Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs

G. Puebla* M. Hermenegildo*

Abstract

Separating programs into modules is a well-known technique which has proven very useful in program development and maintenance. Starting by introducing a number of possible scenarios, in this paper we study different issues which appear when developing analysis and specialization techniques for modular logic programming. We discuss a number of design alternatives and their consequences for the different scenarios considered and describe where applicable the decisions made in the Ciao system analyzer and specializer. In our discussion we use the module system of Ciao Prolog. This is both for concreteness and because Ciao Prolog is a second-generation Prolog system which has been designed with global analysis and specialization in mind, and which has a *strict* module system. The aim of this work is not to provide a theoretical basis on modular analysis and specialization, but rather to discuss some interesting practical issues.

1 Introduction

Writing *modular* programs, i.e., programs which are made of components called *modules*, has proven useful in practice for both program development and maintenance.[1] Program compilation, analysis, and specialization have in common that they receive programs as input and they have to handle them in some way or another. Performing these tasks on modular programs differs from doing so on non-modular programs in several interesting ways. Our purpose is to study a number of issues which appear when developing analysis and specialization techniques for modular logic programming.

By *strict* module systems we refer to those in which a module can only communicate with other modules via its *interface*. The interface of a module usually contains the names of the *exported* predicates and the names of the *imported* modules. Other modules can only use predicates which are among the ones exported by the considered module. Predicates which are not exported are not visible outside the module.

For concreteness, we will concentrate on a particular, strict module system for Prolog [CH99a]: the one used in Ciao Prolog [CLI97]. This module system is in fact quite similar to the module systems of the most popular Prolog implementations. Thus, the discussion in the rest of the paper should apply to such module systems, or, at least, to their subset which is strict.[2] However, it is useful in our discussion that some of the particular choices in the design of the Ciao module system were made keeping the task of global analysis in mind.

This paper builds primarily on [BCHP96], in which many techniques were proposed for dealing with "difficult" features of practical languages (in particular, full ISO Prolog) in the context of analysis. Herein we concentrate on the issue of modular analysis, which was only sketched at the end of [BCHP96]. We also extend the techniques to another application: specialization.

1.1 An Example of a Modular Program

Figure 1 shows the code of a module which implements the well-known quicksort algorithm. The declaration `:- module(qsort,[qsort/2]).` states that the module name is `qsort` and that it

*Department of Computer Science, Technical University of Madrid (UPM), Spain. {herme,german}@fi.upm.es

[1] Modularity is also one of the fundamental principles behind object oriented programming.

[2] However, as already argued in [BCHP96], we feel that a strict module system is beneficial not only for global program analysis and specialization, but also in the more traditional activities of program development and maintenance.

```
:- module(qsort, [qsort/2]).
:- use_module(lists, [append/3]).

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

Figure 1: A module for quicksort

exports the predicate `qsort/2`. The declaration `:- use_module(lists,[append/3]).` indicates that the module `qsort` imports the predicate `append/3` from module `lists`, which is shown in Figure 2. Last, the program has a third module, `tests`, which is the main one. It imports `qsort` and checks that the results produced are sorted and of the right length. Module `tests` is shown in Figure 3. The module declaration `:- module(test,[test/1],[assertions]).` has a third argument, `[assertions]` which indicates that the module uses some extra syntax defined in the library *assertions* which defines the required operators for writing `entry` and `trust` assertions described later in the paper.

```
:- module(lists,[append/3,length/2]).

append([], L, L).
append([E|Es], L, [E|R]) :- append(Es, L, R).

length(L, N) :- var(N), !, llength(L, 0, N).
length(L, N) :- dlength(L, 0, N).

llength([], I, I).
llength([_|L], I0, I) :- I1 is I0+1, llength(L, I1, I).

dlength([], I, I) :- !.
dlength([_|L], I0, I) :- I0<I, I1 is I0+1, dlength(L, I1, I).
```

Figure 2: A very simple lists module

The rest of the paper proceeds as follows: Section 2 describes a number of typical program development scenarios and recalls how *compilation* of modular programs occurs in each of these scenarios. Section 3 introduces some *abstract interpretation* concepts and notation used in the rest of the paper. Sections 4 and 5 then discuss a number of design alternatives which can be considered and the results which will be obtained when performing *analysis* and, respectively, *specialization* of modular programs in each of the typical scenarios of Section 2.

2 Some Characteristic Scenarios

We start by introducing some notation. A *program* P is a finite set of modules $\{m_1, \ldots, m_k\}$. By $imports(m, m')$, $m \neq m'$ we indicate that some (or all) of the predicates exported by m' are imported by m. Figure 4 presents a program P composed of six modules. Modules are represented as boxes and $imports(m, m')$ is represented as an arrow from m to m'. Though a program is generally composed of several modules, there is a distinguished module which defines the entry point to the program. By $main(P)$ we refer to the main module in P. In our example $main(P) = a$. Given a module m, by $imported(m)$ we refer to the set of modules from which m imports some predicates, i.e. $imported(m) = \{m' \in P \text{ s.t. } imports(m, m')\}$. Graphically, a module m' is in

```
:- module(test,[test/1],[assertions]).
:- entry test(X) : ground(X).
:- use_module(qsort).
:- use_module(lists).

test(L):-  length(L,Length), length(Result,Length),
           qsort(L,Result), sorted(Result).

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

Figure 3: A module for testing quicksort

$imported(m)$ iff there is an arrow from m to m'. In our example, $imported(a) = \{b, c\}$. By *dependent(m)* we refer to the set of modules on whose code m depends, i.e., $dependent(m) = \{m' \in P \text{ s.t. } (imports(m, m') \vee \exists m'' \in P \text{ s.t. } imports(m, m'') \wedge m' \in dependent(m''))\}$. Note that the definition of *dependent* is transitive, whereas that of *imports* is not. In our example, $dependent(a) = \{b, c, d, e, f\}$. Note that there may be circular dependencies among modules. In our example, $e \in dependent(d)$ and $d \in dependent(e)$.

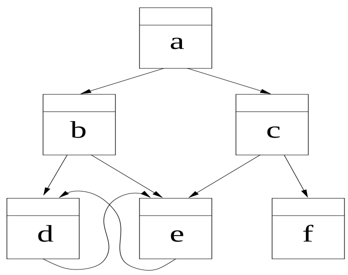

Figure 4: An Example of a Modular Design

We now describe three typical scenarios which appear when dealing with modular programs and which we find of particular practical interest. These scenarios will be used throughout the rest of the paper. In these scenarios, we assume that the *tool* (be it the compiler, analyzer or specializer), is processing a given module (e.g., the one in the current editor buffer). We refer to this module as the *current* module. The different scenarios differ on whether, in addition to the current module, the tool accesses only (some interface information of) the imported modules or it may access all dependent modules, and also on whether the tool processes one module at any point in time or processes several modules simultaneously as one.

2.1 Scenario 1: Dealing with a Single Module (and Related Interfaces)

In this scenario the tool performs its task on the current module without considering the code in any other module. This is a fundamental scenario in modular programming because of its important practical implications: being able to treat properly this situation allows the tool to deal with *incomplete programs*. I.e., the current module can be processed even if the imported modules are still incomplete or completely unavailable. This allows independent development of different parts of the program, which can then perhaps be performed in parallel by different teams. This can for example allow early detection of compile-time errors in the current module without having to wait for the code of the dependent modules to be ready. Another reason why this scenario is important is efficiency, in the sense of the time taken by the processing performed by the tool: clearly, processing a module separately should be more efficient than processing the whole program. However, less than optimal results (in terms of error detection, degree of optimization, etc., depending on the particular tools) may be obtained. Thus, the objective in this scenario is more correctness of the results, rather than optimality.

Because of its practical importance, in the case of compilation this scenario usually receives a special name: *separate compilation*. As examples of this important practical case we consider the compilation of individual "modules" by the Unix C compiler (`cc`) and also the Ciao Prolog standalone compiler (`ciaoc`) [CH99b] [3] `cc` itself performs typically only separate compilation: it is run on a `.c` file and produces a `.o` file containing relocatable machine code. `ciaoc` also performs separate compilation when the `-c` flag is used, compiling the module into a separate object (`.po`) file containing (by default) WAM [AK91] bytecode.

Despite the considerations above, in practice tools typically require that at least some *interface* information be available for the dependent modules in order to be able to do a sensible job on the current module. One of our purposes is to try to identify which is the minimal amount of *interface information* required by the different tools from the related modules to perform their task under each scenario. Typically, in this scenario only information on the imported modules is required. For example, in the case of `ciaoc`, the minimal amount of information needed to process a module and obtain its compiled version is the names/arities of the predicates actually exported by the imported modules. The compiler automatically extracts the interface definition from the source file and stores it in a separate file: the `.itf` –*interface*– file. From that point on, and as long as the source file is not modified, the `.itf` file will be used by the compiler any time the interface part of the corresponding module is needed. Compiling a module requires only its code and the interface files of the imported modules (i.e., the source of the imported modules is not necessary). In the case of `cc`, the needed information is typically added explicitly to the current "module" (as a result, a reduced amount of error checking can be made).[4]

2.2 Scenario 2: Dealing with Several Modules, One-at-a-Time

In this scenario we assume that the tool can access the code of, and process, all the modules in program P. This scenario starts from a request to process a module m which is usually $main(P)$. However, due to the dependencies among modules, in order to process m all modules in $dependent(m)$ (often the rest of modules in the program) may also have to be accessed and processed. Such processing is performed one module at a time, i.e., the tool loads and processes the code of only one module at each step. Thus, in order to deal with this scenario, the tool must be able to change contexts in order to deal with this loading and unloading of modules to be processed. Also, in this scenario typically all modules in $dependent(m)$ and not only those in $imported(m)$ must be processed. Furthermore, since we admit circular dependencies among modules, in order to deal with this scenario the tool has to be able to deal with such circularities correctly. Depending on the task to be performed by the tool, it may be required to process the same module several times. In that case, care must be taken to avoid entering infinite loops and processing terminates when a *fixed point* is reached, i.e., when further iterations do not change the results.

As examples, in a UNIX environment, this scenario is implemented using the `make` application. This corresponds to writing a `makefile` (possibly aided by running the `makedepend` command) followed by issuing a `make` command. I.e., based on the dependencies among modules it is determined which modules have to be recompiled. The Ciao `ciaoc` compiler also automatically performs this process, automatically determining the dependent modules of the current one and follows the dependencies among modules deciding which modules require recompilation to `.po` and finally linking the application, without requiring any input from the user.

[3] The Ciao compiler, itself part of the Ciao library, can be used both from the command line using the `ciaoc` application, and from the familiar interactive toplevel shell. While in the discussion we will mention only `ciaoc`, the descriptions given apply equally to the use of the compiler from the toplevel shell or as a library from another program.

[4] In fairness, C is not really modular – we are using it as an example only because the related compilation tools are very well known.

2.3 Scenario 3: Dealing with Several Modules Simultaneously as One

Though modularity is beneficial from the program development point of view, it usually does not add fundamentally to the expressive power of a language. In fact, a monolithic program can always be constructed which is equivalent to a modular one. The process of constructing such program usually only amounts to renaming predicates in different modules in order to avoid name clashes. Thus, an alternative to scenario 2 in order to deal with a modular program P, in the case in which all the code is available, is to transform P into an equivalent monolithic program P' and then process P' rather than P. We refer to this approach as *scenario 3*. Note that in practice it suffices if the tool can deal with scenario 1 and one of scenario 2 or 3. Which of the latter is more appropriate depends on the particular task the tool has to perform.

As in the case of scenario 2, this scenario usually starts with a request to process a module $m = main(P)$. Using again compilation as an example, this scenario corresponds to first performing module name expansion, then concatenating the code of m with that of all the modules in $dependent(m)$ (often the whole program) and finally running the compiler on the result. This mode is not really supported by `cc/make` nor by `ciaoc` (although it would be relatively easy to build such a file by hand and run it through the compiler). One reason for this is that, as mentioned before, it is sufficient that a tool deal with scenario 2. Compilation based on scenario 3 can be more efficient than that based on scenario 2 when a complete (correct) program is compiled from scratch. However, this scenario is not incremental and any subsequent compilation after some modification of the program is typically much more efficient following scenario 2. Furthermore, consider that user programs very often use library predicates which reside in modules which are typically pre-compiled. Scenario 2 allows avoiding compilation of library modules over and over again.

3 Abstract Interpretation

Program analysis aims at deriving at compile-time certain properties of the run-time behavior of a program. Prior to presenting our proposals regarding analysis of modular programs, we provide some background and notation on *abstract interpretation* [CC77] which is one of the most successful techniques for static program analysis (and the one used throughout in the Ciao system).

We first recall some classical definitions in logic programming. An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. We often use $\bar{t}$ to denote a tuple of terms. A *clause* is of the form H`:-`$B_1, \ldots, B_n$ where H, the *head*, is an atom and $B_1, \ldots, B_n$, the *body*, is a possibly empty finite conjunction of atoms. A *definite logic program*, or *program*, is a finite sequence of clauses.

In abstract interpretation, the execution of the program is "simulated" on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). An abstract value is a finite representation of a, possibly infinite, set of actual values in the concrete domain (D). The set of all possible abstract semantic values represents an abstract domain D_α which is usually a complete lattice or cpo which is ascending chain finite. However, for this study, abstract interpretation is restricted to complete lattices over sets both for the concrete $\langle 2^D, \subseteq\rangle$ and abstract $\langle D_\alpha, \sqsubseteq\rangle$ domains.

Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle\alpha, \gamma\rangle$: *abstraction* $\alpha : 2^D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to 2^D$, such that

$$\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y.$$

Note that in general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall \lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$). Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in some precise sense.

Example 3.1 [A domain for mode analysis] In all our examples we will use the following abstract domain D_α which captures mode information. An abstract substitution λ over a set of variables $\overline{X} = \{X_1, \ldots, X_n\}$ assigns to each variable X_i a value v in the set $\{ground, var, any\}$ where each

v represents an infinite set of terms. The fact that a variable X_i is assigned an abstract value v indicates that X_i will be bound at run-time to some term belonging to v. *ground* is the set of all terms without variables; *var* is the set of unbound variables (possibly aliased to other unbound variables); and *any* is the set of all terms. The abstract domain is complemented by the abstract substitutions $\bot$ and $\top$. As usual in abstract interpretation, $\bot$ denotes the abstract substitution such that $\gamma(\bot) = \emptyset$. The substitution $\top$ is such that $\gamma(\top) = D$. In our domain, $\top$ corresponds to assigning *any* to each variable in $\overline{X}$.

3.1 A Notation for Abstract Substitutions

For the sake of readability, an abstract substitution $\lambda = \{X_1/v_1, ..., X_n/v_n\}$ where each $v_i \in \{ground,\ var,\ any\}$ is represented as the conjunction $(v_1(X_1), ..., v_n(X_n))$. For example, $\{X_1/ground, X_2/var, X_3/any\}$ is represented as $(ground(X_1), var(X_2), any(X_3))$. In addition, statements of the form $any(X_i)$ can be removed. Thus, the above substitution over $\{X_1, X_2, X_3\}$ can be simply written as $(ground(X_1), var(X_2))$. We will use this notation in the examples for both `entry` and `trust` assertions which will be introduced below. Note however that the assertion language used in Ciao [HPB99, PBH99] admits much more general properties in assertions which are also independent from the abstract domain being used. However, we restrict ourselves to abstract substitutions in assertions for simplicity of the presentation.

3.2 Goal-Dependent analysis

Goal-dependent analyses are characterized by generating information which is valid only for a restricted set of calls to a predicate, as opposed to goal-independent analyses whose results are valid for any call to the predicate. Goal-dependent analyses allow obtaining results which are *specialized* (restricted) to a given context. As a result, they provide in general better (stronger) results than goal-independent analyses. In addition, goal-dependent analyses provide information on both the call and success states for each predicate, whereas goal-independent analysis only provide information on success states of predicates. For these reasons, and since program specialization greatly relies on information about call states to predicates, we will restrict the discussion to goal-dependent analyses.

In order to improve the accuracy of goal-dependent analyses, some kind of description of the *initial* calls to the program should be given.[5] With this aim, we will use `entry` declarations in the spirit of [BCHP96]. Their role is to restrict the starting points of analysis to only those calls which satisfy the *assertion* '`:- entry` *Pred* `:` *Call* '. where *Call* is an abstract call substitution for *Pred* in the notation introduced in Section 3.1.[6] For example, the following assertion informs the analyzer that at run-time all initial calls to the predicate `qsort/2` will have a term without variables in the first argument position:

```
:- entry qsort(A,B) : ground(A).
```

The possibly more accurate information generated by a goal-dependent analyzer using the above assertion is valid for any execution of `qsort/2` with the first argument being bound to a term without variables, but may be incorrect for other executions. Goal dependent abstract interpretation takes as input a program P and a set *entries*. This set contains pairs of the form $\langle p, \lambda\rangle$ where p is one of the top-level (exported) predicates and λ is an abstract substitution in the abstract domain D_α, which represents a restriction of the run-time bindings of p. The set *entries* is obtained from the `entry` declarations present for the program. For each declaration '`:- entry` *Pred* `:` *Call* ' a pair $\langle Pred, Call\rangle$ is added to *entries*.

Goal-dependent abstract interpretation computes a set of triples $Analysis(P, entries, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$. In each triple $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$, p_i is an atom and λ_i^c and λ_i^c are, respec-

[5] Predicate calls which are not initial will be called *internal*.

[6] In practice, a more general language, which includes properties defined in the source language, is supported [HPB99].

tively, the abstract call and success substitutions.[7] Due to space limitations, and given that it is now well understood, we do not describe here how we compute $Analysis(P, entries, D_\alpha)$. More details can be found in [Bru91, MH90, MH92, HPMS95, PH96].

Given $Analysis(P, entries, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$, correctness of abstract interpretation guarantees that the following propositions hold:

Proposition 3.2 [Correctness w.r.t. successes] The abstract success substitutions cover all the concrete success substitutions which appear during execution, i.e., $\forall i = 1..n\ \forall \theta_c \in \gamma(\lambda_i^c)$ if $p_i\theta_c$ succeeds in P with computed answer θ_s then $\theta_s \in \gamma(\lambda_i^s)$.

Proposition 3.3 [Correctness w.r.t. calls] The abstract call substitutions cover all the concrete calls which appear during executions described by *entries*. I.e., for any concrete call c originated from an initial goal $p\theta$ s.t. $\exists\langle p, \lambda\rangle \in entries$ with $\theta \in \gamma(\lambda) : \exists\langle p_j, \lambda_j^c, \lambda_j^s\rangle \in Analysis(P, entries, D_\alpha)$ s.t. $c = p_j\theta'$ and $\theta' \in \gamma(\lambda_j^c)$.

Proposition 3.3 is related to the closedness condition [LS91] required in partial deduction.

An analysis is said to be *multivariant on calls* if more than one triple $\langle p, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p, \lambda_n^c, \lambda_n^s\rangle$ $n \geq 0$ with $\lambda_i^c \neq \lambda_j^c$ for some i, j may be computed for the same predicate. Different analyses may be defined with different levels of multivariance [VDCM93]. However, unless the analysis is multivariant on calls, little specialization may be expected in general. Thus, in what follows we restrict ourselves to goal-dependent analyses which are multivariant on calls.

3.3 Aiding the Analysis

Yet another kind of assertions are introduced in [BCHP96] and are intended for use when additional information is to be provided to the analyzer in order to improve its information. An example of this kind of assertions is:

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

which states that upon success `B` is ground, provided that `A` was ground on call. It may be the case that the analyzer cannot prove a `trust` assertion (for example because part of the program is not available or because analysis is not powerful enough), but the analysis will "trust" such assertions and use the information contained in them as if it had been inferred by analysis. Thus, `trust` assertions can be used to analyze incomplete programs (for example, during development of a program or module), by simply providing such an assertion for each predicate which is not implemented. They can also be used to deal with code which is not reachable or understandable by the analyzer, such as predicates written in another programming language. Finally, a `trust` assertion for a predicate p may be used to improve the analysis information for the predicate p. This will happen if the information contained in the assertion is better than that generated by analysis. In that case it may also improve the analysis information of any other predicate p' which depends on p.

More formally, analysis with trust declarations takes as input, in addition to a program P and a set *entries*, a set *trusts* which contains tuples of the form $\langle p_j, \lambda_j^c, \lambda_j^s\rangle$, where p_j is an atom and λ_j^c and λ_j^c are, respectively, abstract call and success substitutions in the abstract domain D_α. The set *trusts* is obtained from the `trust` declarations present for the program. For each declaration '`:- trust success` *Pred* `:` *Call* `=>` *Success* ' a tuple $\langle Pred, Call, Success\rangle$ is added to *trusts*, It is straightforward to modify a goal-dependent analysis in order to perform analysis with trusts.[8]

Algorithm 3.4 [Analysis with trusts] Whenever analysis has to compute the success substitution λ^s which corresponds to an atom p with call substitution λ^c, an improved success substitution λ^s_{trust} is instead computed as follows:

[7] Actually, the analyzers used in practice generate information not only at the *predicate level*, as stated here for simplicity, but also at the *clause literal* level.

[8] In fact, the analyzer in the Ciao preprocessor (PLAI) performs analysis with trusts.

$app\_trusts(p, \lambda^c) = \{\langle p, \lambda^c_j, \lambda^s_j \rangle \in \ trusts \text{ s.t. } \ \lambda^c \sqsubseteq \lambda^c_j\}$
if $app\_trusts(p, \lambda^c) = \emptyset$
  then compute λ^s as usual and $\lambda^s_{trust} = \lambda^s$
  else if p is defined in P
    then compute λ^s as usual
    else $\lambda^s = topmost(\lambda^c)$
    endif
    $\lambda^s_{trust} = (\lambda^s \sqcap \lambda^s_1 \sqcap \ldots \sqcap \lambda^s_m)$ where $app\_trusts(p, \lambda^c) = \{\langle p, \lambda^c_1, \lambda^s_1 \rangle, \ldots, \langle p, \lambda^c_m, \lambda^s_m \rangle\}$
endif

Note that if analysis is *goal-dependent*, the existence of `trust` assertions for a predicate does not avoid analyzing the code of the corresponding code if it is available, as otherwise the internal calls generated in this predicate could be ignored during analysis, resulting in incorrect analysis information. Only after analysis of such a predicate may `trust` assertions be used to improve the analysis information obtained. Note also that if the code of the predicate is not available, the internal calls to predicates in the program that may appear during execution of the missing predicate must have been declared in `entry` assertions for soundness of the analysis. Refer to [BCHP96] for details.

The function *topmost* obtains the topmost success substitution of an abstract call substitution. The notion of *topmost substitution* was already introduced in [BCHP96]. Informally, a topmost substitution of an abstract call substitution keeps those properties which are *downwards closed* whereas it loses those ones which are not. Note that taking $\top$ as the abstract success substitution is always correct but often less accurate than using topmost substitutions. For example, if a variable is known to be ground in the call substitution, it will continue being ground in the success substitution and taking $\top$ as the success substitution would lose this information. However, the fact that a variable is free on call does not guarantee that it will keep on being free on success.

4 Analysis of Modular Programs

In this section we discuss how we can perform goal-dependent analysis of modular programs using abstract interpretation and trust declarations. We first study the problems when trying to obtain *correct* and *optimal* analysis of an *incomplete* program. Then we consider the different scenarios introduced in Section 2.

4.1 Problems with Analysis of Incomplete Programs

By an *incomplete* program we refer to a program whose code is not completely available to the analyzer. There are several reasons why this may happen. One is *dynamic code*, i.e., code which is not available until run-time. The problems which appear in analysis of dynamic code are already studied in depth in [BCHP96]. Another typical situation in which analysis does not receive the code of the whole program is when we analyze a module separately, as is done in scenarios 1 and 2. If our goal is to provide a correct but possibly inaccurate analysis, two problems appear, as already stated in [BCHP96], which are:

The success substitution problem: in an incomplete program P there may be clauses of the form $H\texttt{:-}B_1, \ldots, B_n$ such that the definition for the predicate p called in B_i, $i \in \{1, \ldots, n\}$ is not (completely) available in P. When analysis encounters an abstract call λ^c for p, analysis of P must include a tuple of the form $\langle p, \lambda^c, \lambda^s \rangle$. The substitution λ^s used has to be such that correctness w.r.t. successes is preserved. Note that an incorrect λ^s will also produce incorrect substitutions for other predicates which depend on p.

The extra call pattern problem: the clauses missing in an incomplete program P may include clauses of the form $H\texttt{:-}B_1, \ldots, B_n$ where B_i, $i \in \{1, \ldots, n\}$ is a literal for a predicate p defined in P. Analysis has to take into account such calls which are not visible in the code available at analysis-time, in order to preserve correctness w.r.t. calls for predicate p.

Example 4.1 Consider separate analysis of module `qsort` in Figure 1. In order to start analysis we need a set of entries for the module. The success substitution problem appears since we require some abstract success substitution for predicate `append/3`. The extra call patterns problem may also appear if the entries used are not general enough to include the call to `qsort` from the module `test`.

However, if the goal is to obtain *optimal* analysis, i.e., one which is both correct and as accurate as possible, then the problems we have to face are:

The extra call pattern problem: same as before.

The optimal success problem: this problem replaces the success substitution problem. This new problem is harder to solve. It corresponds to, given a call λ^c for a predicate p not defined in the program, not only to finding a success λ^s which is correct, but also to finding the best, i.e., the most accurate one among them, i.e., $\forall \lambda_i^s$ which is correct $\lambda^s \sqsubseteq \lambda_i^s$.

The optimal calls problem: in modular programs, the code of a predicate p not defined in P but defined in another module m' will eventually be available. As we will see below, one way of computing the success substitutions for p is to analyze the module m'. Since analysis is goal-dependent (and multivariant on calls), in order to have the most accurate possible success substitutions λ^s for p also the most accurate possible call substitutions λ^c should be considered.

Example 4.2 In analysis of predicate `qsort`, taking $\top$ as success substitution for `append` is trivially guaranteed to be correct, however, such substitution is clearly not optimal. The optimal calls problem requires that the predicate `append` is analyzed with call pattern `:- entry append(A,B,C) : (ground(A),ground(B)).` Analysis of `append` with $\top$ as call substitution would provide results which are also correct but not optimal, since it does not allow concluding that `C` is also ground on success.

4.2 Scenario 1: Analysis of a Single Module

In this scenario we aim at performing analysis of a single module, much in the spirit of separate compilation. However, the kind of analysis we are interested in is *global*, i.e., the results of analysis of one part (module) of the program may be used in other parts (modules) of the program. Thus, there is seemingly a contradiction between *global analysis*, which in principle requires the code of all of the program, and single module analysis.

Solving the Extra Call Patterns Problem Consider for example analysis of module `qsort`. The first thing to note is that in order to start separate analysis of a module, we must provide a set of entries which will be the starting point of goal-dependent analysis. Since all exported predicates must be explicitly declared as such, and in order to pose the least possible burden on the programmer, the `module` declaration can be used to automatically build an entry declaration per exported predicate with $\top$ as call substitution. Since the module system used is strict, only those predicates which are exported can be called from outside this module, and are in principle the ones which should appear in the entries. In our case, the automatically generated set of entries is $entries = \{\langle qsort(A,B), \{A/any, B/any\}\rangle\}$. Note that this already solves the extra call patterns problem.[9] However, it is clear that such entries do not provide much information to the analyzer and the user should be able to provide more accurate information on entries if so desired. This is easy to do using additional `entry` declarations. In our example, we could add the declaration `:- entry qsort(A,B) : ground(A).` Note that such declaration should be general enough in order to include all possible calls to the module from outside and do not incur in the extra call patterns problem. For example, we may be tempted to also state `var(B)`. However, it would be incorrect since in the module `test` qsort is called with `(ground(B),any(B))`. The information in the user-provided entry declarations replaces the one with $\top$ which is automatically generated.

[9] The extra call patterns problem could still appear if the module system were not strict.

Solving the Success Substitution Problem Another thing to note is that the module `qsort` uses a predicate defined in another module (`lists`). Thus, the success substitution problem may appear. If we try to apply directly $Analysis(qsort, entries, D_\alpha)$ with the set $entries = \{\langle qsort(A, B), \{A/ground, B/any\}\rangle\}$, then the results obtained are no longer guaranteed to be correct w.r.t. successes, since analysis returns the empty success substitution $\bot$ for any predicate which is not a builtin and whose definition is not included in module `qsort` regardless of whether the undefined predicate is defined in some other module. Thus, rather than $\bot$ we have to use correct success substitutions for `append`. Again, there is a simple and automatic (but possibly inaccurate) solution to this problem. Rather than using $Analysis(P, entries, D_\alpha)$ we should use $Analysis\_with\_trusts(P, entries, D_\alpha, trusts)$ where the set $trusts$ contains a tuple of the form $\langle q, \top, \top\rangle$ for each predicate q defined in the $imported(P)$. This simple solution is guaranteed to provide correct results. Since the code for `append` is not included in `qsort`, $Analysis\_with\_trusts(qsort, entries, D_\alpha, \{\langle append, \top, \top\rangle\})$ with the same entries as above will return topmost substitutions for any call to imported predicates. Another situation in which we can use more accurate information on imported predicates is when the imported module has already been analyzed. Suppose that the code for `append` is available, and that the module `lists` has been analyzed with an entry substitution λ^{entry} which is applicable, i.e. $\lambda^c \sqsubseteq \lambda^{entry}$ and the computed success substitution is λ^{succ}. In that case, the computed λ^{succ} can be taken as success substitution for `append`. If the predicate-level results of the analysis are written as assertions, this can simply be done by adding to the corresponding `.asr` file the assertions which correspond to the exported predicates.[10]

Once again, the automatic approaches may produce a considerable loss of precision. Thus, the user should be able to provide more accurate information on the success substitutions of predicates defined in other modules if so desired. For example, we can add to the `lists` module the declaration `:- trust success append(A,B,C) :(ground(A),ground(B)) => ground(C).` Thus, in order to analyze a module m in scenario 1, we should access $imported(m)$ and collect the existing trust declarations for the exported predicates. Such information can be stored in an auxiliary file, much in the same way as with the interface file. In the case of the Ciao system, this information is stored in a file with extension `.asr`. In fact, this information could be added to the `.itf` file. However, they are kept in separate files because the low-level compiler does not need such information and thus it would unnecessarily slow-down processing `.itf` files.

4.3 Scenario 2: Analysis of Several Modules, One-at-a-Time

In the previous section we have discussed several ways of performing correct analysis of separate modules. However, in scenario 2 we are concerned not only with correctness but also with accuracy, since as already mentioned in Section 2, scenarios 2 and 3 should provide equivalent results. It is important to note that it cannot be guaranteed in general that the results obtained in scenario 1 are as accurate as those which could be obtained if all the code in the program were available to analysis. There are two reasons for this inaccuracy. One is related to the possible inaccuracies of the entries for other modules (the optimal calls problem) and the second to the possible inaccuracies of the success information given in the trusts which correspond to such queries (the optimal success problems). Note that for goal-dependent global analysis in order to obtain optimal solutions, and thus solve the two problems mentioned above, two flows of information are required. One propagates information about calls in a top-down fashion, i.e., from the callers to the callees, whereas the other flow of information propagates information about successes in a bottom-up fashion, i.e., from the callees to the callers. In scenario 2 we are allowed to analyze the code of as many modules as needed, and as many times as required. Unfortunately, even if there are no circular dependencies among modules, there is no fixed order for handling (analyzing) modules which guarantees obtaining the best possible information in a fixed number of iterations. In fact, a global fixed point has to be computed which may require analyzing a module an unbounded

[10] Such assertions are different in nature from the `trust` assertions added by the user. For a more detailed discussion on this topic we relate the reader to [HPB99].

number of times until analysis terminates. We refer to this fixed point as a *distributed global fixed point.*

Solving the optimal calls problem: In scenario 1, entries have to be provided to each module we want to analyze separately. They can be automatically generated and then be as general as possible, or they can be given by the user, but in any case they are not guaranteed to be optimal. In scenario 2, analysis is typically started from $main(P)$. Analysis of each module m may generate calls to other modules in $imported(m)$. In order to solve the optimal calls problems for modules other than $main(P)$, the above mentioned calls should be collected and be the starting point of analysis for the imported modules. This can be automatically obtained by adding to such modules the entry declarations which correspond to the calls generated during analysis of other modules.

Solving the optimal success problem: Assume that we are analyzing a module m and we reach a program point in which there is a call λ^c to a predicate p imported from another module m'. Deciding which success substitution to use for p and λ^c corresponds to the optimal success problem. If the module m' has already been analyzed with p and λ^c as entry, and thus an optimal success substitution exists, then it should be used. Otherwise, there are at least two possibilities regarding how to proceed with the analysis of m:

1. Assume $\perp$ as an accurate (but possibly incorrect) success substitution. Then we continue the analysis of m until a fixed point for the module is reached.

2. Freeze analysis of m and start analysis of m' taking p and λ^c as entry. Once the analysis of m' is finished, take as optimal success the one just computed.

Both approaches have pros and cons. Possibility 1 may be inefficient because the success substitution $\perp$ is very likely to be incorrect. Thus it may perform a lot of *speculative* work, i.e., work which may end up being useless. Also, the analysis results of every module in the program must be taken with care. Whenever the success substitution of an exported predicate is updated, analysis of the modules which import such predicate also has to be updated. The results of analysis are not guaranteed to be correct until a distributed fixed point is reached. The inefficiency of possibility 1 is less dramatic when incremental analysis is used. In that case, the previous analysis results for the module are used in order to compute the new local fixed point. For this, the *incremental addition* algorithm of [HPMS95] can be used. This is not a great restriction since incremental analysis algorithms can be as fast as non-incremental ones [PH96].

Possibility 2 has the advantage of not performing any *speculative* work as analysis does not continue until an optimal success substitution is computed. However, care must be taken when there are circular dependencies among modules, as we may end up in a deadlock or in an infinite loop. Also, this possibility requires that the analysis engine be capable of freezing an analysis, starting another one (which may in turn be frozen), and resuming computation of the old analysis after that. Unfortunately, none of the existing analysis engines for logic programs that we are aware of can be used directly in this way. This is because analysis engines are rather complex systems which are specialized to obtain maximum efficiency in the particular case in which all the program is available at analysis time.

Example 4.3 Consider analysis of the program composed by the modules `test`, `qsort`, and `lists` using scenario 2 and starting with module `test`. Since $imported(test) = \{qsort, lists\}$, analysis of module `test` has to take into account previous analysis results for modules `qsort` and `lists`. These are denoted $trusts_{qsort}$ and $trust_{lists}$ respectively. Similarly, analysis for `qsort` has to take into account the results of `lists`. Analysis of `lists` does not depend on any other module.

Figure 5 shows a possible sequence of analyses which can be performed and which lead to a distributed fixed point. In the figure, *ground* is abbreviated to *gr*. Initially, both $trusts_{qsort}$ and $trust_{lists}$ are empty since no analysis has been performed yet. We denote with different numbers the different analyses which are started in the process. The initial one is denoted 1 and corresponds to the initial entry `:- entry test(X) : ground(X).`. Analysis of predicate `test/1` processes in

$$
\begin{array}{ll}
0 & trusts_{lists} = \emptyset \wedge trusts_{qsort} = \emptyset \\
1 & Analysis\_with\_trusts(test, \{\langle test(X), gr(X)\rangle\}, trusts_{lists} \bigcup trusts_{qsort}) = ? \\
2 & Analysis(lists, \{\langle length(L,N), (gr(L), var(N))\rangle\}) = \\
 & \qquad \{\langle length(L,N), (gr(L), var(N)), (gr(L), gr(N))\rangle\} \\
 & trusts'_{lists} = \{\langle length(L,N), (gr(L), var(N)), (gr(L), gr(N))\rangle\} \\
1' & Analysis\_with\_trusts(test, \{\langle test(X), gr(X)\rangle\}, trusts'_{lists} \bigcup trusts_{qsort}) = ? \\
3 & Analysis(lists, \{\langle length(L,N), (var(L), gr(N))\rangle\}) = \\
 & \qquad \{\langle length(L,N), (var(L), gr(N)), (any(L), gr(N))\rangle\} \\
 & trusts''_{lists} = trusts'_{lists} \bigcup \{\langle length(L,N), (var(L), gr(N)), (any(L), gr(N))\rangle\} \\
1'' & Analysis\_with\_trusts(test, \{\langle test(X), gr(X)\rangle\}, trusts''_{lists} \bigcup trusts_{qsort}) = ? \\
4 & Analysis\_with\_trusts(qsort, \{\langle qsort(A,B), (gr(A), any(B))\rangle\}, trusts''_{lists}) = ? \\
5 & Analysis(lists, \{\langle append(A,B,C), (gr(A), gr(B), any(C))\rangle\}) = \\
 & \qquad \{\langle append(A,B,C), (gr(A), gr(B), any(C)), (gr(A), gr(B), gr(C))\rangle\} \\
 & trusts'''_{lists} = trusts''_{lists} \bigcup \\
 & \qquad \{\langle append(A,B,C), (gr(A), gr(B), any(C)), (gr(A), gr(B), gr(C))\rangle\} \\
4' & Analysis\_with\_trusts(qsort, \{\langle qsort(A,B), (gr(A), any(B))\rangle\}, trusts'''_{lists}) = \\
 & \qquad \{\langle qsort(A,B), (gr(A), any(B)), (gr(A), gr(B))\rangle\} \\
 & trusts'_{qsort} = \{\langle qsort(A,B), (gr(A), any(B)), (gr(A), gr(B))\rangle\} \\
1''' & Analysis\_with\_trusts(test, \{\langle test(X), gr(X)\rangle\}, trusts'''_{lists} \bigcup trusts'_{qsort}) = \\
 & \qquad \{\langle test(X), gr(X), gr(X)\rangle\}
\end{array}
$$

Figure 5: Analysis of the example in scenario 2

the body of the clause of this predicate from left to right. The first literal is a call to `length` which is defined in another module. Thus, analysis 1 cannot be completed yet and this is indicated in the figure with a question mark. Then, the call pattern to `length` is taken as an entry for analysis of module `lists`, which is denoted as analysis 2 in the figure. When analysis 2 is completed, the set $trusts_{lists}$ is updated and analysis returns to the incomplete analysis 1. This is denoted by 1' in the figure. Scenario 2 proceeds by triggering new analyses whenever new entries for other modules are generated and by updating the *trusts* sets and revising incomplete analyses whenever new success substitutions used in such analyses are generated. This process continues until a distributed fixpoint is reached. Several control strategies can be used for guiding scenario 2 and we are currently experimenting in the Ciao system with different ones. However, it is out of the scope of this paper to provide a detailed discussion of such strategies.

Reducing Memory Consumption: In both of the possibilities seen above, analysis has to switch contexts from one module to another, either once a fixed point has been reached or even in an intermediate state (in possibility 2). Thus, in the worst case the analyzer may end up with the analyses of all modules in the program stored in memory at the same time. Clearly this situation is similar to that of scenario 3 and the system may run out of memory. If we want to reduce the amount of memory required by analysis, rather than keeping in memory the analysis for all modules seen up to now, we may decide to store some or all of the analysis for modules other than the current one in disk and restore the state of analysis when analysis returns to a module stored in disk. The more incremental the analysis algorithm is, the more difficult it is to be able to dump and restore the current state of the analysis of a module. For example, in possibility 1, if incremental analysis is not used, there is no point in dumping to disk the state of analysis, since when analysis of that module has to be resumed it has to be started from scratch anyway. Dumping analysis information for possibility 1 and incremental analysis is not too hard to do. In fact, this is implemented in the Ciao system. However, for the case of possibility 2 we still do not (yet) support dumping and restoring an analysis which is not in a fixed point state since this requires resuming analysis for the exact intermediate analysis situation in which the analysis was frozen.

Which of the above mentioned alternatives is best needs experimentation, which is currently being carried out by our group and we hope to report on shortly.

4.4 Scenario 3: Analysis of Several Modules Simultaneously as One

This scenario poses no theoretical difficulties to analysis since the traditional algorithms used for non-modular programs can be applied. Unlike in the case of compilation, at first sight scenario 2 does not seem very appropriate for analysis, since on one hand it is complicated to implement, and on the other hand we may have to swap modules many times, and this seems to favor scenario 3 in terms of time-efficiency.

However, though scenario 3 seems preferable in principle, there are a couple of considerations which should also be taken into account when choosing between scenario 2 and 3. One is that if the program being analyzed is large, global analysis using scenario 3 may run out of memory. In that situation, one-at-a-time analysis is preferable, as the memory required to analyze modules separately is less than that required for analyzing the whole program at once. Thus, one-at-a-time analysis may turn some programs which analysis cannot handle into tractable ones in return of a somewhat increased analysis time. The second consideration is that scenario 2 can avoid repeated reanalysis of modules, much in the same way as in separate compilation. If neither the code for a module nor the code of the modules the module depends on has changed and the module has already been analyzed for the call pattern of interest, then scenario 2 can avoid recomputation for such call pattern.

5 Specialization of Modular Programs

Program Specialization [JGS93, Gal93, Leu97] aims at optimizing programs by specializing the code to particular cases. Though much of the discussion we present could apply to other kinds of specialization, we will focus on *abstract multiple specialization* [JLW90, GH91, Bru91, Win92, PH95, KMM+96, PH99] which directly uses the results of global analysis in order to optimize the program introducing multivariant specialization if required.

Example 5.1 Consider the `length` predicate in Figure 2. This is a good example of a reversible predicate which can be used in several modes. For example, in the module `test` in Figure 3, there are two different calls to the predicate `length`, which use such predicate in different ways. In the first call, the length of a list is computed and in the second one a list (skeleton) of a fixed length is constructed. This generality forces the code of `length` to consider two cases depending on whether the second argument N, i.e., the length of the list is fixed or not. If N is not fixed, i.e., N is a variable, then the predicate `llength` is used. If N is fixed, the predicate `dlength` is used. Thus, if analysis information allows determining that the second argument will be a variable (resp. non variable) at run-time, a call to `length` can be replaced by a call to `llength` (resp. `dlength`).

In this section we discuss different issues which appear when considering (abstract) specialization of programs split into modules. We assume that the program has already been analyzed according to the scenarios and algorithms presented in Section 4.

An important feature of the multiple specialization algorithm in [PH95, PH99] is that it allows minimizing the number of versions implemented in the final program. For this, there is a flow of information among modules which propagates information bottom-up and corresponds to the potential optimizations which are possible in different versions of a predicate (if they were materialized). This information is required in order to minimize the number of versions without losing opportunities for specialization.

As in the case of analysis we may need to perform iterations until reaching a distributed fixed point. However, and unlike in the case of analysis, for programs without circular dependencies among modules there is a processing order of the modules which guarantees obtaining the best solution possible while only processing each module once.

5.1 Scenario 1: Specialization of a Single Module

In this scenario we should be able to perform specialization of a particular module m without having to specialize other ones. As usual in scenario 1, some information on the imported modules could be required. In this case, the minimal information from each module m' in $imported(m)$ corresponds to knowing the names of the specialized versions which have already been generated for the predicates exported by m' and also the conditions which guarantee that their usage is correct in a particular call. As is the case with analysis results, we propose to provide the information about specialized versions in each module m' by means of (novel) assertions. Such information can be written in the `.asr` file for m' or in a separate file if so desired. For example, the following assertion:

```
:- true pred length(L,N) : var(N) + equiv(llength(L,0,N)).
```

states that a call to `length` with the second argument being a variable can be optimized by replacing the call by a call to `llength`. These assertions contain information which in fact corresponds to the *abstract executability tables*[11] used in [PH99]. Thus, adapting the abstract specializer to understanding the information in these assertions is not a difficult task and is the subject of ongoing work.

Note that in this scenario, we can decide on a modular basis on which parts of the program we want to perform multivariant specialization, and on which other ones we are only interested in monovariant specialization[12] or even no optimization, for example in order not to reduce readability of the code. Thus, in the case of specialization, scenario 1 is also very relevant as it allows a much more fine-grained control on the effect of multiple specialization.

Scenario 1 also fits very well with the idea of having a set of *precooked* specialized versions of predicates. This can easily be achieved by starting analysis from a set of entries which we consider of particular interest and which we believe will give rise to useful optimizations. Then we only have to perform automatic multiple specialization using scenario 1. Note that in order to obtain maximal benefits of the set of precooked specialized versions, the conditions on their applicability should be as weak as possible while remaining correct. Note that though the specialization process is fully automatic, finding the starting set of entries which are of interest from the multiple specialization point of view is not automatic at this stage and is a topic of future research. A particular case in which precooked specialized versions make a lot of sense are libraries. Most modern Prolog systems have a large set of predicates which are already implemented in Prolog but which are not predefined in the language. Users can use them provided they include the corresponding library in their programs. Scenario 1 then avoids re-analyzing and re-specializing such libraries over and over again.

Example 5.2 We can analyze the example library module `lists` for the entries:

```
:- entry length(L,N) : var(N).
:- entry length(L,N) : ground(N).
```

and using the analysis information obtain the following equivalences:

```
:- true pred length(L,N) : var(N)    + equiv(llength(L,0,N)).
:- true pred length(L,N) : ground(N) + equiv(dlength(L,0,N)).
```

The code of the `lists` module remains the same in this case after specialization. The only change is that the predicates `llength` and `dlength` must also be exported.

If we now analyze the module `test` with entry `:- entry test(X) : ground(X).` we can specialize the code of predicate `test` to:

```
test(L):-  llength(L,0,Length), dlength(Result,0,Length),
           qsort(L,Result), sorted(Result).
```

[11]They contain conditions under which builtin predicates can be reduced to *true*, *false* or a set of unifications.

[12]Though multivariant specialization is in principle more powerful than monovariant specialization, it may also significantly increase code size.

5.2 Scenario 2: Specialization of Several Modules, One-at-a-Time

This scenario is expected to produce the same results as scenario 3. Since in scenario 3 the analysis information is optimal, and since specialization is based on the results of analysis, scenario 2 can only be performed if the analysis information available is optimal. Thus it does not make much sense to use scenario 2 for specialization if analysis used scenario 1.

In order to present how to perform specialization in scenario 2 we will consider two cases. The simple one is when there are no circular dependencies among modules. In that case, we should use the dependency graph among modules and start performing the minimization algorithm on the leaf nodes of the graph (which is in this case a tree). Since leaf modules do not depend on other ones, they can be treated as self-contained programs and the usual algorithm applies. Then we have to consider the specialized versions which have been generated for the exported predicates. We take as the condition to be able to use such specialized version that the call substitution corresponds to the call substitution which is associated to the specialized version. For each such specialized version, an `equiv` assertion is generated and written out on the interface part of the module, for example on the `.asr` file. The minimization algorithm should not be performed on a module until all the imported ones are already specialized. Specialization of a module m which is not a leaf requires taking into account the interface of the imported modules as well as the code of m. In order to annotate the possible optimizations which are directly applicable to each call substitution to a predicate in m, in addition to looking at the abstract executability table, which applies to builtin predicates, we also have to see whether the information in the call allows replacing a call to a predicate p defined in another module m' by a specialized version of p. Conceptually, this is equivalent to considering a *dynamic* abstract executability table in the sense that it is extended as specialization proceeds. In the case of specialization of self-contained programs, it suffices to consider a fixed (static) abstract executability table.

The second, and more complicated case corresponds to having circular dependencies among modules. In such case, we must still process the modules in a bottom-up fashion, however the *strongly connected components* (SCCs) of the dependency graph should be processed together and we may have to process modules in the same SCC several times until a fixed point is reached, i.e., no more specialized versions are generated.

5.3 Scenario 3: Specialization of Several Modules Simultaneously as One

Once again, scenario 3 does not pose theoretical difficulties to program specialization. We have seen in the previous section that specialization according to scenario 2 achieves the same results as those which would be obtained in scenario 3. Then, the main question is which of scenario 2 or 3 is preferable in general. Unlike for the case of analysis, in which an approach based on scenario 2 is usually less efficient in time that one based on scenario 3, in the case of specialization, scenario 2 does not add extra difficulties nor inefficiencies to program specialization. In fact, though it may be argued that some adaptation has to be performed on the specializer in order to deal with scenario 2, it is also true that it may significantly reduce the *distance* between the two fixpoints which are obtained during the *reunion* and *splitting* phases [PH99] of the minimization algorithm. In particular, scenario 2 may avoid collapsing into the same version of a predicate different calls which will end up being in different implementations, since keeping them separate allows optimizing other predicates called by them.

Example 5.3 Consider the program composed of the modules `main` and `p` below and the `lists` module already seen in Figure 3.

```
:- module(main,[main/1]).
:- entry main(L) : ground(L).
main(L):- p(L), p(L1).
```

```
:- module(p,[p/1]).
p(L):-  ..., length(L,N), ...
```

We do not show all the code in module `p` since we are only interested in showing a typical situation which occurs in abstract multiple specialization: it is possible to use the specialized versions of a predicate (`length` in this case) but in order to do that we have to also generate specialized versions of some intermediate predicate (`p` in the example). In this case, analysis would generate two versions of predicate `p`, one for its argument being *ground* and another one for *var*. If we use scenario 3, both versions of `p` are collapsed into one during the *reunion* phase of the algorithm since the optimizations allowed in their code is the same for both (no optimizations). However, such versions are separated during the *splitting* phase since they are required in order to create a path from the different calls to `p` in the body of predicate `main` to the specialized versions of `length`. The modules `main` and `p` after specialization are shown below.

```
:- module(main,[main/1]).
:- entry main(L) : ground(L).
main(L):- p1(L), p2(L1).
```

```
:- module(p,[p1/1,p2/1]).
p1(L):- ..., dlength(L,0,N), ...
p2(L):- ..., llength(L,0,N), ...
```

However, using scenario 2, the order in which modules are specialized (bottom-up) is: `lists`, `p`, and `main`. After specializing `lists`, two optimized versions for `length` are generated and the corresponding equivalence assertions added to the module. These assertions are now part of the abstract executability table. When specialization reaches module `p`, the two versions of predicate `p` are *not* collapsed during the reunion phase since in their code different optimizations are possible (using `llength` in one version and `dlength` in the other).

Acknowledgments

The authors would like to thank Francisco Bueno for many interesting discussions on analysis of modular programs, and the anonymous referees for their constructive comments. This work was funded in part by projects DiSCiPl (ESPRIT LTR 22532 and CICYT TIC 97-1640-CE) and ELLA (CICYT TIC 96-1012-C02-01).

References

[AK91] Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT Press, 1991.

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CH99a] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems.* N.M. State U., December 1999.

[CH99b] D. Cabeza and M. Hermenegildo. The Ciao Modular Compiler and Its Generic Program Processing Library. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems.* N.M. State U., December 1999.

[CLI97] The CLIP Group. CIAO System Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, Facultad de Informática, UPM, August 1997.

[Gal93] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[GH91] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[HPB99] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[HPMS95] 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.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prenctice Hall, New York, 1993.

[JLW90] D. Jacobs, A. Langen, and W. Winsborough. Multiple specialization of logic programs with run-time tests. In *1990 International Conference on Logic Programming*, pages 718–731. MIT Press, June 1990.

[KMM+96] A. Kelly, A. Macdonald, K. Marriott, P. Stuckey, and R. Yap. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 37–51. MIT Press, 1996.

[Leu97] Michael Leuschel. *Advanced Techniques for Logic Program Specialisation.* PhD thesis, K.U. Leuven, May 1997.

[LS91] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3–4):217–242, 1991.

[MH90] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[PBH99] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming.* Springer-Verlag, 1999. To Appear.

[PH95] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

[PH96] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[PH99] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[VDCM93] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michael. The Impact of Granularity in Abstract Interpretation of Prolog. In *Workshop on Static Analysis*, number 724 in LNCS, pages 1–14. Springer-Verlag, September 1993.

[Win92] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.