficient, incremental analysis [1]. Consider the program in Figure 1, defining a module which exports the `qsort` predicate and imports predicates `geq` and `lt` from module `compare`. During the analysis of this program, Ciaopp will take advantage of the fact that the only predicate that can be called from outside is the exported predicate `qsort`. This allows Ciaopp to infer more precise information than if it had to consider that all predicates may be called in any possible way (as would be true had this been a simple "user" file instead of a

most directly related to Ciaopp, which are typically our own work. We ask the reader to kindly forgive this. The publications referenced do contain comprehensive references to related work.

module). Also, assume that the `compare` module has already been analyzed. This allows Ciaopp to be more efficient, since it will use the information obtained for `geq` and `lt` during analysis of `compare` instead of reanalyzing them. Assuming that `geq` and `lt` have a similar binding behavior as the standard comparison predicates, a mode and independence analysis ("sharing+freeness") of the module using Ciaopp yields the following results:

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

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).

Assertions and Properties: The above output is given in the form of *assertions*. 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 [1]. For example, if we commented out the `use_module/2` declaration in Figure 1, we could describe the mode of the (now missing) `geq` and `lt` predicates to the analyzer for example as follows:

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

Finally, assertions with a `check` prefix can be used to specify the *intended* semantics of the program, which can then be used in debugging and/or validation [19, 20], as we will see in Section 3. The same assertions are also used to generate documentation automatically [17].

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. The following is a fragment of the output produced by Ciaopp for the program in Figure 1 when information is requested at this level:

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

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

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

defines the property "list". A list is an instance of a very useful class of user-defined properties called regular types, which is 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]).
```

Type Analysis: Ciaopp can infer (parametric) types for programs both at the predicate level and at the literal level. The output for Figure 1 at the predicate level, assuming that we have imported the `lists` library, is:

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

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

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

We can use `entry` assertions [1] (essentially, "`trust calls`" assertions) to specify a restricted class of calls to the module entry points as acceptable:

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

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

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

Non-failure and Determinacy Analysis: Ciaopp includes a non-failure analysis, based on [6], 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 which can detect predicates which produce at most one solution, or predicates whose clause tests are disjoint, even if they are not fully deterministic (because they call other predicates which are nondeterministic). For example, the result of these analyses for Figure 1 includes the following assertion:

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

(The `+` field in `pred` assertions can contain a conjunction of computational properties which are global to the predicate.)

Size, Cost, and Termination Analysis: Ciaopp can also infer lower and upper bounds on the sizes of terms and the computational cost of predicates [5, 7]. 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),
             upper_size(A,length(A)), upper_size(B,length(B)),
             upper_size(C,length(B)+length(A)) )
         + upper_time(length(A)+1).
```

Note that in this example the size measure used is list length. The assertion `upper_size(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),
             lower_size(A,length(A)), lower_size(B,length(B)),
             lower_size(C,length(B)+length(A)) )
         + ( not_fail, covered, lower_time(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.

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*.

```
:- module(qsort, [qsort/2], [assertions]).

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

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

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

Figure 2: A tentative qsort program.

3 Program Debugging and Assertion Validation

Within Ciaopp, global analysis is not only used to infer program properties, but also to detect classes of errors at compile-time which go well beyond the usual syntactic checks. Errors can be detected in conventional programs or, alternatively, assertions can be added to such programs stating intended program properties, and which can then be validated or falsified, in the latter case detecting an error.

Static Debugging: The idea of using analysis information for debugging comes naturally after observing analysis outputs for erroneous programs. Consider the program in Figure 2. 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.

Ciaopp includes a number of facilities to help in the debugging task beyond manual inspection of the analyzer output. For example, Ciaopp can find incompatibilities between the ways in which program predicates are called and their definitions. It can also detect incompatibilities between the way library predicates are called and their intended mode of use, expressed in the form of assertions in the libraries themselves. In order to use these capabilities, we add to the program the same declaration of its intended use of previous examples:

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

Turning on compile-time error checking and selecting type and mode analysis produces 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`.

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 [3], indicating that the program does not satisfy the specification given (that of the builtin predicates, in this case). Checking the indicated call to `partition` and inspecting its arguments we detect that in the definition of `qsort`, `partition` is called with the second and third arguments in reversed order – the correct call is `partition(L,X,L1,L2)`.

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

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

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.

Validation of User Assertions: 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

:- 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, the

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

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 (see [19]).

Ciaopp can *check* the assertions above, by comparing them with the assertions inferred by analysis (see [3, 20] for details), producing:

```
:- 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
```

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`). From the rest of the output we can conclude that the rest of the specification has been partially validated: assertions `A1`, `A3`, and `A4` have been detected to hold, but it was not possible to prove statically assertion `A2`, which has remained with `check` status. Ciaopp can, on request, introduce run-time tests in the program which will call the definition of `sorted_num_list` at the appropriate times. Note 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 and modes) 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]

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. 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)).
```

[6] Not 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.

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

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

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

L = [2,1] ?
```

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

4 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), and multiple program specialization. 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.

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 of 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(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`. The notion of *abstract executability* [23, 12] can reduce some literals to `true`, `fail`, or a set of primitives (typically, unifications) based on the information available from abstract interpretation.

Ciaopp can also apply abstract specialization in the optimization of programs with dynamic scheduling (e.g., using `delay` declarations) [21]. 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 [14].

Parallelization: Another application of global analysis in Ciaopp is in automatic program parallelization [2]. It is also performed as a source-to-source transformation, in which the input program is *annotated* with parallel expressions. A number of heuristic parallelization algorithms are available, which guarantee certain no-slowdown properties [16] for the parallelized programs. We consider again the program of Figure 1. A possible parallelization (obtained in this case with the "MEL" annotator) is:

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

which indicates that, provided that `L1` and `L2` do not have variables in common (at execution time), then the recursive calls can be run in parallel. Given the information inferred by, e.g., the mode and independence analysis (see Section 2), where `L1` and `L2` are ground after `partition` (and therefore do not share variables) 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.

Granularity Control: Another application of the information produced by the Ciaopp analyzers, in this case the cost analysis, is to perform run–time task granularity control [10] 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[7] to decide for parallel or sequential execution. Optimizations to this general process include cost function simplification and improved term size computation, both of which are illustrated in the following example.

Consider again the qsort program in Figure 1. We use Ciaopp to perform a transformation for granularity control, using the analysis information of type, sharing+freeness, and upper bound cost analysis, and taking as input the parallelized code obtained in the previous section. Ciaopp adds a clause "`qsort(_1,_2) :- g_qsort(_1,_2).`" (to preserve the original entry point) and produces `g_qsort/2`, the version of `qsort/2` that performs granularity control (`s_qsort/2` is the sequential version):

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

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

Note that if the lengths of the two input lists to the qsort program are greater than a threshold (a list length of 7 in this case) then versions which continue performing granularity control are executed in parallel. Otherwise, the two recursive calls are executed sequentially. The executed version of each of such calls depends on its grain size: if the length of its input list is not 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.[8] 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) [9].

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 [24] 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

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

the analyzer (by means of an `entry` declaration) with accurate information on the expected modes of use of the program. However, in the interesting case in which the user does not provide such declaration, the code generated contains a large number of run-time tests. We include below the code for predicate `multiply` which multiplies a matrix by a vector:

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

Four independence tests and one groundness test have to be executed prior to executing in parallel the calls in the body of the recursive clause of `multiply`. 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 [24]. 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.

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 [25] we present a specialization framework which integrates the better information propagation of abstract interpretation with the powerful program transformations performed by partial evaluation.

References

[1] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[2] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[3] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linkoping, Sweden, May 1997. U. of Linkoping Press. Available from `ftp://clip.dia.fi.upm.es/pub/papers/aadebug_discipldeliv.ps.gz`.

[4] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[5] S. K. Debray, P. López García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[6] S. K. Debray, P. López García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press.

[7] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[8] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[9] P. López García and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Pres.

[10] P. López García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *J. of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

[11] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Trans. on Programming Languages and Systems*, 18(5):564–615, 1996.

[12] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[13] M. Hermenegildo. Writing "Shell Scripts" in SICStus Prolog, April 1996. Available from `http://www.clip.dia.fi.upm.es/`. Posting in `comp.lang.prolog`.

[14] M. Hermenegildo, F. Bueno, D. Cabeza, M. García de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*. Nova Science, Commack, NY, USA, April 1999.

[15] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

[16] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[17] M. Hermenegildo and The CLIP Group. An Automatic Documentation Generator for (C)LP – Reference Manual. The CIAO System Documentation Series – TR CLIP5/97.1, Facultad de Informática, UPM, August 1997.

[18] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[19] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *ILPS'97 WS on Tools and Environments for (C)LP*, October 1997. `ftp://clip.dia.fi.upm.es/pub/papers-/assert_lang_tr_discipldeliv.ps.gz`.

[20] G. Puebla, F. Bueno, and M. Hermenegildo. A Framework for Assertion-based Debugging in Constraint Logic Programming. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, Venezia, Italy, September 1999.

[21] G. Puebla, M. García de la Banda, K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on Logic Programming*, pages 93–107, Cambridge, MA, June 1997. MIT Press.

[22] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[23] G. Puebla and M. Hermenegildo. Abstract Specialization and its Application to Program Parallelization. In J. Gallagher, editor, *VI International Workshop on Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.

[24] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999. In press.

[25] G. Puebla, M. Hermenegildo, and J. Gallagher. An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework. In O Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, number NS-99-1 in BRISC Series, pages 75–85. University of Aarhus, Denmark, January 1999.