ASAP

IST-2001-38059

Advanced Analysis and Specialization for Pervasive Systems

Specialization of Real Life CLP Languages

Deliverable number:	D6
Workpackage:	Integrated Tool (WP7)
Preparation date:	1 March 2004
Due date:	1 March 2004
Classification:	Public
Lead participant:	Tech. Univ. of Madrid (UPM)
Partners contributed:	Tech. Univ. of Madrid (UPM), Univ. of Southampton, Roskilde Univ

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

Short description:

This deliverable reflects the progress made in this period in WP7 Tasks 7.1 and 7.2, on the improvement of the mechanisms for analyzing and specializing real-life programs and the integration of these mechanisms into the first prototype. The two main areas of focus of the deliverable are developing techniques for a) dealing with large programs that are divided into modules (including the case when each individual module is large), and b) dealing with language features which are difficult to handle, such as builtins and "impure" predicates (including those that have side-effects, are extra-logical, may raise errors or exceptions, etc.).

As decided in the first progress report, the work in these tasks has continued beyond the originally planned month 12 into month 16 in order to produce additional results. For this reason, only a draft version of this deliverable was available in the first review. That draft version contained only Part IV and preliminary versions of Parts I and VI of this final version. This final deliverable includes all new Parts II, III and V, updated versions of Parts I and VI, and, in order to be self-contained, also Part IV unchanged.

1 Dealing with large, modular programs

As mentioned above, one of the fundamental areas of focus of this deliverable is to study and find solutions to the difficulties posed by large, real-life, modular programs. Part I of the deliverable ("A Generic Framework for Context-Sensitive Analysis of Modular Programs", which has now been published as a book chapter) presents the complete framework developed for the analysis and specialization of modular programs. This framework analyzes or reanalizes each module as needed, iterating over the module dependency graph and propagating among dependent modules the analysis results obtained, until the analysis results stabilize (i.e., a fixpoint is reached). The framework allows preprocessing a program fragment (a module) even if other parts of the system are not available yet, with only minimal interface information needed. Although results of preprocessing a separate module may be suboptimal, they often suffice for debugging the module code. When all the code of the program is available a more accurate analysis and specialization can be carried out. The increased precision obtained allows detecting new bugs that are due to module interactions and also to produce a more highly optimized executable for the modular program.

Part II of the deliverable ("Experiments in Context-Sensitive Analysis of Modular Programs") reports on our additional progress in this area: the experimental evaluation of the behavior of context-sensitive analysis and specialization of real-life programs decomposed in modules. Since the previous period we have completed an implementation of the framework in `CiaoPP` and assessed its behavior and performance with a set of large, modular programs. We have benchmarked the different models of analysis of modular programs proposed in previous work, each with different characteristics and representing different trade-offs. We provide an empirical comparison of these different models, as well as experimental data on the different choices left open in those designs. We have also explored the scalability of these models by using larger modular programs as benchmarks. In addition, the performance of the system when reanalyzing a modular program after changes in the source code has also been measured, in order to explore the ability of this approach to handle efficiently incremental changes in large systems. This assessment shows that in some critical cases, the incremental approach provides significantly better performance results than those achievable by analyzing the whole program at once.

Part III of the deliverable ("Efficient Local Unfolding with Ancestor Stacks for Full Prolog", to appear in the LOPSTR'04 proceedings) reports on our progress in improving the ability of our specialization tools to deal with individual program modules which are large. This is important because one of the issues that has prevented the integration of powerful partial evaluation methods into practical compilers to date has been their efficiency. The most successful unfolding rules used currently are based on structured orders applied over (covering) *ancestors*, which introduce significant overhead. In this period, we have proposed an efficient, practical *local* unfolding rule based on the notion of covering ancestors which can be used in combination with any structural order and allows a stack-based implementation without losing any opportunities for specialization. We have integrated these techniques in the common tool, in particular in the partial evaluator embedded in `CiaoPP`, and studied the resulting performance. Our experimental results show that they are significantly more efficient in time and somewhat more efficient in memory usage than previous techniques.

2 Handling builtins and impure features

The second fundamental focus of the deliverable in order to deal with real-life programs is to study and find solutions to the difficulties posed by certain language features which which are difficult to handle. In particular, CLP languages typically have a number predefined predicates, or *builtins*, which must be handled in a special way by analyzers and specializers. The usual approach is to embed in the tool a table of the builtins that are understood by the tool, with specific mechanisms to handle them. However, this approach is not extensible. We have proposed instead an approach in which assertions describing builtins are used throughout the libraries of the system (as opposed to the table embedded in the analyzer or specializer) which include the appropriate information. The same approach can be used for parts of an application which are written in foreign languages (e.g., C or Java). These assertions are presented in detail in the manual of the Ciao and CiaoPP systems, and their use in partial evaluation is discussed for example already in Part III.

Another important problem related to builtins is to guarantee that such builtins are called with the correct argument modes and types. This can be tackled by a *backwards analysis*. Conditions on the entry predicates are derived, which guarantee satisfaction of the assertions defining the pre-conditions on builtins. In Part IV ("A Program Transformation for Backwards Analysis of Logic Programs," which has been published in the Proceedings of LOPSTR'03) we presented a framework for performing backwards analysis within a standard abstract interpretation. The approach is based on a transformation that makes the dependencies of the calls to builtins on the initial goals explicit. The transformed program can be analyzed using a standard abstract interpretation algorithm, rather than the special-purpose frameworks used for backwards analysis in the literature.

Part V of the deliverable ("Flexible and Accurate Partial Deduction of Full Prolog using Assertions and Backwards Analysis") reports significant additional progress that we have made during this period in the area of partial deduction of real-life CLP programs containing *impure* predicates. Impure predicates include those which may raise errors, exceptions or side-effects, external predicates whose definition is not available, etc. Existing proposals allow obtaining correct residual programs while still allowing non-leftmost unfolding steps, but at the cost of accuracy: bindings and failure are not propagated backwards to predicates which are classified as impure. Motivated by recent developments in the *backwards* analysis of logic programs, we have developed, implemented and integrated in the common tool a partial deduction algorithm which can handle impure features and non-leftmost unfolding in a more accurate way and, thus, we have made possible and show some optimizations which were not feasible using previously proposed partial deduction techniques. We believe this is an important step forward compared to existing approaches since the method developed is a) accurate, given that the classification of pure versus impure is done at the level of atoms instead of predicates, b) flexible, as the user can annotate programs using assertions, which can guide the partial deduction process, and c) automatic, since backwards analysis can be used to automatically infer the required assertions.

Finally, we have also studied the precise requirements and implications that a precise analysis of builtins imposes on abstract domains. In particular, Part VI ("Set-Sharing is not always redundant for Pair-Sharing", now published in FLOPS'04) studies the case of the important and popular "sharing" domain and proves that certain assumptions that had been made to simplify the operations associated with this domain are in fact not valid in the presence of builtins.

Contents

Part I

A Generic Framework for Context-Sensitive Analysis of Modular Programs

1 Summary

Context-sensitive analysis provides information which is potentially more accurate than that provided by context-free analysis. Such information can then be applied in order to validate/debug the program and/or to specialize the program obtaining important improvements. Unfortunately, context-sensitive analysis of modular programs poses important theoretical and practical problems. One solution, used in several proposals, is to resort to context-free analysis. Other proposals do address context-sensitive analysis, but are only applicable when the description domain used satisfies rather restrictive properties. In this paper, we argue that a general framework for context-sensitive analysis of modular programs, i.e., one that allows using all the domains which have proved useful in practice in the non-modular setting, is indeed feasible and very useful. Driven by our experience in the design and implementation of analysis and specialization techniques in the context of CiaoPP, the Ciao system preprocessor, in this paper we discuss a number of design goals for context-sensitive analysis of modular programs as well as the problems which arise in trying to meet these goals. We also provide a high-level description of a framework for analysis of modular programs which does substantially meet these objectives. This framework is generic in that it can be instantiated in different ways in order to adapt to different contexts. Finally, the behavior of the different instantiations w.r.t. the design goals that motivate our work is also discussed.

2 Introduction

Analysis of logic programs has received considerable theoretical and practical attention. A number of successful compile-time techniques have been proposed and implemented which allow obtaining useful information on the program and using such information to debug, validate, and specialize the program, obtaining important improvements in correctness and efficiency. Unfortunately, most of the existing techniques are still only used in prototypes and, though numerous experiments demonstrate their effectiveness, they have not made their way into existing real-life systems. Perhaps one of the reasons for this is that most of these techniques were originally designed to be applied to a complete, monolithic program, while programs in practice invariably have a more complex structure combining a number of user modules with system libraries. Clearly, organizing program code in this modular way has many practical advantages for both program development and maintenance. On the other hand, performing global techniques such as program analysis on modular programs differs from doing so in a monolithic setting in several interesting ways and poses non-trivial problems which must be solved.

In this work we concentrate on *strict* module systems in which procedures external to a module are *visible* to it only if they are part of its *interface*. The interface of a module usually contains the names of the *exported* procedures and the names of the procedures *imported* from other modules. The module can only import procedures which are among the ones exported by the other modules. Procedures which are not exported are not visible outside the module.

Driven by our experience in the design and implementation of context-sensitive analysis and specialization techniques in the CiaoPP system [PH03, HPBLG03a], in this paper we present a high level description of a framework for analysis of modular programs. This framework is generic in that it can be instantiated in different ways in order to adapt to different contexts. The correctness, accuracy, and efficiency of the different instantiations is discussed and compared.

The analysis of modular programs has been addressed in a number of previous works. However, most of them have focused on specific analyses with particular properties and using more or less ad-hoc techniques. In [CDG93] a framework is proposed for performing compositional analysis of logic programs in a modular fashion, using the concept of an *open program*, introduced in [BGLM94a]. An open program is a program in which part of the code is not available to the analyzer. Nevertheless, this interesting framework is valid only for a particular set of abstract domains of analysis—those which are *compositional*.

Another interesting framework for compositional analysis for logic programs is presented in [VB00], in this case, for *binding-time analysis*. Although the most natural way to describe abstract interpretation-based binding-time analyses is arguably to use a top-down, goal-dependent framework, in this work a goal-independent analysis framework is used in order to simplify the handling of the issues stemming from modularity. The choice is based on the fact that context-sensitivity brings important problems to a top-down analysis framework. Both this paper and [CDG93] stress compositionality as a very attractive property, since it greatly facilitates modular analysis. However, there are many useful abstract domains which do not meet this property, and thus these approaches are not of general applicability.

In [Pro02] a control-flow analysis-based technique is proposed for call graph construction in the context of object oriented languages. Although there has been other work in this area, the novelty of this approach w.r.t. previous proposals is that it is context-sensitive. Also, [BJ03] shows a way to perform modular class analysis by translating the object oriented program into *open* DATALOG programs, in the sense of [BGLM94a]. These two contributions are tailored to specific analysis domains with particular properties, so an important part of their work is not generally applicable nor reusable in a general framework.

In [RRL99] a two-phase analysis is proposed for incomplete imperative programs, starting with a fast, imprecise global analysis and then continuing with a (possibly context sensitive) analysis for each module in the program. This approach is not abstract interpretation-based. It is interesting to see that it appears to follow from the theory of abstract interpretation that if in such a two-pass approach the first pass "overshoots" the fixed-point, the maximum precision may not be recovered in the second pass.

In [TJ94] a method for performing separate control-flow analysis by means of abstract interpretation is proposed. This paper does not deal with the inter-modular approach studied in the present work, although it does have points in common with our module-aware analysis framework (Section 6). However, in this work the initial information needed by the abstract interpretation-based analyzer is provided by other analysis techniques (types and effects techniques), instead of taking advantage of the actual results from the analysis of the rest of the modules in the program.

A preliminary study of the extension of analysis and specialization to the case of modular programs was presented in [PH00]. A full practical proposal for modular program analysis was presented in [BdlBH$^+$01], which also presented some preliminary data from its implementation in the context of the Ciao system. Also, an implementation of [BdlBH$^+$01] in the context of the HAL system [GDMS02] has been reported in [Net02].

The rest of the paper proceeds as follows: Section 3 presents a review of program analysis based on abstract interpretation and of the non-modular framework that we use as a starting point. Section 4 then presents some additional notation related to modular programs and a first, simple approach to extending the framework to handling such modular programs: the "flattening" approach. This approach is used as baseline for comparison throughout the rest of the paper. Section 5 then identifies a number of characteristics that are desirable of a modular analysis system and which the simple approach does not meet in general. Achieving (at least a subset of) these characteristics justifies the more involved approach presented in the rest of the paper. To this end, Section 6 first discusses the modifications made to the analysis framework for non-modular programs in order to be able to handle one module at a time. Section 7 then presents the actual full framework for analysis of modular programs. The framework proposed is parametric on the *scheduling policies*. The following sections discuss two scheduling policies which are fundamentally different: *manual scheduling* (Section 8), which corresponds to a scenario where one or more users decide when and what modules to analyze individually (but in a context-sensitive way), such as in distributed program development, and *automatic scheduling* (Section 9), where a full scheduling policy automatically determines in which order the modules will be analyzed and continues until the process is completed (a fixed-point is reached). Section 10 addresses some practical implementation issues, including persistence and handling of libraries. Finally, Section 11 compares the behavior of the different instantiations of the generic framework proposed together with that of the flattening approach w.r.t. the desirable design features discussed in Section 5, and presents some conclusions.

3 A Non-Modular Context-Sensitive Analysis Framework

The aim of context-sensitive program analysis is, for a particular description domain, to take a program and a set of initial call patterns and to annotate the program with information about the current environment at each program point whenever that point is reached when executing calls described by the initial call patterns.

3.1 Program Analysis by Abstract Interpretation

Abstract interpretation [CC77] is a technique for static program analysis in which execution of the program is simulated on a description (or abstract) domain (D_α) which is simpler than the actual (or concrete) domain (D). Values in the description domain and sets of values in the actual domain are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : 2^D \rightarrow D_\alpha$ and *concretization* $\gamma : D_\alpha \rightarrow 2^D$ which form a Galois connection, i.e.

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

The set of all possible descriptions represents a description domain D_α which is usually a complete lattice or cpo for which all ascending chains are finite. 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. A description $\lambda \in D_\alpha$ *approximates* a set of concrete values $x \in 2^D$ if $\alpha(x) \sqsubseteq \lambda$. Correctness of abstract interpretation guarantees that the descriptions computed approximate all of the actual values which occur during execution of the program.

Different description domains may be used which capture different properties with different accuracy and cost. Also, for a given description domain, program, and set of initial call patterns there may be many different analysis graphs. However, for a given set of initial call patterns, a program and abstract operations on the descriptions, there is a unique *least analysis graph* which gives the most precise information possible.

3.2 The Generic Non-Modular Analysis Framework

We will now briefly describe the main ingredients of a generic context-sensitive analysis framework which computes the least analysis graph. This framework generalizes the particular analysis algorithms used in systems such as PLAI [MH90, MH92], GAIA [CV94], and the CLP($\mathcal{R}$) analyzer [KMM$^+$98], and we believe captures the essence of most context-sensitive, non-modular analysis systems. More details on this generic framework can be found in [HPMS00, PH96].

We first introduce some notation. *CD* and *AD* stand for descriptions in the abstract domain. The expression P : *CD* denotes a *call pattern*. This consists of a predicate call together with a call description for that predicate call. Similarly, P : *AD* denotes an answer pattern, though it will be referred to as *AD* when it is associated to a call pattern P : *CD* for the same predicate call.

The least analysis graph for the program is implicitly represented in the algorithm by means of two data structures, the *answer table* and the *dependency table*. Given the information in these data structures it is straightforward to construct the graph and the associated program point annotations. The answer table contains entries of the form P : *CD* $\mapsto$ *AD*. It is interpreted as: the answer pattern for calls of the form *CD* to P is *AD*. A dependency is of the form P : $CD_0 \Rightarrow B_{key} : CD_1$. This is interpreted as follows: if the procedure P is called with description CD_0 then this causes the procedure B to be called with description CD_1. The subindex *key* can be used in order to uniquely identify the program point within P where B is called with calling pattern CD_1. Dependency arcs represent the arcs in the program analysis graph from procedure calls to the corresponding call pattern.

Intuitively, different analysis algorithms correspond to different graph traversal strategies which place entries in the answer table and dependency table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixed-point algorithms, we use a priority queue. The queue contains the events to process. Different priority strategies correspond to different analysis algorithms. Thus, the third, and final, structure used in our generic framework is a *tasks queue*.

When an event being added to the tasks queue is already in the queue, a single event with the maximum of the priorities is kept in the queue. Also, only one arc of the form P : *CD* $\Rightarrow B_{key}$: *CD*$'$ for each tuple (P, CD, B_{key}) exists in the dependency table: the last one added. The same holds for entries P : *CD* $\mapsto$ *AD* for each tuple (P, CD) in the answer table.

Figure 1 shows the architecture of the framework. The *Code* corresponds to the (source)

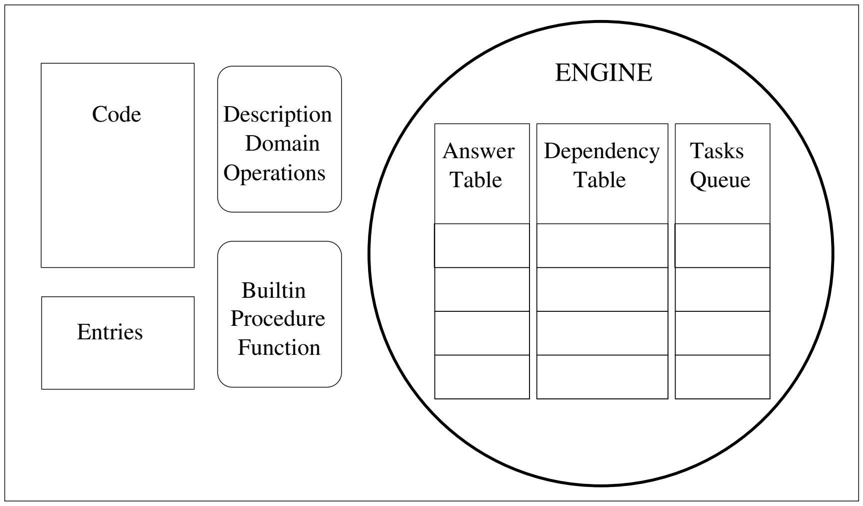

Figure 1: Non-Modular Analysis Framework

code of the program to be analyzed. By *Entries* we denote the initial starting points for analysis. The box *Description Domain Operations* represents the definition of operations which are domain dependent. The circle represents the *Analysis Engine*, which has the three data-structures mentioned above, i.e., the answer table, the dependency table, and the tasks queue. Initially, for each analysis these three structures are empty and the analysis engine takes care of processing the events on the priority queue by repeatedly removing the highest priority event and calling the appropriate event-handling function. This in turn consults and modifies the contents of the answer and dependency tables. When the tasks queue becomes empty then the analysis engine has reached a fixed-point. This implies that the least analysis graph has been found. We will use $Analysis_{D_\alpha}(Q, E) = (AT, DT)$ to denote that the analysis of program Q for initial descriptions E in domain D_α produces the answer table AT with dependency table DT.

3.3 Predefined Procedures

In order to simplify their presentation, formalizations of program analysis often do not consider *predefined* procedures. However, in practice, program analysis implementations allow the use of predefined (language built-in and/or library) procedures[1] in the programs to be analyzed. These *external* procedures whose code is not available in the program being analyzed are often handled in an *ad-hoc* way. Thus, in fairness, non-modular program analyses are more accurately represented by adding to the framework a *builtin procedure function* which essentially hardwires

[1]In our modular design, a library can be treated simply as (yet another) module in the program. However, special practical considerations for them will be discussed in Section 10.3.

the answer table for these external procedures. This function is represented in Figure 1 by the box *builtin procedure function*. We will use $\mathcal{CP}$ and $\mathcal{AP}$ to denote, respectively, the set of all call patterns and the set of all answer patterns. The builtin procedure function can be formalized as a function $BF : \mathcal{CP} \rightarrow \mathcal{AP}$. For all call pattern $P : \mathit{CD}$ where P is a builtin procedure $BF(P : \mathit{CD})$ returns a description *AD* which is assumed to be correct in the sense that it is a safe approximation, i.e. an over-approximation of the actual answer pattern for $P : \mathit{CD}$.

It is important to note that the data structures which are outside the analysis engine, *code*, *entries*, *description domain operations*, and *builtin procedure function* are read-only. However, though the code and entries are supposed to change for the analysis of each particular program, the *builtin procedure function* can be considered to be fixed, for each description domain D_α, in that it does not vary from the analysis of one program to another. Indeed, it can be considered to be part of the analyzer. Thus, the builtin procedure function is not explicitly represented as an input to the analysis algorithm.

4 The Flattening Approach to Modular Processing

We start by introducing some notation. We will use m and m' to denote *modules*. Given a module m, by $imports(m)$ we denote the set of modules which m imports. Figure 2 presents a modular program. Modules are represented as boxes and there is an arrow from m to m' iff m imports m'. In our example, $imports(a) = \{b, c\}$. By *depends*(m) we refer to the set generated by the transitive closure of $imports$, i.e. *depends*(m) is the least set such that $imports(m) \subseteq$ *depends*(m) and $m' \in$ *depends*(m) implies that $imports(m') \subseteq$ *depends*(m). In our example, $depends(a) = \{b, c, d, e, f\}$. Note that there may be circular dependencies among modules. In our example, $e \in depends(d)$ and $d \in depends(e)$. A module m is a *leaf* if $depends(m) = \emptyset$. In our example, the only leaf module is f. By *callers*(m) we denote the set of modules which import m. In the example, *callers*$(e) = \{b, c, d\}$. Also, we define *related*$(m) =$ *callers*$(m) \cup$ *imports*(m). In our example, *related*$(b) = \{a, d, e\}$.

The *program unit* of a given module m is the finite set of modules containing m and the modules on which m depends: *program_unit*$(m) = \{m\} \cup depends(m)$. m is called the *top-level* module of its program unit. In our example, *program_unit*$(a) = \{a, b, c, d, e, f\}$ and *program_unit*$(c) = \{c, d, e, f\}$. A program unit U is self-contained in the sense that $\forall\ m \in U\ : m' \in imported(m) \rightarrow m' \in U$.

Several *compilation tasks* such as program analysis and specialization are traditionally considered *global*, as opposed to *local*. Usually, local tasks process one procedure at a time and all the information required for performing the task can be obtained by inspecting that procedure.

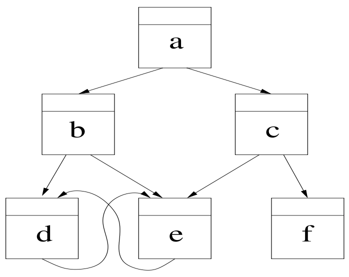

Figure 2: An Example of Module Dependencies

In contrast, in global tasks the results of processing a part of the program (say, a procedure) may be needed in order to process other parts of the program. Thus, global processing often requires iterating on the whole program until a fixed-point is reached.

In a modular setting, it may well be the case that part of the information needed to perform the task on (a procedure in) module m has to be computed in modules other than m. We will refer to the information originated in modules different from m as *inter-modular* information in contrast to the information originated in m itself, which we will call *intra-modular*.

Example 4.1 *In context-sensitive program analysis there is an information flow of both call and success patterns to and from procedures in different modules. Thus, program analysis requires inter-modular information. For example, the module c receives call patterns from module a since* $\text{callers}(c) = \{a\}$*, and it has to propagate the corresponding success patterns to a. In turn, c provides* $\{e, f\} = \text{imports}(c)$ *with call patterns and receives success patterns from them.*

4.1 Flattening a Program Unit vs. Modular Processing

Applying a framework for non-modular programs to a module m has the difficulty that m may not be self-contained. However, there should be no problem in applying the framework if m is a leaf module. Furthermore, given a global process such as program analysis, at least in principle, it is not obvious that it makes much sense to apply the process to a module m alone. In principle, it makes more sense to apply it to program units since they are conceptually self-contained. Thus, given a module m one natural approach seems to be to apply the tool (simultaneously) to all the modules in $U = \textit{program\_unit}(m)$.

Given a program unit U it is always possible to build a single module m_{flat} which is equivalent to U and which is a leaf. The process of constructing such a module m_{flat} usually only amounts to renaming apart identifiers in the different modules in U so as to avoid name clashes. We will use $\textit{flatten}(U) = m_{flat}$ to denote that the module m_{flat} is the result of renaming apart the code in each module in U and concatenating its code into a monolithic module m_{flat}. This points to a simple solution to the problem of processing modular programs (at least for the case in which all the code is available): to transform $program\_unit(m)$ into the equivalent monolithic program m_{flat}. It is then straightforward to apply any tool for non-modular programs to the leaf module m_{flat}. Figure 3 represents the case in which the non-modular analysis framework is used on the flattened program.

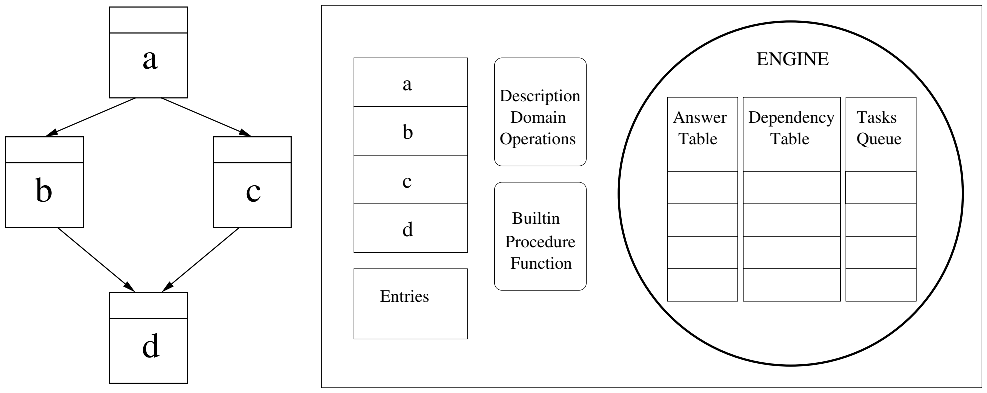

Figure 3: Using non-modular analysis on a flattened program

Given the existence of an implementation for non-modular analysis, this approach is often simple to apply. Also, this flattening approach has theoretical interest. It can be used, for example, in order to compare the efficiency of different approaches to modular handling of programs w.r.t. the flattening approach. However, as a practical way in which to actually perform analysis of program units this approach has important drawbacks. This issue will be discussed in more detail in Section 11.

5 Design Goals for Analysis of Modular Programs

Before presenting our proposals for analysis of modular programs, we will discuss the main features which should be taken into account when designing and/or implementing a tool for context-sensitive analysis of modular programs. As often happens in practice, some of the features presented are conflicting with others and this might make it impossible to find a framework which behaves optimally w.r.t. all of them.

Module-Awareness We consider a framework *module-aware* when it has been designed with modules in mind. Thus, it is applicable to a module m by using the code of m and some "interface" information for the modules in *imports*(m). Such interface information will in general consist of a summary of previous analysis results for such modules, if such results are available, or a safe approximation if they are not.

Though transforming a non-modular framework into a module-aware one may seem trivial, it requires identifying precisely which is the required information on the result of applying the tool in each of the modules in *imports*(m) which should be stored in order to apply the tool to m. This corresponds in general to the inter-modular information. It is also desirable that the amount of such information be minimal.

Example 5.1 *The framework for non-modular analysis in Section 3 is indeed non-modular since it requires the code of all procedures (except possibly for some predefined ones) to be available to the analyzer. It will produce wrong results when applied to non-leaf modules since a missing procedure can only be deemed as an error, unless the framework is aware that such a procedure can be imported.*

Correctness The results of applying the tool to a module m should produce results which are *correct*. The notion of correctness itself can in general be lifted from the non-modular case to the modular case without great difficulties. A more complex issue is how to extend a framework to the modular case in such a way that correctness is preserved.

Accuracy Similarly, the analysis results for a module m should be as accurate as possible. The notion of accuracy can be defined by comparing the analysis results with those which would be obtained using the flattening approach presented in Section 4.1 above, since the latter always computes the most accurate information possible, which corresponds to the least analysis graph.

Termination A framework for analysis of modular programs should guarantee termination (at least) in all cases in which the flattening approach terminates (which, typically, is for every program). Such termination is guaranteed by choosing description domains with some specific characteristics such as having finite height, finite ascending chains, etc., and/or incorporating a *widening operator*.

Efficiency in Time The time required to apply the tool should be reasonable. We will understand "reasonable" as not over an acceptable threshold on the time taken using the flattening approach.

Efficiency in Memory In general, one of the main expected advantages of the modular approach is that the total amount of memory required to handle each module separately should be smaller than that needed in the flattening approach.

No Need for Analyzing All Call Patterns Under certain circumstances, applying a tool on a module m may require processing only a subset of the call patterns rather than all call patterns for m. In order to achieve this, the model must keep track of fine-grained dependencies. This will allow marking exactly those call patterns which need processing. Other call patterns not marked do not need to be processed.

Support for the Co-Existence of Multiple Program Units/Applications In a modular setting it is often the case that a particular module is used in several applications. Support for software reuse is thus a desirable feature. However, this poses additional and interesting challenges to the tools, some of which will be discussed in Section 10.

Support for Source Changes What happens if the source of a module changes during processing? Some tools will not allow this at all and if it happens all the processing has to start again from scratch. This has the disadvantage that the tool is then not incremental since a (possibly minor) change in a module invalidates the information for all the program unit. Other tools may delete the information which may depend on the changed code, but still keep the information which does not depend on it.

Persistence This feature indicates that the inter-modular information can be stored in a persistent medium, such as a file stored on disk or a database, and allow later recovery of such information.

6 Analysis of Modular Programs: The Local Level

As a first step towards introducing our analysis framework for modular programs, which will be presented in Section 7 below, in this section we discuss the main ingredients which have to be added to an analysis framework for non-modular programs in order to be able to handle one module at a time.

Analyzing a module separately presents the difficulty that, from the point of view of analysis, the code to be analyzed is *incomplete* in the sense that the code for procedures imported from other modules is not available to analysis. More precisely, during analysis of a module m there

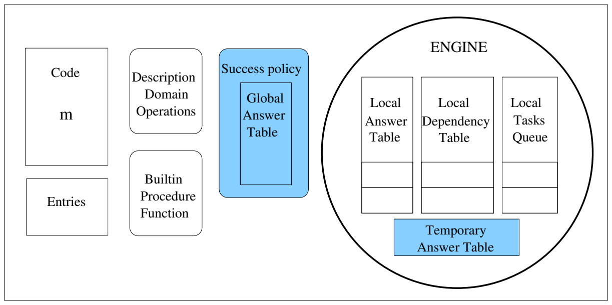

Figure 4: Module-aware analysis framework

may be calls $P : CD$ such that the procedure P is not defined in m but instead it is imported from another module $m' \in \mathit{imports}(m)$. We refer to determining the value of *AD* to be used for $P : CD \mapsto AD$ as the *imported success problem*. In addition, in order to obtain analysis information for m' which is as accurate as possible we need to somehow propagate the call $P : CD$ to m' so that the next time m' is analyzed such a call pattern is taken into account. We refer to this as the *imported calls problem*. Note that in this case analysis has to be module-aware in order to determine whether a given procedure is either local or imported (or predefined).

Figure 4 shows the architecture of an analysis framework which is module-aware. This framework is an extension of the non-modular framework in Figure 1. One minor change is that the read/write data structures internal to the analysis engine have been renamed with the prefix "local". So now we have the *local answer table*, the *local dependency table*, and the *local task queue*. Also, the box which represents the code now contains m indicating that it contains the single module m.

The shaded boxes in Figure 4 indicate the main differences w.r.t. the non-modular framework. One is that in the module-aware framework there is an additional read-only[2] data structure, the *global answer table*, or *GAT* for short. Its contents are identical in format to those in the answer table of the non-modular framework. There are however some differences: (1) the *GAT* contains analysis results which were obtained previously to the current analysis step. (2) The *GAT* contains entries which correspond to predicates defined in *imports*(m), whereas all entries in the local answer table (or *LAT* for short) are for predicates defined in m itself. (3) Only information of exported predicates is available in *GAT*. The *LAT* contains information for all predicates in m regardless of whether they are exported or not.

[2]In fact, this data structure is read/write at the global level discussed in Section 7 below, but it is read-only as regards our engine for analysis of one module.

6.1 Solving the Imported Success Problem

The second important difference is that the module-aware framework requires the use of a *success policy*, or *SP* for short, which is represented in Figure 4 with a shaded box surrounding the *GAT*. The *SP* can be seen as an intermediator between the *GAT* and the analysis engine. The behavior of the analysis engine for predicates defined in m remains exactly as before. *SP* is needed because though the information in the *GAT* will be used in order to obtain answer patterns for imported predicates, given a call pattern $P : CD$ it will often be the case that an entry of exactly the form $P : CD \mapsto AD$ does not exist in *GAT*. In such case, the information already present in *GAT* may be of value in order to obtain a (temporary) answer pattern *AD*. Note that the *GAT* together with *SP* will allow solving the "imported success problem".

In contrast, in many formalizations of non-modular analysis there is no explicit success policy. This is because if the call pattern $P : CD$ has not been analyzed yet, the analysis algorithm forces its computation. Thus, the results of analysis do not depend on any particular success policy: when analysis reaches a fixed-point there is always an entry of the form $P : CD \mapsto AD$ for any call pattern $P : CD$ which appears in the analysis graph. Unfortunately, in a modular setting it is not directly possible to force the analysis of predicates defined in other modules. Those modules may have already been analyzed or they may be analyzed in the future. We will simply do what we can given the information available in *GAT*.

We will use $\mathcal{GAT}$ to denote the set of all global answer tables. The success policy can be formalized as a function $SP : \mathcal{CP} \times \mathcal{GAT} \rightarrow \mathcal{AP}$. Several success policies can be defined which provide over- or under-approximations of the exact answer pattern $AD^{=}$ with different degree of accuracy. Note that this exact value $AD^{=}$ is the one which the flattening approach would compute. In this work we consider two kinds of success policies, those which are guaranteed to always provide over-approximations, i.e. $AD^{=} \sqsubseteq SP(P : CD, AT)$, and those which provide under-approximations, i.e., $SP(P : CD, AT) \sqsubseteq AD^{=}$. We will use the superscript $^{+}$ (resp $^{-}$) to indicate that a success policy over-approximates (resp. under-approximates). As will be discussed later in the paper, both over- and under-approximations are useful in different contexts and for different purposes. Since it is always required to know whether a success policy over- or under-approximates we will mark all success policies in either way.

Example 6.1 *A very precise over-approximating success policy is the function SP^{+}_{All} defined below, already proposed in [PH00]:*

$$\begin{aligned} SP^{+}_{All}(P : CD, GAT) &= topmost(CD) \sqcap_{AD' \in app} AD' \textit{ where} \\ app &= \{AD' \mid (P : CD' \mapsto AD') \in GAT \textit{ and } CD \sqsubseteq CD'\} \end{aligned}$$

The function $topmost$ *obtains the topmost answer pattern for a call pattern. The notion of* topmost description *was already introduced in [BCHP96]. Informally, a topmost description keeps those properties which are* downwards closed *whereas it loses those ones which are not. Note that taking* $\top$ *as answer pattern is a correct over-approximation, but often less accurate than using topmost substitutions. For example, if a variable is known to be ground in the call pattern, it will continue being ground in the answer pattern and taking* $\top$ *as the answer pattern 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.*

We refer to this success policy as SP^{+}_{All} *because it uses* all *entries in* GAT *which are* applicable *to the call pattern in the sense that the call pattern already computed is more general than the call being analyzed.*

Example 6.2 *The counter-part of* SP^{+}_{All} *is the function* SP^{-}_{All} *which is defined as:*

$$SP^{-}_{All}(P : CD, GAT) = \sqcup_{AD' \in app} AD' \text{ where}$$
$$app = \{AD' \mid (P : CD' \mapsto AD') \in \ GAT \text{ and } CD' \sqsubseteq CD\}$$

Note the change in the direction of the applicability relation (the call pattern in the GAT *has to be more particular than the one being analyzed) and the use of the lub operator instead of the glb. Also, note that taking, for example,* $\bot$ *as an under-approximation is correct but* SP^{-}_{All} *is more precise.*

6.2 Solving the Imported Calls Problem

The third important difference w.r.t. the non-modular framework is the use of the *temporary answer table* (or *TAT* for short) and which is represented as a shaded box within the analysis engine of Figure 4. This answer table will be used to store call patterns for imported predicates which are not yet present in *GAT* and whose answer pattern has been obtained (approximated) using the success policy on the entries currently stored in *GAT*. The *TAT* is used as a cache for imported call patterns and their corresponding answer patterns, thus avoiding having to repeatedly apply the success policy on the *GAT* for equivalent call patterns, which is an expensive operation. Also, after analysis of the current module is finished, the existence of the *TAT* simplifies the way in which the global data structures need to be updated. This will be discussed in more detail in Section 7 below.

We use $MAnalysis_{D_\alpha}(m, E_m, SP, GAT) = (LAT_m, LDT_m, TAT_M)$ to denote that the module-aware analysis framework returns (LAT_m, LDT_m, TAT_M) when applied to module m for initial call patterns E_m with SP and *GAT*.

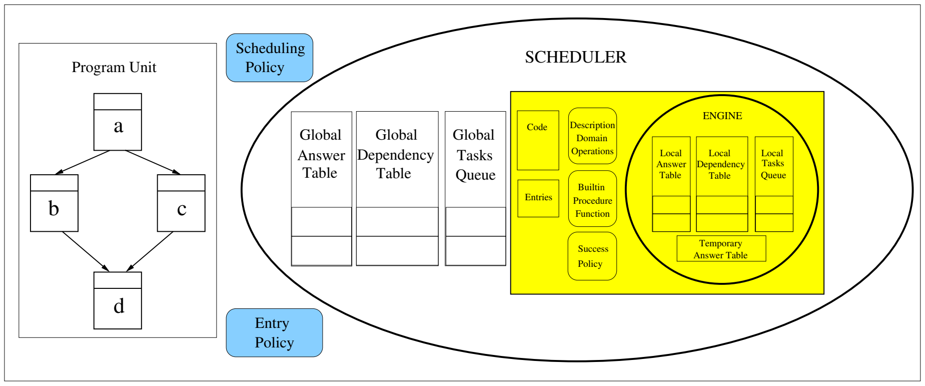

Figure 5: A two-level framework for analysis of modular programs

7 Analysis of Modular Programs: The Global Level

After discussing the *local-level* issues which appear when analyzing a module, in this section we present a complete framework for the analysis of modular programs. Since analysis is a global task, an analysis framework should not only deal with local-level information, but also with global-level information. A graphical representation of our framework is depicted in Figure 5. The main idea is that we have to add a higher-level component to the framework which takes care of the *inter-modular* information, as opposed to the *intra-modular* information which is handled by the local-level subsystem described in the previous section.

As a result, analysis of modular programs is best seen as a two-level process. Note that the inner, lightly shaded, rectangle corresponds exactly to Figure 4 as it is a module-aware analysis system. It is interesting to see how the data structures in the global and local levels are indeed very similar. The similarities and differences between the *GAT* and *LAT* have been discussed already in Section 6 above. Regarding the global and local dependency tables (*GDT* and *LDT* respectively), they are used in order to be able to propagate as precisely as possible which parts of the analysis graph have to be recomputed. The *GDT* is used in order to add events to the global task queue (*GTQ*) whereas the *LDT* is used to add events (*arcs*) to be (re-)analyzed to the local task queue (*LTQ*). We can define the events to be processed at the global level using different levels of granularity. As usual, the finer-grained these events are, the more detailed and thus more effective the handling of the events can be. One obvious possibility is to use modules as events. This means that all call patterns which correspond to a module are handled simultaneously whenever the module is selected at the global level. A more refined possibility is to keep events at the call pattern level. This, together with sufficiently detailed information in the *GDT* will allow incrementality at the call pattern level rather than module level.

7.1 Parameters of the Framework

The framework has three parameters. The *program unit* corresponds to the program unit to be analyzed. Note that the code may not be physically stored in the tool's memory since it is already on external storage. However, the framework may maintain some information on the program unit, such as dependencies among modules, *strongly connected components*, and any other information which may be useful in order to guide analysis. In the figure the *program unit* is represented, as an example, containing a program unit composed of four modules. The second parameter is the *entry policy*, which determines the way in which the *GTQ* and *GAT* should be initialized whenever analysis of a program unit is started. Depending on how the success policy is defined, entries for all procedures exported in each of the modules in the program unit may be required in *GAT* and *GTQ* or not.

Finally, the *scheduling policy* determines the order in which the entries in the *GTQ* should be processed. The efficiency with which the fixed-point is reached can differ very much from some scheduling policies to others. Since the framework presented in Figure 5 has just one analysis engine, processing a call pattern in a different module from that currently loaded has a relevant cost associated to it, since this often requires context switching from the current module to a new module. Thus, it is often a good idea to process all or many of the call patterns in *GTQ* which correspond to the module which is being analyzed in order to minimize the number of times the analysis tool has to switch from one module to another. In the rest of the paper we consider that events in *GTQ* are answer patterns which would benefit from (re-)analysis. The role of the scheduling policy is to select a set of patterns from *GTQ* which must necessarily belong to the same module m to be analyzed. Note that a scheduling policy based on modules can always be obtained by simply processing at each analysis step all events in *GTQ* which correspond to m.

7.2 How the Global Level Works

As already mentioned, analysis of a modular program starts by initializing the global data structures as indicated by the entry policy. At each step, the scheduling policy is used to determine the set E_m of entries for module m which are to be processed. They are removed from *GTQ* and copied into the data structure *Entries*. The code of the module m is also copied to *code*. Then, $\mathit{MAnalysis}(m, E_m, SP) = (\mathit{LAT}_m, \mathit{LDT}_m, \mathit{TAT}_m)$ is computed. Then, the global data structures are updated, as detailed in Section 7.3 below. As a result of this, new events may be added to *GTQ*. Analysis terminates when there are no more events to process in *GTQ* or when the scheduling strategy does not select any further events.

Each entry in *GTQ* is of one of the following three types: *over-approximation*, *under-approximation*, or *invalid*, according to the reason why they should be re-analyzed. An entry $P : CP \mapsto AP$ which is an over-approximation is marked $P : CP \mapsto^+ AP$. This indicates that the answer pattern AP is possibly an over-approximation since it depends on a call pattern whose answer pattern has been determined to be an over-approximation. In other words, the accuracy of $P : CP \mapsto AP$ may be improved by re-analysis. Similarly, under-approximations are marked $P : CP \mapsto^- AP$ and they indicate that AP is probably an under-approximation since it depends on a call pattern whose success pattern has increased. As a result, the call pattern should be re-analyzed to guarantee correctness. Finally invalid entries are marked $P : CP \mapsto^\perp AP$. They indicate that the relation between the current answer pattern AP and one resulting from recomputing it for $P : CP$ is unpredictable. This often indicates that the source code of the module has changed in a way that the analysis results for some of the exported procedures are just incompatible with previous ones. Handling this kind of events is discussed in more detail in Section 7.4 below.

7.3 Updating the Global State

In Section 6 it has been presented how the local level subsystem, given a module m, can compute the corresponding *LAT*$_m$, *LDT*$_m$, and *TAT*$_m$. However, once analysis of module m is done, the analysis results of module m have to be used in order to update the global state prior to starting analysis of any other module.

We now briefly discuss how this updating is done. For each initial call pattern $P : CP$ in *Entries* we compare the previous answer pattern AP with the newly computed one AP'. If $AP = AP'$ then this call pattern has not been affected by the latest analysis. However, it is also possible that the answer pattern "evolves" in different analysis iterations. If we use SP^+, the natural thing is that the new answer pattern is more specific than the previous one, i.e., $AP' \sqsubset AP$. In such case those call patterns which depend on $P : CP$ can also improve their success pattern. We use the *GDT* to locate all such patterns and we add them to the *GTQ* with the $^+$ mark. Conversely, if we use SP^-, the natural thing is that $AP \sqsubset AP'$. We then add events marked $^-$.

In a typical situation, and if modules do not change, all events in *GTQ* will be approximations of the same sign. This depends on the success policy used. If the success policy is of kind SP^+ (resp. SP^-) then the events which will be added to *GTQ* will also be over-approximations (resp. under-approximations). In turn, when they are processed they will introduce other over-approximations (resp. under-approximations).

The *TAT*$_m$ is also used to update the global state. All entries in *TAT*$_m$ are added to *GAT* and *GTQ* marked with the same sign as the success policy used. Last, we also have to update the

GDT. For this, we first erase all entries for any of the call patterns which we have just analyzed, and which are thus stored in *entries*$_m$. Then we add an entry of the form $P : CP \to H : CP'$ for each imported procedure H which is reachable with call pattern CP' from an initial call pattern $P : CP$. Note that this can easily be determined using *LDT*.

7.4 Recovering from an Invalid State

If code of a module m has changed since it was last analyzed, it can be the case that the global information available is invalid. This happens when in the results of re-analysis of m any of the exported predicates has an answer pattern which is incompatible with the previous results. In this case, all information dependent on the new answer patterns might have become invalid, as discussed in Section 7.2. The question is how to minimize the impact of such a situation.

The simplest solution is to (transitively) erase any information of other modules which depends on the invalidated one. This solution may not be very efficient, as it ignores all results of previous analyses of other modules even if the changes performed in the module are minor, or only affect directly related modules. Another alternative is to launch an automatic recovery process as soon as invalid analysis results are detected (see [BdlBH$^+$01]). This process has to reanalyze the modules directly affected by the invalidated answer pattern(s). If the new answer patterns coincide with the old ones then the changes do not affect this module and the process terminates. Otherwise, it continues transitively with the directly related modules.

8 Using a Manual Scheduling Policy

Consider, for example, the relevant case of independent development of different parts of the program, which can then even be performed in parallel by different teams. In this setting, it makes sense that the analyzer performs its job on the current module without analyzing other modules in the program unit, i.e., it allows separate analysis. This will typically allow early detection of compile-time errors in the current module without having to wait for the code of the dependent modules to be fully developed. Moreover, in this setting, it is the user (or users) who decide when and what to analyze. Thus, we refer to this as the *manual* setting. Furthermore, we assume that in this setting analysis for a module m has to do its best with only the code for m plus the results of previous analyses (if any) of the modules in $depends(m)$. These assumptions have important implications. The setting allows the users of different modules to decide when they should be processed. And thus, any module could be (re-)analyzed at any point. As a result, strong requirements must hold for the whole approach to be correct. In return, the results obtained may not be optimal (in terms of error detection, degree of optimization, etc., depending on the particular tools) w.r.t. those achievable using automatic scheduling.

So the question is, is there any combination of the three parameters of the global analysis framework which allows handling the manual setting? The answer to this question is yes. Our earlier paper [BdlBH$^+$01] essentially describes such an instantiation of the analysis framework. In the terminology of the current paper, the model in [BdlBH$^+$01] corresponds to waiting until the user requests that a module m in the program unit U be analyzed. The success policy is over-approximating. This guarantees that in the absence of invalidated entries in the *GTQ* all events will be marked $^+$. This means that the analysis information available is correct, though perhaps not as accurate as possible. Since the scheduling is manual, no other analyses should be triggered until the user requires so. Finally, the entry policy is simply to include in *GTQ* an event such as $P : \top \mapsto^+ \top$ per predicate exported by any of the modules in *U* to be analyzed (it is called *all* entry policy). The initial events are required to be so general to keep the overall correctness of the analysis while allowing the users to choose the order of the modules to be analyzed.[3] The model in [BdlBH$^+$01] has the very important feature of being guaranteed to always provide correct results without the need of reaching a global fixed-point.

9 Using an Automatic Scheduling Policy

In spite of the evident interest of the manual setting, there are situations in which the user is interested in obtaining the most accurate analysis results possible. For this, it may be required to analyze the modules in the program unit several times in order to converge to a distributed global fixed-point. We will refer to this as the *automatic* setting, in which the user decides when to start global analysis of a program unit. From then on it is the global analysis framework by means of its *scheduling policy* who decides when and what to analyze. Note that the manual and automatic settings roughly correspond to scenario 1 and scenario 2 of [PH00] respectively. Since we admit circular dependencies among modules, the strategy has to be able to deal with such circularities correctly and efficiently without entering infinite loops. The question now is what are the values for the different parameters to our generic framework which should be used in order to obtain satisfactory results? One major difference of the automatic setting w.r.t. the manual setting is that in addition to over-approximations, now also under-approximations can be used. This is because though under-approximations do not guarantee correctness in general, when an inter-modular fixed-point is reached, analysis results are guaranteed to be correct. Below we consider the use

[3]In the case of the Ciao system it is possible to use *entry* declarations (see for example [PBH00a]) in order to improve the set of initial call patterns for analysis.

of SP^+ and SP^- separately.

9.1 Using Over-Approximating Success Policies

If a success policy SP^+ is used, we are in a situation similar to the one in Section 8 in that independently of how many times each module has been analyzed, if there have not been any code changes, the analysis results are guaranteed to be correct. The main difference is that now the system keeps on automatically requesting further analysis steps until a fixed-point is reached.

Regarding the entry policy, an important observation is that in the automatic mode, much as in the case of intra-modular analysis, inter-modular analysis will eventually compute all call patterns which are needed in order to obtain information which is correct w.r.t. calls, i.e., the set of computed call patterns covers all possible calls which may occur at run-time for the class of initial calls considered, i.e., those for the top-level of the program unit U. This will allow us to use a different entry policy from that used in the manual mode: rather than introducing events of the form $P : \top \mapsto^+ \top$ in the *GTQ* for exported predicates in all modules in U, it suffices to introduce them for predicates exported by the top-level of U (this entry policy is named *top-level* entry policy). This has several important advantages: (1) It avoids analyzing all predicates for the most general call pattern, since this may end up introducing plenty of call patterns which are not used in our particular program unit U. (2) It will help to have a more guided scheduling policy since there are no requests for processing a module until it is certain that a call pattern should be analyzed. (3) If multiple specialization is being performed based on the set of call patterns for each procedure (possibly proceeded by a minimization step for eliminating useless versions [PH99]), the fact that a call pattern with the most general call pattern exists implies that a non-optimized version of the predicate must always exist. Another way out of this problem is to eliminate useless call patterns once an inter-modular fixed-point has been reached.

Since reaching a global fixed-point can be a costly task, one interesting possibility can be the introduction of a time-out. The user can ask the system to request (re-)analysis as needed towards improving the analysis information. However, if after performing n analysis steps the time-out is reached before analysis $n + 1$ is finished, the global state corresponding to state n is guaranteed to be correct. In this case, the entry policy used has to be to introduce most general call patterns for all exported predicates, either before starting analysis or when a time-out is reached.

9.2 Using Under-Approximating Success Policies

Another alternative is to use SP^-. As a result, the analysis results are not guaranteed to be correct until an inter-modular fixed-point is reached. Thus, it may take a large amount of time to perform this global analysis. On the other hand, once a fixed-point is reached, the accuracy which will be obtained is optimal, since it corresponds to the least analysis graph, which is exactly the same which the flattening approach would have obtained.

Regarding the entry policy, the same discussion as above applies. The only difference being that the *GTQ* should be initialized with events of the form $P : \top \mapsto^{-} \bot$ since now the framework computes under-approximations. Clearly, $\bot$ is an under-approximation of any description.

Another important thing to note is that, since the final results of automatic analysis are optimal, they do not depend on the use of a particular success policy SP_1^- or another SP_2^-. Of course, the efficiency using SP_1^- can be very different from that obtained using SP_2^-.

9.3 Hybrid policy

In practice we may wish to use a manual scheduling policy with an over-approximating success policy during program development, and then use an automatic scheduling policy with an under-approximating success policy just before program release, so as to ensure that the analysis is as precise as possible, thus allowing as much optimization as possible in the final version.

Fortunately, in such a situation we can often reuse much of the analysis information obtained using the over-approximating success policy. The reason is that if the analysis with the over-approximating success policy has reached a fixed-point, the answers obtained for module m are as accurate as those obtained with an under-approximating success policy as long as there are no cyclic dependencies between the modules in $depends(m)$. Thus in the common case that no modules are mutually dependent we can simply use the answer tables from the manual scheduling policy and use an automatic scheduling policy with an over-approximating success policy to obtain the fixed-point. Even in the case that some modules are mutually dependent we can use this technique to compute the answers for the modules which do not contain cyclic dependencies or do not depend on modules that contain them (e.g., leaf-modules).

9.4 Computation of an Intermodular Fixed-Point

Determining the optimal order in which the different modules in the program unit should be analyzed in order to get to a fixed-point as efficiently as possible is not trivial and it is the topic of ongoing work.

Finding good scheduling strategies for intra-modular analysis is a topic which has received considerable attention and highly optimized algorithms exist which converge to a fixed-point quickly. Unfortunately, it is not possible to directly translate the same heuristics used in the intra-modular case to the inter-modular case. In the inter-modular case we have to take into account the time required to change from analysis of one module to another since this typically means reading a new module from disk. Thus, requests to process call patterns have to be grouped by modules in order to reduce the number of times we change context.

Taking the heuristics in [PH96, HPMS00] as a starting point we are investigating and experimenting with different scheduling policies which take into account different aspects of the structure of the program unit such as dependencies, strongly connected components, etc. with promising results. It also remains to be explored which of the approaches to success policy results in more efficiently reaching a global fixed-point and whether the heuristics to be applied in either case coincide or are mostly different.

10 Some Practical Implementation Issues

In this section we discuss several issues not addressed in the previous sections and which are very important in order to have practical implementations of context-sensitive analysis systems. These issues are related to the persistence of global information and the analysis of libraries.

10.1 Making Global Information Persistent

The two-level framework presented in Section 7 needs to keep information both at the local and global level. One relevant question, due to its practical implications, is where this global information actually resides. One possibility is to have the global analysis tool running continuously as a kind of "compilation server" which stores the global state in its program memory. In a *manual* setting, this global tool would wait for the user(s) to place requests to analyze modules. When a request is received, the corresponding module is analyzed for the appropriate call patterns and using the global information available at the time in the memory of the global analyzer. After analysis terminates, the global information is updated and remembered by the process for subsequent requests. If we are in an *automatic* setting, the global tool itself requests the analysis of different modules until a global fixed-point (or a time-out) is reached.

This approach outlined above is not fully persistent in the sense that if the computer crashes all information about the global state is lost and analysis would have to start from scratch again. In order to implement the more general kind of persistence discussed in Section 5, a way to save and restore the global state of analysis is needed. This requires storing the value of the three global-level data-structures: GAT, GDT, and GTQ. A level of granularity which seems appropriate in this context is clearly the module level. I.e., the global state of analysis is saved and restored between two consecutive steps of (module) analysis, but not during the analysis of a given module, which, from the point of view of the two-level framework, is an atomic operation.

The ability to save and restore the global state of analysis has several advantages:

1. The global tool does not need to be running continuously: it can save its state, stop, restart when needed, and restore the global state. This is specially interesting when using a manual scheduling policy, since two consecutive analysis requests can be separated by large intervals.
2. Even if the automatic scheduling policy is used, any information about the global state which is still valid can be directly used. This means that analysis can be *incremental* in the sense that (global level) analysis information which is known to be valid is reused.

10.2 Splitting Global Information

Consider the analysis of module b in the program unit $U = \{a, b, c, d, e, f, g, h\}$ depicted in Figure 6. In principle, the global state includes information regarding exported predicates in any of the modules in U. As a result, if we can save the global state to disk and restore it, this would involve storing and retrieving information about all modules in U. However, analysis of b only requires retrieving the information for modules in *related(m)*. The small boxes which appear on the side of every module represent the portion of the global structures related to each module. To analyze the module b, the information of the global tables that we need is that of modules a, d and e, as indicated by the dashed curved line.

This is straightforward to do in practice by splitting the information in the global data structures into several parts, each one associated to a module. This allows easily identifying the pieces of global information which are needed in order to process a given module.

This optimization of the handling of global information has several advantages:

1. The time required to save and restore the information to disk is reduced since the total amount of information transferred is smaller.
2. The use of the data structures during analysis can be more efficient since search space is reduced.
3. The total amount of memory required in order to analyze a module can be significantly reduced: only the local data structures plus a possibly very reduced part of the global data structures are actually required to analyze the module.

One question which we have intentionally left open is where the persistent information should reside. In fact, all the discussion above is independent on how and where the global state is stored, as long as it is persistent. One possibility is to use a database which stores the global state and information is grouped by modules in order to minimize the amount of information which has to be retrieved or updated for each analysis. Another, very common, possibility is to store the global information associated to each module to disk, in the same way as temporary information (such as relocatable code) is stored in many traditional compilers. In fact, the actual implementation of modular analysis in both CiaoPP and HAL [Net02] systems is based on this idea: a module m has a `m.reg` file associated to it which contains the part of the global data structures which are associated to m.

10.3 Handling Libraries and Predefined Modules

Many compilers and program development systems include a large number of predefined modules and libraries which can be readily reused by programmers –an obviously interesting feature since it greatly reduces the time required to develop applications. From the point of view of analysis, these predefined modules and libraries differ from user programs in a number of ways:

1. They are designed with reusability in mind and thus they can be used by a comparatively large number of user programs.

2. Sometimes the source code for libraries and predefined modules may not be available. One common reason for this is that they are implemented in a lower-level language.

3. The total amount of code available as libraries can be extremely large. Thus, reanalyzing the libraries over and over again for slightly different call patterns can be costly.

Given these characteristics, it makes sense to develop a specialized treatment for libraries. We propose the following scheme. For each library module, the analysis results for a sufficient set of call patterns should be precomputed. This set should cover all possible correct call patterns for the library. In addition, the answer pattern for those call patterns have to be an over-approximation of the actual answers, independently of whether a SP^+ or SP^- success policy is used for the programs which use such library. In addition, in order to provide more accurate information, more particular call patterns which are expected to occur often in programs which use that library module can also be included. This information is added to the *GAT* of the program units which use the library. Thus, the success policy will be able to use this information directly for obtaining answer patterns. The reason for requiring pre-computed answer patterns for library modules to be over-approximations is that, much in the same way as for predefined procedures, even if an automatic scheduling policy is used, library modules are (in principle) not

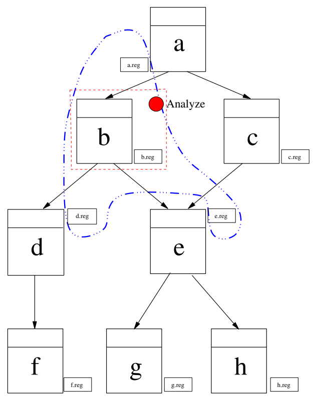

Figure 6: Using Distributed Scheduling and Local Data Structures

analyzed for calling patterns other than those which are pre-computed. Note that this is conceptually equivalent to considering the interface information of library modules *read-only*, since any program using them can read this information, but no additional call patterns will be analyzed. As a result, the global level framework will ignore new call patterns to library procedures that might be generated during the analysis of user programs. More precisely, entries of the form $P : CP \mapsto AP$ in *TAT* such that P is a library predicate do not need to be added to the *GTQ* since they will not be analyzed. In addition, no entries of the form $P : CP \rightarrow H : CP'$ need be added to *GDT* if H is a library predicate, since the answer pattern for library predicates is never modified and thus those dependencies are useless.

Deciding which is the best set of call patterns for which a library module should be analyzed is a non-trivial problem. One possibility can be to extract call patterns from correct programs which use the library and study which are the call patterns most often used. Another possibility is to have the library developer decide which are the call patterns of interest.

In spite of the considerations above, it is sometimes the case that we are interested in treating a library module using the general scheme, i.e., effectively considering the library information writable and allowing the analysis of new call patterns and the storage of the corresponding results. This can be interesting if the source code of a library is available and the set of initial call patterns for which it has been analyzed is not very representative. Note that hopefully this will happen often only when the library is relatively new. Once the code of the library stabilizes and a good set of initial patterns is obtained, it will generally be considered read-only. Allowing reanalysis of a library can also be useful when we are interested in using the analysis results from such call patterns to optimize the code of the library for the particular cases that correspond to those calls. For this case it may be interesting to store the corresponding information locally to the calling module, as opposed to inserting it into the library directories.

In summary, the implementation of the framework needs to treat libraries in a special way and also allow applying the general scheme for some designated library modules.

11 Discussion and Conclusions

Table 1 summarizes some characteristics of the different instantiations of the generic framework presented in the paper, in terms of the design features discussed in Section 5. The corresponding entries for the flattening approach of Section 4 –our baseline as usual– are also provided for comparison, listed in the column labeled Flattening. The Manual column lists the characteristics of the manual scheduling policy described in Section 8. The last two columns correspond to the two instantiations of the automatic scheduling policy, which were presented in Sections 9.1 and 9.2 respectively. Automatic^{+} (resp. Automatic^{-}) indicate that an over-approximating (resp. under-approximating) success policy is used.

The first three rows, i.e., Scheduling policy, Success policy, and Entry policy correspond to the values of these parameters in each instantiation.

All instances of the framework for modular analysis are *module-aware*, in contrast to Flattening, which is not. Both instances described of the modular framework proposed are incremental, in the sense that only a subset (instead of every module) in the program unit needs to be reanalyzed, and they also both achieve the goal of *not needing to reanalyze all call patterns* every time a module is considered for analysis.

Regarding correctness, both the Flattening and Automatic^{-} approaches have in common that correctness is only guaranteed when analysis comes to an end. This is because the approximations used are under-approximations and thus the results are only guaranteed to be correct when a (global) fixed-point is reached. However, in the Manual and Automatic^{+} approaches the information in the global state is correct after any number of local analysis steps.

On the other hand, both the Flattening and Automatic^{-} approaches are guaranteed to obtain the most accurate information possible, i.e., the least analysis graph, when a fixed-point is reached. In contrast, the Manual approach cannot guarantee optimal accuracy for two reasons. The first one is that there is no guarantee that modules will be processed the number of times that is necessary for an inter-modular fixed-point to be reached. Second, even if such a fixed-point is reached, it may not be the least fixed-point. This is because this approach uses over-approximations of the analysis results which are improved ("narrowed") in the different analysis iterations until a fixed-point is reached. On the other hand, if there are no circular dependencies among predicates in different modules, then the fixed-point obtained will be the least one, i.e., the most accurate.

Table 1: Comparison of Approaches to Modular Analysis

	Flattening	Manual	Automatic$^+$	Automatic$^-$
Scheduling policy	automatic	manual	automatic	automatic
Success policy	SP^-	SP^+	SP^+	SP^-
Entry policy	top-level	all	top-level	top-level
Module-aware	no	yes	yes	yes
No Rean. of all CPs	no	n/a	yes	yes
Correct	at fixed-point	yes	yes	at fixed-point
Accurate	yes	no	no circularities	yes
Efficient in time	yes	n/a	no	no
Efficient in memory	no	yes	yes	yes
Termination	finite asc. chains	finite asc. chains	finite chains	finite asc. chains

Regarding efficiency in time we will consider two cases. The first one is when we have to perform analysis of the program unit from scratch. In this case, Flattening can be highly optimized in order to converge quickly to a fixed-point. In contrast, in this situation the instances of the modular framework have the disadvantage that loading and unloading modules during analysis introduces a significant overhead. As a result, in order to maintain the number of context changes low, call patterns may be solicited from imported modules which use temporary information and which are not needed in the final analysis graph. These call patterns which end up being useless are known as *spurious* versions. This problem also occurs in Flattening, though to a much lesser degree if good algorithms are used. Therefore, the modular approaches may end up performing work which is speculative, and thus the total amount of work performed in the automatic approaches to modular analysis is in principle an upper bound of that needed in Flattening.

On the other hand, consider the second case in which a relatively large amount of intra-modular analysis has already taken place for the modules to be analyzed in our programming unit and that the global information is persistent. In this case, the automatic approaches can update their global data structures using the precomputed information, rather than starting from scratch as is done in Flattening. In such a case the automatic approaches may perform much less work than Flattening. It is to be expected that once module m becomes stable, i.e., it is fully developed, it will quickly be analyzed for a relatively large set of calling patterns. In such a case it is likely that it will be possible to analyze any other module m' which uses m by simply reusing the existing analysis results for m. This is specially true in the case of *library modules*, as discussed in Section 10.3.

Regarding the efficiency in terms of memory, it is to be expected that the instances of the modular framework will outperform the non-modular, flattening approach. This was in fact already observed in the case of [BdlBH$^+$01]. Indeed, one important practical difficulty that appears during the (monolithic) analysis of large programs is that the amount of information which is kept in memory is very large and the storage needed can become too large to fit in memory. The modular framework proposed needs less memory because: a) at each point in time, only one module requires to be loaded in the code area, and b) the local answer table only needs to hold entries for the module being analyzed, and not for other modules. Also, in general, the total amount of memory required to store the global data structures is not very high when compared to the memory required locally for the different modules. In addition, not all the global data structures are required when analyzing a module m, but only that associated with the modules in *related*(m).

Finally, regarding termination, except for Flattening, in which only one level of termination is required, the three other cases require two levels of termination: at the intra-modular and at the inter-modular level. In Flattening, since analysis results increase monotonically until a fixed-point is reached, termination is often guaranteed by considering description domains which do not contain infinite ascending chains: no matter what the current description is, top ($\top$), which is trivially guaranteed to be a fixed-point, is only a finite number of steps away. Exactly the same condition is required for guaranteeing termination of Automatic$^-$. The manual approach only requires guaranteeing intra-modular termination since the number of call patterns analyzed is finite. However, in the case Automatic$^+$, finite ascending chains are required for ensuring local termination *and* finite descending chains are required for ensuring global termination. As a result, termination requires domains with finite chains, or appropriate widening operators.

In summary, the proposed two-level generic framework for analysis and its instantiations meet a good subset of our stated objectives. We hope the discussion and the concrete proposal presented in this paper will provide a better understanding of the handling of context-sensitive program analysis on modular programs and contribute to the widespread use of such context-sensitive analysis techniques for modular programs in practical systems. An implementation of the framework, as a generalization of the pre-existing CiaoPP modular analysis components, is currently being completed. In this context, we are experimenting with different scheduling policies for the global level, for concrete, practical analysis situations.

Part II

Experiments in Context-Sensitive Analysis of Modular Programs

1 Summary

Several models for context-sensitive analysis of modular programs have been proposed, each with different characteristics and representing different tradeoffs. The advantage of these context-sensitive analyses is that they provide information which is potentially more accurate than that provided by context-free analyses. Such information can then be applied to validating/debugging the program and/or to specializing the program in order to obtain important performance improvements. Some very preliminary experimental results have also been reported for some of these models which provided initial evidence on their potential. However, further experimentation, which is needed in order to understand the many issues left open and to show that the proposed modes scale and are usable in the context of large, real-life modular programs, was left as future work. The aim of this paper is two-fold. On one hand we provide an empirical comparison of the different models proposed in previous work, as well as experimental data on the different choices left open in those designs. On the other hand we explore the scalability of these models by using larger modular programs as benchmarks. The results have been obtained from a realistic implementation of the models, integrated in a production-quality compiler (CiaoPP/Ciao). Our experimental results shed light on the practical implications of the different design choices and of the models themselves. We also show that context-sensitive analysis of modular programs is indeed feasible in practice, and that in certain critical cases it provides better performance results than those achievable by analyzing the whole program at once, specially in terms on memory consumption and when reanalyzing after making changes to a program, as is often the case during program development.

2 Introduction and Motivation

Global analysis of logic programs has received considerable theoretical and practical attention and as a result it is now possible to infer a wide range of program properties with a considerable degree of accuracy and for a significant number of programs. Also, tools have been developed which in addition to inferring these properties, allow debugging, validating, and specializing programs, achieving important improvements in both correctness and efficiency. However, most of these techniques were originally designed to be applied to a complete, monolithic program. In contrast, real programs invariably have a more complex structure combining a number of user modules with other modules from system libraries. This is one of the reasons why most global analysis tools are still prototypes and, though numerous experiments demonstrate their effectiveness, they have not yet made their way into existing real-life programming systems.

Performing global analysis on modular programs differs from doing so in a monolithic setting in several interesting ways and poses non-trivial problems which must be solved. A preliminary study of the extension of analysis and specialization to the case of modular programs was presented in [PH00]. A full practical proposal for context-sensitive analysis of modular programs was presented in [BdlBH$^+$01]. In fact, in [BdlBH$^+$01] a collection of models was proposed, each of them with different characteristics and representing different tradeoffs. Some very preliminary experimental data was also reported for an implementation of some of these models in the context of the Ciao system. Also, another implementation of [BdlBH$^+$01] in the context of the HAL system [GDMS02] was reported in [Net02]. This previous preliminary experimental results provided initial evidence on the overall potential of the approach. However, it was left as future work to perform further experimentation in order to understand the many issues left open and to show that the proposed modes scale and are usable in the context of large, real-life modular programs.

The aim of this paper is two-fold. On one hand we provide an empirical comparison of the different models proposed in [BdlBH$^+$01], as well as experimental data on the different choices left open in those designs. To this end we have completed a full implementation in CiaoPP of the framework for context-sensitive analysis described in [PCH$^+$04] and its different instances and we have studied experimentally the behaviour of the resulting system. These results have been compared with traditional, non modular analyses in several parameters.

Our second aim is to explore the scalability of these models and the implementation. To this end we have used some larger modular programs as benchmarks, including some real-life examples such as a working partial evaluator and parts of the Ciao compiler.

In the following Section we present an overview of the general problems in analyzing large modular programs, and the solutions proposed in previous work, including the major design tradeoffs. Section 4 then describes the tests performed and analyzes the results obtained. Finally, Section 5 presents our conclusions.

3 Analysis of modular programs

As mentioned in the previous section, the framework used herein is based in [PH00, PCH$^+$04], where a detailed description of the issues related to the analysis of modular programs and the different approaches to it can be found. The following subsections present an overall summary of [PCH$^+$04], with special emphasis on the issues that are most relevant to our experimental study.

3.1 Modular programs

A program is said to be modular when its source code is distributed in several source units named modules, and they contain language constructions to clearly define the interface of every module with the rest of the modules in the program. This interface is composed of two sets of predicates: the set of exported predicates (those accessible from other modules), and the set of imported predicates. For concreteness, and because of its appropriateness for global analysis, in our implementation we will use the module system of [CH00]. This module system is *strict* in the sense that procedures external to a module are visible to it only if they are part of its *interface*. A predicate defined in a given module can be called from another module only if it appears in the exported list of its module and in the imported list of the caller module, i.e., procedures which are not exported are not visible outside the module in which they are defined.

We note the distinction between *global* tasks and *local* tasks. In global tasks the results of processing a part of the program (say, a procedure or a module) may be needed in order to process other parts of the program. In contrast, a local task processes only one procedure or module at a time and, most importantly, all the information required for performing the task can be obtained by inspecting that procedure or module. The fundamental issue is that global processing often requires iterating on the whole program until a fixed-point is reached.

Context-sensitive program analysis is an example of a *global* task: in a modular setting, it may well be the case that part of the information needed to perform the analysis on (a procedure in) module m has to be computed in modules other than m. We will refer to the information originated in modules different from m as *inter-modular* information in contrast to the information originated in m itself, which we will call *intra-modular*.

3.2 Flattening a Program Unit vs. Modular Processing

Applying a framework for non-modular programs to a module m which belongs to a modular program has the difficulty that m may not be self-contained. However, there should be no problem in applying the framework if m is a leaf module. Furthermore, given a global process such as program analysis, at least in principle, it is not obvious that it makes much sense to apply the process to a module m alone. In fact, it makes sense to apply analysis to the complete program instead, since it is conceptually self-contained.

Given a modular program P it is always possible to build a single module m_{flat} which is equivalent to P and which is a leaf. The process of constructing such a module m_{flat} usually only amounts to renaming apart identifiers in the different modules in P so as to avoid name clashes. We will use *flatten*$(P) = m_{flat}$ to denote that the module m_{flat} is the result of renaming apart the code in each module in P and concatenating its code into a monolithic module m_{flat}. This points to a simple solution to the problem of processing modular programs (at least for the case in which all the code is available): to transform P into the equivalent monolithic program m_{flat}. It is then straightforward to apply any tool for non-modular programs to the leaf module m_{flat}. In the rest of this work, we will refer to this approach as the *flattened* or *monolithic* approach.

Assuming the existence of an implementation for non-modular analysis, this approach to analyzing modular programs is often simple to apply. Also, the flattening approach has theoretical interest: in our case it will be used to compare the efficiency of different approaches to modular handling of programs w.r.t. it. However, as a practical way in which to actually perform analysis of large program the flattening approach also has important potential drawbacks. The most important is that the complete program must be loaded into the analyzer, and thus large programs may make the analyzer run out of memory. Moreover, as the internal analysis data structures include information for all the program source code, in the monolithic case, analysis of a given procedure may take more time than keeping in memory only the module in which it resides. Another, perhaps more important drawback, is that the program must be self-contained: this can be a problem if the analyzer is used while developing the program, when some modules are not yet implemented, or if there are calls to external procedures, i.e., procedures for which the source code is not available, or which are implemented in other languages.[4]

3.3 Analyzing one module at a time

The approach taken in [PCH^{+}04] and implemented in CiaoPP is based on the separate analysis of the modules in a modular program. The analyzer is invoked (possibly several times) for each module in the program, in order to obtain the analysis results needed by the analysis of other program modules. We denote the process of obtaining the answer value *AD* of any predicate P

[4]Several approaches have been proposed for the analysis of incomplete programs (*open programs*), for example [BCHP96, BJ03].

for a call *CD* as: P : *CD* $\mapsto$ *AD*. The analysis results obtained for the exported predicates of every module are stored in a *Global Answer Table* (GAT).

Analyzing a module separately presents the difficulty that, from the point of view of analysis, the code to be analyzed is *incomplete* in the sense that the code for procedures imported from other modules is not available to analysis. More precisely, during the analysis of a module m there may be calls P : *CD* such that the procedure P is not defined in m but instead it is imported from another module m'. We refer to determining the answer value of P, *AD* (P : *CD* $\mapsto$ *AD*) as the *imported success problem*. In addition, in order to obtain analysis information for m' which is as accurate as possible we need to somehow propagate the call P : *CD* from m to m' so that the next time m' is analyzed such a call pattern is taken into account. We refer to this as the *imported calls problem*.

3.3.1 Solving the Imported Success Problem

The imported success problem is solved by means of a *success policy*, or *SP* for short. The behavior of the analyzer for predicates defined in m remains exactly as before. *SP* is needed because given a call pattern P : *CD* it will often be the case that an entry of exactly the form P : *CD* $\mapsto$ *AD* does not exist in the analysis results stored in the GAT for m'. In such case, the information already present may be of value in order to obtain a (temporary) answer pattern *AD*, and continue the analysis of module m.

In contrast, in many formalizations of non-modular analysis there is no explicit success policy. This is because if the call pattern P : *CD* has not been analyzed yet, the analysis algorithm forces its computation. Thus, the results of analysis do not depend on any particular success policy: when the analyzer reaches a fixed-point there is always an entry of the form P : *CD* $\mapsto$ *AD* for any call pattern P : *CD* which appears in the analysis graph. However, in a modular setting it is often convenient to delay the analysis of predicates defined in other modules until those modules are revisited. In general, those modules may have already been analyzed or they may be analyzed in the future. We will simply do the best possible given the information available in the *GAT*.

Several success policies can be defined which provide over- or under-approximations of the exact answer pattern *AD*$^{=}$ with different degree of accuracy. Note that this exact value *AD*$^{=}$ is the one which the flattening approach would compute. In this work we consider two kinds of success policies, those which are guaranteed to always provide over-approximations, i.e. *AD*$^{=} \sqsubseteq$ *SP*$(P :$ *CD*$, GAT)$, and those which provide under-approximations, i.e., *SP*$(P :$ *CD*$, GAT) \sqsubseteq$ *AD*$^{=}$. We will use the superscript $^{+}$ (resp $^{-}$) to indicate that a success policy over-approximates (resp. under-approximates).

In the experiments shown in this work, a very precise over-approximating success policy has been used, already proposed in [PH00] and defined as:

$$\begin{array}{l} SP^{+}_{All}(P : \mathit{CD}, GAT) = topmost(\mathit{CD}) \sqcap_{\mathit{AD}' \in app} \mathit{AD}' \text{ where} \\ app = \{\mathit{AD}' \mid (P : \mathit{CD}' \mapsto \mathit{AD}') \in \; GAT \text{ and } \mathit{CD} \sqsubseteq \mathit{CD}'\} \end{array}$$

The function $topmost$ obtains the topmost answer pattern for a call pattern. The notion of *topmost description* was already introduced in [BCHP96]. Informally, while a topmost description preserves the information on properties which are *downwards closed* whereas it loses those which are not. Note that taking $\top$ as answer pattern is also a correct over-approximation, but often less accurate than using topmost substitutions. For example, if a variable is known to be ground in the call pattern, it will continue being ground in the answer pattern and taking $\top$ as the answer pattern 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.

We refer to this success policy as SP^{+}_{all} because it uses *all* entries in GAT which are *applicable* to the call pattern in the sense that the call pattern already computed is more general than the call being analyzed.

The counter-part of SP^{+}_{all} is the function SP^{-}_{all} which is defined as:

$$\begin{array}{l} SP^{-}_{All}(P : \mathit{CD}, GAT) = \sqcup_{\mathit{AD}' \in app} \mathit{AD}' \text{ where} \\ app = \{\mathit{AD}' \mid (P : \mathit{CD}' \mapsto \mathit{AD}') \in \; GAT \text{ and } \mathit{CD}' \sqsubseteq \mathit{CD}\} \end{array}$$

Note the change in the direction of the applicability relation (the call pattern in the *GAT* has to be more particular than the one being analyzed) and the use of the lub operator instead of the glb. Also, note that taking, for example, $\bot$ as an under-approximation is correct but SP^{-}_{all} is more precise.

As shown in [PCH^{+}04] using SP^{+} policies has the advantage that at any point during the modular analysis, even when a fixpoint has not been reached yet, the information obtained for each module is always a correct over-approximation. The drawback is that when the fixpoint is reached it may not be minimal, i.e., information is not as precise as it could be. In contrast, SP^{+} policies obtain the least fixpoint (most precise information) but only produce correct results when the fixpoint it reached. SP^{+} policies can be useful during program development.

3.3.2 Solving the Imported Calls Problem

As the analysis is context-sensitive, the call patterns for imported predicates are only known after the calling module is analyzed, but they cannot be processed until the imported module is selected for (re)analysis. These call patterns are therefore stored in another global data structure, the *temporary answer table* (TAT for short)[5]. When the imported module is scheduled for (re)analysis, all call patterns in the TAT are used as input for the analyzer.

3.4 Computing an intermodular fixed point

The intermodular fixed-point algorithm of CiaoPP takes one module of the program that needs (re)analysis, analyzes it storing the relevant information in GAT and TAT tables, and looks for another module which needs reanalysis. When a module is analyzed, it updates the entries in the global tables, and marks the modules which import it if the analysis results may improve the results of those modules. An intermodular fixed-point has been reached when there are no modules which need reanalysis.

Determining the optimal order in which the different modules in the program unit should be analyzed in order to get to a fixed-point as efficiently as possible is not trivial. Finding good scheduling strategies for intra-modular analysis is a topic which has received considerable attention and highly optimized algorithms exist which converge to a fixed-point quickly. Unfortunately, it is not possible to directly translate the same heuristics used in the intra-modular case to the inter-modular case. In the inter-modular case we have to take into account the time required to change from analysis of one module to another since this typically means reading a new module from disk. Thus, requests to process call patterns have to be grouped by modules in order to reduce the number of times we change context.

In the current implementation, two simple strategies have been used, in order to study the behavior of the analysis of modular programs in clearly different scenarios. Both strategies take the list of modules in a given order (a top-down and a bottom-up traversal of the intermodule dependency graph, respectively[6]), and traverse the list analyzing the modules which have pending call patterns, updating the corresponding global tables with the analysis results. This process is repeated until there are no pending call patterns for any module in the program.

We will refer to this intermodular fixed-point algorithm, scheduling one module at a time for analysis as the *modular approach*.

[5]In fact, GAT and TAT are implemented using the same table, and TAT entries are marked as needing reanalysis, in order to provide more precise results than those obtained applying the success policy, as soon as the module is scheduled for (re)analysis. There are more details in Section 7 and [PCH$^+$04].

[6]All modules which belong to the same cycle in the graph have been considered at the same depth, and therefore those modules will be selected in any order.

4 Empirical results

The CiaoPP implementation of the framework summarized above has been tested by parameterizing it in several ways, in order to study the overall behavior of the system. Different tradeoffs and characteristics of the analysis of modular programs have been studied:

Flattened vs. modular First, the flattened approach of Section 4.1 has been compared to the intermodular fixpoint of Section 7. Although it is predictable that the modular, separate analysis will be slower than the flattened approach (due to the overhead in loading/unloading modules, etc.), it is interesting to study by how much. In addition, in some cases the analysis of a whole program may be unfeasible due to hardware limitations, but in the intermodular fixpoint approach this limitation can be overcome ** We have to find an example of this! **.

Intermodular scheduling policies Another aspect to study is related to the influence of the module selection policy in the efficiency of the analysis. The scheduling policies used have been already described in Section 7. We will refer to them as *naive_top_down* and *naive_bottom_up*, respectively.

Success policies Two success policies have been compared in both scheduling policies: an over-approximating policy, SP^{+}_{all}, and an under-approximating one, SP^{-}_{all}, as described in Section 3.3. Although there may be other success policies, we estimate that these ones are the most effective policies, as they bring the closest results to $SP^{=}$.

Incremental analysis of modular programs Finally, the analysis of a modular program from scratch using the monolithic approach has been compared to the reanalysis of that program after making specific modifications in the source code. This comparison illustrates the advantages of analyzing only the module which has changed (and the modules affected by that change) instead of reanalyzing the whole program from scratch.

Three different kinds of source code modifications have been done: a simple change that keeps the same analysis results, a modification in the source code such that exported predicates produce after the change more general analysis results, and a change that results in the exported predicates producing a more precise answer pattern.

Note that when there are changes in the source code which do not improve or invalidate previous analysis results, nor generate new call patterns for imported modules, there are clear advantages in using the modular approach, since only one module must be analyzed at a time. In contrast, in the monolithic, non-modular analysis the whole program must be analyzed. Also note that this kind of changes may occur more often if assertions are used on a regular basis, as they can bring very precise answer patterns, similar to the results provided during the analysis.

The second kind of change studied corresponds to performing a modification in an exported predicate which results in this predicate providing more general analysis results. The change consists in the addition of a clause to all the exported predicates of a module in which all arguments are free variables[7]. This approach then forces the reanalysis of the modules which call the changed module.

The third type of source change represents a change that makes the analysis results for exported predicates be more precise than the ones obtained before. This is done by removing all clauses of exported predicates of a module except the first non recursive one [8]. This will bring in general analysis results which are more specific than the results previously obtained, making them invalid in most cases, and producing the reanalysis of the calling modules.

In following subsections the selected benchmark programs are described, and the results of the tests are studied in detail. Two modes domains have been considered: Def, a simplified version of the Pos domain, and *Sharing-freeness*, which gets information on set sharing and freeness.

4.1 Brief description of the benchmarks used

A brief description of the selected benchmarks follows:

aiakl This program is the initialization phase for abstract unification in the AKL analyzer (by D. Sahlin and T. Sjöland). It is composed by 4 modules, two of them import each other, and therefore there is a cycle in the intermodular dependency graph.

ann This is the &-Prolog implementation of the MEL annotator (by K. Muthukumar, F. Bueno, M. garcía de la Banda, and M. Hermenegildo). In this case the code is distributed in 3 modules with no cycles in the intermodular dependency graph.

[7]In the $Sharing - Freeness$ domain this addition might not provide a more general analysis result, as this kind of clause does not provide a top success substitution. However, the tests have been performed using the same change also in the case of $Sharing - Freeness$ to make the tests homogeneous across the different domains.

[8]Mutually recursive predicates are also considered. If the exported predicate has only recursive clauses, they are replaced by a fact with all arguments ground.

bid This program computes an opening bid for a bridge hand, by J. Conery. It is composed by 7 modules, with no cycles in the intermodular dependency graph.

boyer The boyer benchmark is a reduced version of the Boyer/Moore theorem prover (by E. Tick). The program has been separated in four modules with a cycle between two modules.

hanoi This is the classic hanoi towers program, distributed in two simple modules with no cycles in the intermodular graph.

peephole This program is the SB-Prolog peephole optimizer. In this case, the program is split in three modules, but there are two cycles in the intermodular dependency graph, and there are several intermodular cycles at the predicate call level.

prolog_read corresponds to a simplified version of the code used by the Ciao compiler for reading terms. It is composed by three modules, having a cycle between two of them.

unfold_ is a fragment of the CiaoPP preprocessor which contains the partial evaluator. It is distributed in 7 modules with no cycles between them, although many other modules of CiaoPP source code, while not analyzed, are consulted in order to get assertion information.

4.2 Analysis of a modular program from scratch

Table 2 shows the absolute times in milliseconds spent in analyzing the programs using the flattening approach. For every benchmark, the number of modules is shown, and the total analysis time is divided in several categories, represented by the following columns:

Load This column corresponds to the time spent in loading modules into Ciaopp system. This time includes the time used for reading the module to be analyzed and the time spent in reading the assertions of the imported modules (including system libraries).

Anal. This is the time spent in transforming the program to a normalized form, suitable for analyzing it.

Gen. Corresponds to the task of generating the global information (referred to before as the GAT and TAT tables). The information generated is related to the analysis results of all exported and multifile predicates, new call patterns of imported predicates generated during the analysis of each module, and the modules that import the module and can improve their analysis results by reanalysis.

Total Time elapsed since the analyzer is called until it finishes completely. It is the sum of previous columns, plus some extra time spent in other tasks, such as the generation of the intermodular dependency graph, handling the list of modules to get the next module to be analyzed, etc.

Tables 3 and 4 contain the time spent in analyzing the benchmarks with the Def and *Sharing-freeness* domains, respectively, and using the different scheduling policies described in Section 7. The numbers in these tables are relative to the monolithic analysis time. The *naive_bottom_up* and *naive_top_down* global scheduling policies are compared, as well as the SP^-_{all} and SP^+_{all} success policies.

In these and the following tables referring to execution time, column **#It** represents the number of iterations of the intermodular fixed-point algorithm: it contains the total number of times any module of the program is selected for (re)analysis. This number will always be greater or equal than **Mod**. When the number of iterations is greater than the number of modules, some modules have been reanalyzed several times in order to reach a fixed-point.

The **overall** row stands for the weighted overall results summarizing the different benchmarks in a single set of numbers. These tables show that the SP^+_{all} success policy is clearly more efficient than SP^-_{all} in both domains and both scheduling policies. The SP^-_{all} success policy is so inefficient when analyzing from scratch because this policy returns $\perp$ as success pattern for all the calls to imported predicates defined in modules which have not been analyzed yet, thus causing more iterations than SP^+_{all} (with some exceptions, such as `aiakl` and `hanoi`).

Comparing the scheduling policies, we only can observe a slight difference in the time taken using the *naive_top_down* or the *naive_bottom_up* strategies. This result seems to reflect that the order of the modules, at least in simple approaches as the ones used in this work, is not so relevant when analyzing a modular program as was initially expected.

Memory Consumption when analyzing from scratch. We now compare the maximum memory required for the analysis in the flattened and the modular approaches to the analysis of modular programs from scratch. Tables 5 and 6 show the maximum memory consumption during the analysis of the flattened approach (column **Monolithic**), and the use of memory of the modular approach (using both global scheduling policies described before) relative to the monolithic case (columns SP^+_{all} and SP^-_{all} for the corresponding success policies). The results show that the modular approach is clearly better in terms of maximum memory consumption than the monolithic approach (except for the outlying value of the `aiakl` test in SP^+_{all}).

4.3 Reanalysis of a modular program after a change in the code

As explained in Section 4, we have also studied the incremental cost of reanalysis of a modular program after a change, for different typical changes. The figures presented have been obtained computing the weighted average of applying the given change in each module of the program and then reanalysing the whole program. The weight of a module has been measured as the number of clauses of the module. Numbers are relative to those of table 2.

In the first case, a simple change in a module with no implications in the analysis results of that module has been tested. The results in the Def and $Sharing - freeness$ domains are shown in Tables 7 and 8, respectively. In this test we can observe that the analysis domain used is very relevant to the efficiency of the modular approach: the analysis of a complete program in the $Sharing - freeness$ domain is much more expensive than the reanalysis of a module, while the difference is smaller (although still significant) in the case of Def.

In addition, in the case of the $Sharing - freeness$ domain the `unfold` benchmark has more relative impact on the results, since it is much more expensive than the rest of the benchmarks, as can be seen in Table 2.

The second kind of change, a modification in the exported predicates of a module that makes the analysis produce more general results, is shown in Tables 9 and 10. This change may in general imply the reanalysis of the modules which import the changed module, and thus it means that the time taken in reaching a new fixed-point will be greater than in the previous case. Nevertheless, the "overall" row shows that the reanalysis time is always smaller than the time spent in analyzing the flattened program. As in the previous test, the analysis domain has a vital relevance in the relative advantage of the modular approach.

Finally, Tables 11 and 12 show how the modular approach behaves in the case of a change which produces more precise analysis results (making invalid former results). Similar conclusions as before can be inferred from these tables.

5 Conclusions

We have provided an empirical study of several models proposed context-sensitive analysis of modular programs with the objective of providing experimental evidence on the scalability of these models and, specially, on the impact on performance of the different choices left open in those models.

Our results shed light on the different choices available. In the case of analyzing a modular program from scratch, the modular analysis approach is slower than the flattening approach (i.e., having the complete program in memory, and analyzing it as a whole), due to the impact of code

Def					
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	905	16	82	1147
ann	3	891	334	229	1653
bid	7	1165	35	187	1639
boyer	4	891	145	154	1412
hanoi	2	699	8	41	852
peephole	3	1316	232	316	2132
prolog_read	3	856	344	372	1731
unfold_	7	3359	2016	4156	9989
Sharing-freeness					
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	871	26	97	1151
ann	3	924	679	276	2069
bid	7	1148	49	229	1696
boyer	4	901	227	161	1482
hanoi	2	683	18	59	850
peephole	3	1266	573	529	2603
prolog_read	3	863	4660	1841	7519
unfold_	7	3117	555083	7874	566558

Table 2: Time spent in seconds by the monolithic analysis of different benchmark programs

and related analysis information load and unload times. On the other hand, it does imply a lower maximum memory consumption which in some cases may be of advantage since it may allow analyzing programs of a certain critical size that would not fit in memory using the flattening approach. Also, this suggests future work on reducing the time spent in loading/unloading modules and storing analysis results.

We have also considered the case of reanalyzing a previously analyzed program, after making changes to it. This is relevant because this is the standard situation during program development, in which some modules change while others (and the libraries) remain unchanged. While in this phase the analysis results may not be needed in order to obtain highly optimized programs they are for static program validation and debugging. In this context the modular analysis, because of its more incremental nature, shows clear advantage in both time and memory consumption over the monolithic approach.

Global scheduling policy: naive_top_down											
		automatic SP^+					**automatic** SP^-				
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**#It**	**Load**	**Anal.**	**Gen.**	**Total**	**#It**
aiakl	4	6.77	10.94	2.16	6.02	10	5.59	7.13	1.55	4.87	9
ann	3	4.50	1.62	1.65	3.24	7	15.56	3.68	3.38	9.87	24
bid	7	7.15	7.97	1.34	5.81	17	12.62	11.60	2.32	9.81	28
boyer	4	2.83	1.23	0.66	2.27	5	9.01	2.67	1.38	6.39	15
hanoi	2	2.91	5.87	1.95	2.77	4	3.08	5.25	1.66	2.85	4
peephole	3	2.70	1.25	0.69	2.14	5	6.77	3.52	1.59	5.02	12
prolog_read	3	4.90	2.31	0.78	3.29	6	8.45	3.30	1.15	5.28	9
unfold_	7	3.60	1.87	0.85	2.04	9	9.30	3.69	2.05	4.93	19
Overall		4.25	1.94	0.91	2.82		9.05	3.70	2.00	5.77	

Global scheduling policy: naive_bottom_up											
		automatic SP^+					**automatic** SP^-				
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**#It**	**Load**	**Anal.**	**Gen.**	**Total**	**#It**
aiakl	4	6.40	11.31	1.85	5.65	10	5.08	7.63	1.51	4.63	9
ann	3	4.49	1.44	1.36	3.12	7	13.35	3.29	2.90	8.52	21
bid	7	7.19	8.23	1.64	5.83	17	12.31	11.29	2.34	9.62	27
boyer	4	2.91	1.32	0.72	2.30	5	8.81	2.96	1.44	6.28	15
hanoi	2	2.95	6.50	1.98	2.82	4	3.02	5.50	2.02	2.86	4
peephole	3	2.73	1.24	0.66	2.14	5	6.62	3.33	1.52	4.92	12
prolog_read	3	4.20	1.88	0.71	2.81	6	7.84	3.58	1.20	5.03	9
unfold_	7	3.44	1.13	0.43	1.65	10	7.30	2.32	0.95	3.41	18
Overall		4.12	1.41	0.58	2.57		8.01	2.80	1.16	4.85	

Table 3: Non-modular vs. SP^+ and SP^- policies when analyzing in the Def domain.

Global scheduling policy: naive_top_down											
		automatic SP^+					**automatic** SP^-				
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**#It**	**Load**	**Anal.**	**Gen.**	**Total**	**#It**
aiakl	4	6.32	11.58	1.74	5.47	10	6.11	5.38	1.33	5.16	9
ann	3	4.43	1.94	1.57	3.01	7	15.71	3.30	3.10	8.72	25
bid	7	8.03	7.39	1.34	6.23	17	14.22	9.71	2.19	10.56	28
boyer	4	2.73	1.20	0.76	2.18	5	8.66	2.42	1.48	6.05	15
hanoi	2	3.07	3.44	1.42	2.87	4	3.08	2.39	1.22	2.83	4
peephole	3	2.71	1.16	0.64	1.88	5	8.24	3.30	1.60	5.26	15
prolog_read	3	4.97	1.43	0.25	1.57	7	8.48	1.35	0.54	1.99	10
unfold_	7	4.63	1.32	0.69	1.33	12	11.42	0.98	1.55	1.04	21
Overall		4.66	1.32	0.66	1.36		10.17	0.99	1.43	1.15	

Global scheduling policy: naive_bottom_up											
		automatic SP^+					**automatic** SP^-				
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**#It**	**Load**	**Anal.**	**Gen.**	**Total**	**#It**
aiakl	4	5.76	10.08	1.87	5.04	10	5.18	5.04	1.18	4.42	9
ann	3	4.07	1.62	1.13	2.67	7	14.21	3.24	2.68	7.99	22
bid	7	7.21	6.00	1.24	5.58	17	11.73	7.98	1.72	8.78	27
boyer	4	2.77	1.18	0.74	2.20	5	10.68	4.20	2.01	7.66	15
hanoi	2	3.27	3.94	1.66	3.07	4	3.14	2.89	1.42	2.93	4
peephole	3	3.85	1.73	0.87	2.62	5	8.96	3.69	1.75	5.72	15
prolog_read	3	5.73	1.90	0.33	1.98	7	8.90	1.30	0.44	1.99	10
unfold_	7	5.00	1.29	0.70	1.31	12	9.06	0.98	0.96	1.03	21
Overall		4.83	1.30	0.68	1.35		9.22	0.99	0.99	1.13	

Table 4: Non-modular vs. SP^+ and SP^- policies when analyzing in the *Sharing-freeness* domain.

Global scheduling policy: naive_top_down				
Bench	**Mod**	**Monolithic**	SP^+	SP^-
aiakl	4	588480	0.45	2.63
ann	3	1770948	0.57	0.72
bid	7	1169196	0.39	0.46
boyer	4	1573700	0.66	0.85
hanoi	2	342028	0.79	1.79
peephole	3	1679244	0.38	0.70
prolog_read	3	1733272	0.74	0.87
unfold_	7	3412184	0.79	1.45
Overall			0.62	1.05

Global scheduling policy: naive_bottom_up				
Bench	**Mod**	**Monolithic**	SP^+	SP^-
aiakl	4	588480	0.59	0.80
ann	3	1770948	0.68	0.74
bid	7	1169196	0.54	0.53
boyer	4	1573700	0.86	0.86
hanoi	2	342028	0.86	0.86
peephole	3	1679244	0.60	0.66
prolog_read	3	1733272	0.82	0.89
unfold_	7	3412184	1.56	1.59
Overall			0.94	0.99

Table 5: Memory consumption of Non-modular vs. SP^+ and SP^- policies when analyzing in the Def domain.

Global scheduling policy: naive_top_down				
Bench	**Mod**	**Monolithic**	SP^+	SP^-
aiakl	4	654144	1.30	0.74
ann	3	2209388	0.57	0.72
bid	7	1195852	0.49	0.53
boyer	4	1747848	0.61	0.76
hanoi	2	409956	0.68	1.15
peephole	3	2248980	0.41	0.65
prolog_read	3	5385648	0.56	0.81
unfold_	7	11039512	0.62	0.52
Overall			0.60	0.65

Global scheduling policy: naive_bottom_up				
Bench	**Mod**	**Monolithic**	SP^+	SP^-
aiakl	4	654144	1.43	0.73
ann	3	2209388	0.67	0.71
bid	7	1195852	0.74	0.54
boyer	4	1747848	0.79	0.77
hanoi	2	409956	0.74	1.15
peephole	3	2248980	0.63	0.65
prolog_read	3	5385648	0.59	0.80
unfold_	7	11039512	0.65	0.69
Overall			0.67	0.72

Table 6: Memory consumption of Non-modular vs. SP^+ and SP^- policies when analyzing in the *Sharing-freeness* domain.

Global scheduling policy: naive_top_down									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	0.549	1.629	0.249	0.742	0.568	0.758	0.238	0.752
ann	3	0.653	0.319	0.371	0.728	0.612	0.344	0.571	0.707
bid	7	0.428	0.648	0.153	0.740	0.627	0.943	0.195	0.951
boyer	4	0.560	0.621	0.236	0.695	0.544	0.547	0.237	0.706
hanoi	2	0.764	1.708	0.423	0.865	1.042	1.958	0.585	1.117
peephole	3	0.634	0.383	0.242	0.690	0.619	0.356	0.226	0.685
prolog_read	3	0.879	0.656	0.236	0.920	0.804	0.773	0.214	0.864
unfold_	7	0.614	0.728	0.478	0.683	0.607	0.552	0.260	0.547
Overall		0.621	0.652	0.422	0.724	0.649	0.548	0.267	0.679

Global scheduling policy: naive_bottom_up									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	0.568	1.458	0.263	0.780	0.529	0.754	0.254	0.719
ann	3	0.603	0.326	0.408	0.708	1.261	0.961	1.045	1.463
bid	7	0.653	1.175	0.203	1.030	0.431	0.627	0.158	0.726
boyer	4	0.561	0.645	0.233	0.694	0.564	0.603	0.219	0.696
hanoi	2	0.774	1.708	0.472	0.876	0.722	1.500	0.390	0.831
peephole	3	0.655	0.420	0.264	0.727	0.642	0.405	0.234	0.703
prolog_read	3	0.728	0.571	0.184	0.693	0.728	0.670	0.203	0.708
unfold_	7	0.574	0.364	0.105	0.432	0.581	0.428	0.134	0.446
Overall		0.622	0.418	0.144	0.610	0.648	0.524	0.189	0.647

Table 7: Non-modular vs. SP^+ and SP^- policies after touching a module in the Def domain.

Global scheduling policy: naive_top_down									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	0.576	1.995	0.238	0.767	0.576	0.533	0.197	0.721
ann	3	0.609	0.284	0.436	0.628	0.622	0.281	0.645	0.655
bid	7	0.470	0.617	0.146	0.745	0.443	0.479	0.140	0.728
boyer	4	0.552	0.383	0.207	0.660	0.565	0.401	0.201	0.657
hanoi	2	0.781	0.963	0.328	0.877	0.757	0.722	0.316	0.846
peephole	3	0.674	0.301	0.223	0.629	0.647	0.329	0.263	0.625
prolog_read	3	0.733	0.565	0.086	0.500	0.710	0.478	0.101	0.450
unfold_	7	0.589	0.025	0.173	0.032	0.604	0.025	0.230	0.033
Overall		0.610	0.030	0.169	0.049	0.606	0.029	0.218	0.049

Global scheduling policy: naive_bottom_up									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	0.572	2.262	0.276	0.794	0.576	0.505	0.221	0.732
ann	3	0.811	0.315	0.507	0.791	0.608	0.300	0.667	0.654
bid	7	0.456	0.593	0.134	0.783	0.523	0.498	0.139	0.852
boyer	4	0.885	0.458	0.320	1.031	0.563	0.371	0.227	0.662
hanoi	2	0.770	0.944	0.384	0.879	1.022	0.759	0.390	1.171
peephole	3	0.674	0.318	0.235	0.649	0.668	0.325	0.240	0.642
prolog_read	3	0.731	0.568	0.092	0.503	0.705	0.441	0.084	0.421
unfold_	7	0.799	0.028	0.134	0.036	0.613	0.026	0.112	0.032
Overall		0.723	0.034	0.147	0.055	0.638	0.030	0.132	0.049

Table 8: Non-modular vs. SP^+ and SP^- policies after touching a module in the *Sharing-freeness* domain.

Global scheduling policy: naive_top_down									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	1.465	2.475	0.509	1.395	1.855	3.150	0.564	1.773
ann	3	1.968	0.772	0.973	1.605	3.814	1.022	1.524	2.660
bid	7	1.516	1.968	0.329	1.466	3.817	4.155	0.675	3.185
boyer	4	2.069	0.949	0.432	1.674	2.102	0.949	0.444	1.685
hanoi	2	0.522	1.083	0.407	0.529	1.017	2.167	0.675	0.964
peephole	3	1.072	0.466	0.326	0.948	1.264	0.546	0.388	1.092
prolog_read	3	1.071	0.907	0.286	0.970	3.001	1.722	0.515	2.116
unfold_	7	0.203	0.134	0.074	0.148	0.217	0.143	0.071	0.152
Overall		0.998	0.385	0.168	0.713	1.693	0.543	0.221	1.088

Global scheduling policy: naive_bottom_up									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	1.265	2.158	0.392	1.225	1.503	2.725	0.470	1.432
ann	3	1.734	0.791	1.291	1.521	4.320	1.122	1.713	3.001
bid	7	1.444	1.671	0.280	1.402	3.799	4.497	0.778	3.238
boyer	4	2.096	1.011	0.498	1.706	2.022	0.961	0.456	1.618
hanoi	2	0.514	1.125	0.415	0.525	1.029	2.208	0.659	0.970
peephole	3	1.074	0.516	0.355	0.974	1.314	0.535	0.383	1.126
prolog_read	3	1.073	0.781	0.248	0.885	2.809	1.458	0.449	1.919
unfold_	7	0.211	0.085	0.021	0.119	0.199	0.082	0.022	0.114
Overall		0.956	0.343	0.138	0.675	1.682	0.487	0.190	1.065

Table 9: Non-modular vs. SP^+ and SP^- policies after adding a most general clause to exported predicates in the Def domain.

Global scheduling policy: naive_top_down									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	1.630	5.444	0.495	1.579	2.106	3.369	0.528	1.915
ann	3	3.232	1.065	1.285	2.165	3.894	1.514	1.841	2.645
bid	7	1.564	1.553	0.257	1.434	3.845	2.958	0.554	3.068
boyer	4	2.006	0.854	0.514	1.614	1.998	0.820	0.555	1.611
hanoi	2	0.529	0.778	0.328	0.538	1.033	1.056	0.418	0.958
peephole	3	1.081	0.382	0.409	0.856	1.325	0.463	0.464	1.008
prolog_read	3	0.999	0.747	0.114	0.648	4.433	1.781	0.347	1.748
unfold_	7	0.196	0.023	0.033	0.024	0.218	0.022	0.042	0.023
Overall		1.147	0.031	0.113	0.055	1.896	0.039	0.182	0.077

Global scheduling policy: naive_bottom_up									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	1.583	5.487	0.520	1.591	1.525	2.897	0.447	1.423
ann	3	3.111	0.911	1.246	2.011	5.113	1.971	2.000	3.398
bid	7	1.478	1.311	0.250	1.365	3.612	3.003	0.605	2.919
boyer	4	2.023	0.916	0.540	1.631	2.041	0.868	0.493	1.635
hanoi	2	0.533	0.630	0.266	0.531	1.045	1.148	0.492	0.973
peephole	3	1.199	0.399	0.429	0.927	1.374	0.459	0.480	1.035
prolog_read	3	1.014	0.745	0.117	0.646	4.169	1.710	0.321	1.658
unfold_	7	0.220	0.022	0.021	0.024	0.197	0.022	0.018	0.024
Overall		1.147	0.031	0.105	0.055	1.914	0.040	0.165	0.078

Table 10: Non-modular vs. SP^+ and SP^- policies after adding a most general clause to exported predicates in the *Sharing-freeness* domain.

Global scheduling policy: naive_top_down									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	0.587	0.475	0.213	0.655	0.343	0.221	0.171	0.452
ann	3	2.018	0.393	0.459	1.391	1.762	0.390	0.440	1.258
bid	7	0.673	0.662	0.186	0.837	0.375	0.354	0.139	0.601
boyer	4	1.030	0.158	0.244	0.901	1.007	0.142	0.251	0.882
hanoi	2	0.494	0.958	0.358	0.505	0.204	0.083	0.236	0.248
peephole	3	1.997	0.606	0.345	1.565	1.791	0.407	0.262	1.380
prolog_read	3	0.796	0.286	0.128	0.671	0.803	0.306	0.107	0.657
unfold_	7	0.260	0.230	0.095	0.197	0.245	0.184	0.083	0.174
Overall		0.849	0.286	0.138	0.613	0.716	0.236	0.118	0.528

Global scheduling policy: naive_bottom_up									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	0.591	0.658	0.233	0.653	0.346	0.146	0.160	0.451
ann	3	1.666	0.328	0.379	1.192	1.455	0.333	0.391	1.071
bid	7	0.758	0.789	0.241	0.907	0.395	0.379	0.132	0.642
boyer	4	1.015	0.135	0.229	0.874	1.041	0.169	0.255	0.900
hanoi	2	0.501	0.833	0.260	0.501	0.207	0.083	0.195	0.249
peephole	3	2.196	0.611	0.311	1.683	1.688	0.421	0.259	1.324
prolog_read	3	0.749	0.234	0.096	0.629	0.759	0.293	0.099	0.611
unfold_	7	0.221	0.122	0.034	0.132	0.215	0.107	0.026	0.123
Overall		0.836	0.205	0.085	0.577	0.668	0.181	0.072	0.483

Table 11: Non-modular vs. SP^+ and SP^- policies after removing all clauses of exported predicates except the first non-recursive clause in the Def domain.

Global scheduling policy: naive_top_down									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	0.720	0.444	0.236	0.741	0.459	0.197	0.170	0.530
ann	3	1.587	0.336	0.343	1.003	1.343	0.327	0.381	0.932
bid	7	0.705	0.547	0.149	0.822	0.393	0.282	0.112	0.603
boyer	4	1.041	0.147	0.296	0.887	1.020	0.134	0.280	0.877
hanoi	2	0.525	0.389	0.249	0.517	0.212	0.037	0.153	0.245
peephole	3	1.937	0.396	0.253	1.247	1.749	0.371	0.282	1.179
prolog_read	3	0.818	0.271	0.039	0.313	0.681	0.207	0.039	0.259
unfold_	7	0.249	0.000	0.045	0.003	0.234	0.000	0.047	0.003
Overall		0.832	0.004	0.070	0.023	0.684	0.004	0.071	0.020

Global scheduling policy: naive_bottom_up									
		automatic SP^+				**automatic** SP^-			
Bench	**Mod**	**Load**	**Anal.**	**Gen.**	**Total**	**Load**	**Anal.**	**Gen.**	**Total**
aiakl	4	0.757	0.477	0.232	0.774	0.463	0.195	0.171	0.541
ann	3	1.639	0.342	0.373	1.034	1.373	0.325	0.345	0.909
bid	7	0.710	0.603	0.162	0.829	0.381	0.302	0.132	0.601
boyer	4	1.064	0.143	0.382	0.947	1.114	0.127	0.397	0.950
hanoi	2	0.509	0.444	0.249	0.504	0.218	0.037	0.169	0.261
peephole	3	2.066	0.412	0.279	1.340	1.868	0.416	0.300	1.241
prolog_read	3	0.737	0.269	0.038	0.303	0.676	0.208	0.036	0.257
unfold_	7	0.234	0.000	0.024	0.003	0.248	0.000	0.030	0.003
Overall		0.847	0.004	0.058	0.023	0.715	0.003	0.061	0.021

Table 12: Non-modular vs. SP^+ and SP^- policies after removing all clauses of exported predicates except the first non-recursive clause in the *Sharing-freeness* domain.

Part III

Efficient Local Unfolding with Ancestor Stacks for Full Prolog

1 Summary

In spite of the important research efforts in the area, the integration of powerful partial evaluation methods into practical compilers for logic programs is still far from reality. This is related both to 1) efficiency issues and to 2) the complications of dealing with practical programs. Regarding efficiency, the most successful unfolding rules used nowadays are based on structural orders applied over (covering) *ancestors*, i.e., a subsequence of the atoms selected during a derivation. Ancestor (sub)sequences are used to improve the specialization power of unfolding while still guaranteeing termination and also to reduce the number of atoms for which the wfo or wqo has to be checked. Unfortunately, maintaining the structure of the ancestor relation during unfolding introduces significant overhead. We propose an efficient, practical *local* unfolding rule based on the notion of covering ancestors which can be used in combination with any structural order and allows a stack-based implementation without losing any opportunities for specialization. Regarding the second issue, we propose assertion-based techniques which allow our approach to deal with real programs that include (Prolog) built-ins and external predicates in a very extensible manner. Finally, we report on our implementation of these techniques in a practical partial evaluator, embedded in a state of the art compiler which uses global analysis extensively (the `Ciao` compiler and, specifically, its preprocessor `CiaoPP`). The performance analysis of the resulting system shows that our techniques, in addition to dealing with practical programs, are also significantly more efficient in time and somewhat more efficient in memory than traditional tree-based implementations. We believe that our approach contributes to the practicality of state-of-the-art partial evaluation techniques.

2 Introduction

In spite of the important research efforts in the area, the integration of *Partial Deduction* (PD) [LS91, Gal93] methods into compilers seems to be still far from reality. We believe that the general uptake of PD methods is being hindered by two factors: the relative inefficiency of the PD method, and the complications brought about by the treatment of real programs. Indeed, the integration of powerful strategies to the unfolding rule –like the use of structural orders combined with the ancestor relation– can introduce a significant cost both in time and memory consumption of the specialization process. Regarding the treatment of real programs which include external predicates, non-declarative features, etc, the complications range from how to identify which predicates include these non-declarative features (ad-hoc but difficult to maintain tables are often used in practice for this purpose) to how to deal with such predicates during PD. A main objective of this paper is to contribute to the uptake of PE techniques by addressing some of these issues.

State-of-the-art partial evaluators integrate terminating unfolding rules for local control based on *structural* orders, like homeomorphic embedding [LB02] which can obtain very powerful optimizations. Moreover, they allow performing the ordering comparisons over *subsequences* of the full sequence of the selected atoms. In particular, the use of *ancestors* for refining sequences of visited atoms, originally proposed in [BSM92], greatly improves the specialization power of unfolding while still guaranteeing termination and also reduces the length of the sequences for which admissibility of new atoms has to be checked. Unfortunately, having to maintain dependency information for the individual atoms in each derivation during the generation of SLD trees has turned out to introduce overheads which seem to cancel out the theoretical efficiency gains expected. In order to address this issue, we introduce a novel unfolding rule based on the notion of covering ancestors which allows a very efficient implementation technique based on stacks. Our technique can significantly reduce the overhead incurred by the use of covering ancestors without losing any opportunities for specialization. We outline as well a generalization that allows certain non-leftmost unfoldings with the same assurances.

In order to deal with real programs that include (Prolog) built-ins and external predicates, we rely on assertion-based techniques [PBH00b]. The use of assertions provides *extensibility* in the sense that users and developers of partial evaluators can deal with new external predicates during PE by just adding the proper assertions to these predicates –without having to maintain ad-hoc tables or modifying the partial evaluator itself. We report on our implementation of our technique in a practical, state-of-the-art partial evaluator, embedded in a production compiler which uses assertions and global analysis extensively (the `Ciao` compiler [BCC+04] and, specifically, its preprocessor `CiaoPP`[HPBLG03b]).

3 Background

We assume some basic knowledge on the terminology of logic programming. See for example [Llo87b] for details. Very briefly, an *atom* A is a syntactic construction of the form

$p(t_1, \ldots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \ldots, t_n$ are terms. The function $pred$ applied to atom A, i.e., $pred(A)$, returns the predicate symbol p/n for A. A *clause* is of the form $H \leftarrow B$ where its head H is an atom and its body B is a conjunction of atoms. A *definite program* is a finite set of clauses. A *goal* (or query) is a conjunction of atoms. The concept of *computation rule* is used to select an atom within a goal for its evaluation.

Definition 3.1 (computation rule) *A* computation rule *is a function $\mathcal{R}$ from goals to atoms. Let G be a goal of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$, $k \geq 1$. If $\mathcal{R}(G) = A_R$ we say that A_R is the* selected *atom in G.*

The operational semantics of definite programs is based on derivations.

Definition 3.2 (derivation step) *Let G be $\leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause in P. Then G' is* derived *from G and C via $\mathcal{R}$ if the following conditions hold:*

$$\begin{array}{r} \theta = mgu(A_R, H) \\ G' \textit{ is the goal } \leftarrow \theta(B_1, \ldots, B_m, A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_k) \end{array}$$

The definition above differs from standard formulations (such as that in [Llo87b]) in that the atoms newly introduced in G' are not placed in the same position where the selected atom A_R used to be, but rather they are placed to the left of any atom in G. For definite programs, this is correct since goals are conjunctions, which enjoy the commutative property.

As customary, given a program P and a goal G, an *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees*. A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer for goal G. Such a derivation is called *failed* if it is not possible to perform a derivation step with G_n.

In order to compute a *partial deduction* (PD) [LS91], given an input program and a set of atoms (goal), the first step consists in applying an *unfolding rule* to compute finite (possibly incomplete) SLD trees for these atoms. Given an atom A, an unfolding rule computes a set of finite SLD derivations $D_1, \ldots, D_n$ (i.e., a possibly incomplete SLD tree) of the form $D_i = A, \ldots, G_i$ with computer answer substitution θ_i for $i = 1, \ldots, n$ whose associated resultants are $\theta_i(A) \leftarrow G_i$. Therefore, this step returns the set of resultants, i.e., a program, associated to the root-to-leaf derivations of these trees. We refer to [LB02] for details. In order to ensure the local termination of the PD algorithm while producing useful specializations, the unfolding rule must incorporate some non-trivial mechanism to stop the construction of SLD trees. Nowadays, well-founded orderings (wfo) [BSM92, MD96] and well-quasi orderings (wqo) [SG95, Leu98] are broadly used in the context of on-line PE techniques (see, e.g., [Gal93, LMDS98, SG95]). Formally, let $\leq_S$ be a wqo, we denote by $Admissible(A, (A_1, \ldots, A_n), \leq_S)$, with $n \geq 0$, the truth value of the expression $\forall A_i,\ i \in \{1, \ldots, n\}\ : A \leq_S A_i$. In wfo, it is sufficient to verify that the selected atom is strictly smaller than the previous comparable one (if one exists). Let $<$ be a wfo, by $Admissible(A, (A_1, \ldots, A_n), <)$, with $n \geq 0$, we denote the truth value of the expression $A < A_n$ if $n \geq 1$ and $true$ if $n = 0$. We will denote by *structural order* a wfo or a wqo (written as $\triangleleft$ to represent any of them). Among the structural orders, well-quasi orderings (and *homeomorphic embedding* [Kru60] in particular) have proved to be very powerful in practice.

State-of-the-art unfolding rules allow performing ordering comparisons over *subsequences* of the full sequence of the selected atoms of a derivation by organizing atoms in a *proof tree* [Bru91], achieving further specialization in many cases while still guaranteeing termination. The essence of the most advanced techniques is based on the notion of *covering ancestors* [BSM92].

Definition 3.3 (ancestor relation) *Given a derivation step and A_R, B_i, $i = 1, \ldots, m$ as in Def. 3.2, we say that A_R is the* parent *of the instance of B_i, $i = 1, \ldots, m$, in the resolvent and in each subsequent goal where the instance originating from B_i appears. The* ancestor *relation is the transitive closure of the parent relation.*

Usually, the ancestor test is only applied on *comparable* atoms, i.e., ancestor atoms with the same predicate symbol. This corresponds to the original notion of covering ancestors [BSM92]. Given an atom A and a derivation D, we denote by $Ancestors(A, D)$ the sequence of ancestors of A in D as defined in Def. 3.3. It captures the dependency relation implicit within a *proof tree*.

It has been proved [BSM92] that any infinite derivation must have at least one inadmissible *covering ancestor* sequence, i.e., a subsequence of the atoms selected during a derivation. Therefore, it is sufficient to check the selected ordering relation $\triangleleft$ over the covering ancestor subsequences in order to detect inadmissible derivations. An SLD derivation is *safe* with respect to an order (wfo or wqo) if all covering ancestor sequences of the selected atoms are admissible with respect to that order.

4 The Usefulness of Ancestors

We now illustrate some of the ideas discussed so far and, specially, the relevance of ancestor tracking, through an example. Our running example is the program in Figure 7, which imple-

```
qsort([],R,R).                partition([],_,[],[]).
qsort([X|L],R,R2) :-          partition([E|R],C,[E|Left1],Right) :-
   partition(L,X,L1,L2),         E =< C, partition(R,C,Left1,Right).
   qsort(L2,R1,R2),           partition([E|R],C,Left,[E|Right1]) :-
   qsort(L1,R,[X|R1]).           E > C,  partition(R,C,Left,Right1).
```

Figure 7: A quick-sort program

$$\begin{tikzcd}
\mathbf{1}.\underline{\mathtt{qs([1,1,1],R,[])}}^{\{\}} \arrow[d] \\
\mathbf{2}.\underline{\mathtt{p([1,1],1,L1,L2)}^{\{1\}}}, \mathbf{3}.\mathtt{qs(L2,R1,[])}^{\{1\}}, \mathbf{4}.\mathtt{qs(L1,R,[1|R1])}^{\{1\}} \arrow[d, "\{\mathtt{L1}\mapsto\mathtt{[1|L]}\}"] \\
\mathbf{5}.\underline{\mathtt{1 =< 1}}^{\{1,2\}}, \mathbf{6}.\mathtt{p([1],1,L,L2)}^{\{1,2\}}, \mathbf{3}.\mathtt{qs(L2,R1,[])}^{\{1\}}, \mathbf{4}.\mathtt{qs([1|L],R,[1|R1])}^{\{1\}} \arrow[d] \\
\mathbf{6}.\boxed{\underline{\mathtt{p([1],1,L,L2)}}}^{\{1,2\}}, \mathbf{3}.\mathtt{qs(L2,R1,[])}^{\{1\}}, \mathbf{4}.\mathtt{qs([1|L],R,[1|R1])}^{\{1\}} \arrow[d, "\{\mathtt{L}\mapsto\mathtt{[1|L']}\}"] \arrow[dddddd, dash, dashed, bend right] \\
\mathbf{7}.\underline{\mathtt{1 =< 1}}^{\{1,2,6\}}, \mathbf{8}.\mathtt{p([],1,L',L2)}^{\{1,2,6\}}, \mathbf{3}.\mathtt{qs(L2,R1,[])}^{\{1\}}, \mathbf{4}.\mathtt{qs([1,1|L'],R,[1|R1])}^{\{1\}} \arrow[d] \\
\mathbf{8}.\underline{\mathtt{p([],1,L',L2)}^{\{1,2,6\}}}, \mathbf{3}.\mathtt{qs(L2,R1,[])}^{\{1\}}, \mathbf{4}.\mathtt{qs([1,1|L'],R,[1|R1])}^{\{1\}} \arrow[d, "\{\mathtt{L'}\mapsto\mathtt{[]},\mathtt{L2}\mapsto\mathtt{[]}\}"] \\
\mathbf{3}.\underline{\mathtt{qs([],R1,[])}^{\{1\}}}, \mathbf{4}.\mathtt{qs([1,1],R,[1|R1])}^{\{1\}} \arrow[d, "\{\mathtt{R1'}\mapsto\mathtt{[]}\}"] \\
\mathbf{4}.\underline{\mathtt{qs([1,1],R,[1])}^{\{1\}}} \arrow[d] \\
\mathbf{9}.\boxed{\underline{\mathtt{p([1],1,L1',L2')}}}^{\{1,4\}}, \mathbf{10}.\mathtt{qs(L2',R1',[1])}^{\{1,4\}}, \mathbf{11}.\mathtt{qs(L1',R,[1|R1'])}^{\{1,4\}}
\end{tikzcd}$$

Figure 8: Derivation with Ancestor Annotations

ments the well known quick-sort algorithm, "`qsort`", using difference lists. Given an initial query of the form ←*qsort(List,Result,Cont)*, where *List* is a list of numbers, the algorithm returns in *Result* a sorted difference list which is a permutation of *List* and such that its continuation is *Cont*. For example, for the query $\leftarrow qsort([1,1,1],L,[])$, the program should compute `L=[1,1,1]`, constructing a finite SLD tree.

Consider now Fig. 8, which presents an incomplete SLD derivation for our quick-sort program and the query $\leftarrow qsort([1,1,1],R,[])$ using a leftmost unfolding rule. For conciseness, predicates `qsort` and `partition` are abbreviated as `qs` and `p`, respectively in the figure. Note that each atom is labeled with a number (an identifier) for future reference[9] and a superscript which contains the list of ancestors of that atom. Let us assume that we use the *homeo-*

[9]By abuse of notation, we keep the same number for each atom throughout the derivation although it may be further instantiated (and thus modified) in subsequent steps. This will become useful for continuing the example later.

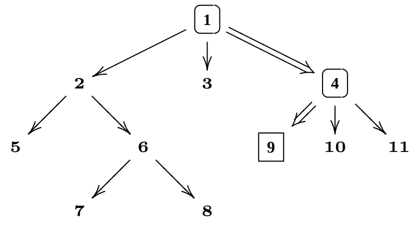

Figure 9: Proof tree for the example.

morphic embedding order [Leu98] as structural order. If we check admissibility w.r.t. the full sequence of atoms, i.e., we do not use the ancestor relation, the derivation will stop when atom number **9**, i.e., $\mathtt{p}([1], 1, \mathtt{L}', \mathtt{L2}')$, is found for the second time. The reason is that this atom is not strictly smaller than atom number **6** which was selected in the third step, indeed, they are equal modulo renaming.[10]

This unfolding rule is too conservative, since the process can proceed further without risking termination. The crucial point is that the execution of atom number **9** does not depend on atom number **6** (and, actually, the unfolding of **6** has been already *completed* when atom number **9** is being considered for unfolding). Figure 9 shows the proof tree associated to this derivation where nodes are labeled with the numbers assigned to each atom, instead of the atoms themselves. Note that, in order to decide whether or not to evaluate atom number **9**, it is only necessary to check that it is strictly smaller than atoms **4** and **1**, i.e., than those which are its *ancestors* in the proof tree. On the other hand, and as we saw before, if the full derivation is considered instead, as in Fig. 8, atom **9** will be compared also with atom **6** concluding imprecisely that the derivation may not be safe.

Despite their obvious relevance, unfortunately the practical applicability of unfolding rules based on the notion of covering ancestor is threatened by the overhead introduced by the implementation of this notion. A naive implementation of the notion of ancestor keeps –for each atom– the list of its ancestors, as it is depicted in Fig. 8. This implementation is relatively efficient in time but presents a high overhead in memory consumption. Our experiments show that the partial evaluator can run out of memory even for simple examples. A more reasonable implementation maintains the proof tree as a global structure. This greatly reduces memory consumption but the cost of traversing the tree for retrieving the ancestors of each atom introduces a significant slowdown in the PE process. We argue that our implementation technique is efficient in time and space, overcoming the above limitations.

[10]Let us note that the two calls to the builtin predicate `=<` which appear in the derivation can be executed since the arguments are properly instantiated. However, they have not been considered in the admissibility test since these calls do not endanger the termination of the derivation, as we will discuss in Sect. 6.

5 An Efficient Implementation for Local Unfolding

Our definition of *local unfolding* is based on the notion of *ancestor depth*.

Definition 5.1 (ancestor depth) *Given an SLD derivation $D = G_0, \ldots, G_m$ with $G_m = \leftarrow A_1, \ldots, A_k$, $k \geq 1$, the* ancestor depth *of A_i for $i = 1, \ldots, k$, denoted $depth(A_i, D)$ is the cardinality of the ancestor relation for A_i in D.*

Intuitively, the ancestor depth of an atom in a goal is the depth at which this atom is located in the proof tree associated to the derivation.

Definition 5.2 (local computation rule) *A computation rule $\mathcal{R}$ is* local *if $\forall D = G_0, \ldots, G_n$ such that $G_i = \leftarrow A_{i1}, \ldots, A_{im_i}$ for $i = 0, .., n$, it holds that:*

$$depth(\mathcal{R}(G_i), D) \geq depth(A_{ij}, D) \quad \forall j = 1, \ldots, m_i$$

Intuitively, a computation rule is local if it always selects one of the atoms which is deepest in the proof tree for the derivation. As a result, local computation rules traverse proof trees in a depth-first fashion, though not necessarily left to right nor in any other fixed order. Thus, in principle, in order to implement a local computation rule we need to record (part of) the derivation history (its proof tree). Note that the computation rule used in most implementations of logic programming languages, such as Prolog, always selects the leftmost atom. This computation rule, often referred to as left-to-right computation rule, is clearly a local computation rule. Selecting the leftmost atom in all goals guarantees that the selected atom is of maximal depth within the proof tree as it is traversed in a depth-first fashion –without the need of storing any history about the derivation.

An instrumental observation in our approach is that if the proof tree which is used in order to capture the ancestor relation is traversed depth-first, left-to-right, it can be interpreted as an *activation tree* [ASU86]. In fact, the ancestor subsequence in any point in time corresponds to the current *control word* [RS97] by simply regarding selected atoms as procedure calls. The control word for each execution state can be seen as the set of procedures whose execution has started and is not yet completed, bearing a strong relation with the stack of activation records which most compilers use as a run-time data structure. This data structure takes normally the form of a stack, and this suggests one of the central ideas of our approach: using stacks for storing ancestors. Another important observation is that the control word idea does not need to be restricted to leftmost computation and it works equally well as long as the computation rule is local. Indeed, sibling atoms have the same ancestor depth, they can be selected in any order and the notion of control word still applies. The advantages of computing the control word instead of the proof tree are clear: the control word corresponds to a single branch in the proof tree from the current selected atom to all its ancestors in the proof tree. Thus, the control word offers advantages both from memory and time consumption. The main difficulty for computing control words is to determine exactly when each item in the control word should be removed. To do this, we need to know when the computation of each predicate is finished. In logic programming terminology this corresponds to determining the success states for all predicates in the derivation. In principle, success states are not observable in SLD resolution other than for the top-level query.

We now propose an easy-to-implement modification to SLD resolution as presented in Section 3 in which success states for all internal calls are observable –and where the control word is available at each state. We will refer to this resolution as SLD resolution with ancestor stacks, or *ASLD* for short. The proposed modification involves 1) augmenting goals with an *ancestor stack*, which at each stage of the computation contains the control word of the derivation, which corresponds to *the ancestors of the next atom which will be selected for resolution*, and 2) adding pseudo-atoms to the goals used during resolution which mark a scope whose purpose is twofold: 2.1) when a mark is leftmost in a goal, it indicates that the current state corresponds to the success state for the call which is now on top of the ancestor stack, i.e., the call is completed, and the atom on top of the ancestor stack should be popped; 2.2) the atoms within the scope of the leftmost mark have maximal ancestor depth and thus a local unfolding strategy can be easily defined in the presence of these pseudo-atoms. We use the pseudo-atom $\uparrow$ (read as "pop") to indicate the end of a depth scope, i.e., after it we move up in the proof tree. It is guaranteed not to clash with any existing predicate name.

The following two definitions present the derivation rules in our ASLD semantics. Now, a state S is a tuple of the form $\langle G \mid \mathit{AS}\rangle$ where G is a goal and *AS* is an ancestor stack (or *stack* for short). To handle such stacks, we will use the usual stack operations: empty, which returns an empty stack, $\mathsf{push}(\mathit{AS}, Item)$, which pushes *Item* onto the stack *AS*, and $\mathsf{pop}(\mathit{AS})$, which pops an element from *AS*. In addition, we will use the operation $\mathsf{contents}(\mathit{AS})$, which returns the sequence of atoms contained in *AS* in the order in which they would be popped from the stack *AS* and leaves *AS* unmodified.

Definition 5.3 (derive) *Let $G = \leftarrow A_1, \ldots, A_R, \ldots, A_k$ be a goal with $A_1 \neq \uparrow$. Let $S = \langle G \mid \mathit{AS}\rangle$ be a state and AS be a stack. Let $\lhd$ be a structural order. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$ with $A_R \neq \uparrow$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause. Then $S' = \langle G' \mid \mathit{AS}'\rangle$ is* derived *from S and C via $\mathcal{R}$ if the following conditions hold:*

$$\begin{aligned} & Admissible(A_R, \mathsf{contents}(\mathit{AS}), \lhd) \\ & \theta = mgu(A_R, H) \\ G' \textit{ is the goal } & \leftarrow \theta(B_1, \ldots, B_m, \uparrow, A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_k) \end{aligned}$$

$$AS' = \textsf{push}(\textit{AS}, ren((A_R)))$$

The **derive** rule behaves as the one in Definition 3.2 but in addition: i) the mark $\uparrow$ ("pop") is added to the goal, and ii) a renamed apart copy of A_R, denoted $ren(A_R)$, is pushed onto the ancestor stack. As before, the **derive** rule is non-deterministic if several clauses in P unify with the atom A_R. However, in contrast to Definition 3.2, this rule can only be applied if 1) the leftmost atom in the goal is not a $\uparrow$ mark, and 2) the current selected atom A_R together with its ancestors does constitute an admissible sequence. If 1) holds but 2) does not, this derivation is stopped and we refer to such a derivation as *inadmissible*.

Definition 5.4 (pop-derive) *Let $G = \leftarrow A_1, \ldots, A_k$ be a goal with $A_1 = \uparrow$. Let $S = \langle G \mid \textit{AS}\rangle$ be a state and AS be a stack. Then $S' = \langle G' \mid AS'\rangle$ with $G' = \leftarrow A_2, \ldots, A_k$ and $AS' = \textsf{pop}(\textit{AS})$ is* pop-derived *from S.*

The **pop-derive** rule is used when the leftmost atom in the resolvent is a $\uparrow$ mark. Its effect is to eliminate from the ancestor stack the topmost atom, which is guaranteed not to belong to the ancestors of any selected atom in any possible continuation of this derivation.

Computation for a query G starts from the state $S_0 = \langle G \mid \textsf{empty}\rangle$. Given a non-empty derivation D, we denote by *curr_goal(D)* and *curr_ancestors*(D) the goal and the stack in the last state in D, respectively. At each step of a derivation D at most one rule, either **derive** or **pop-derive**, can be applied depending on whether the first atom in $curr\_goal(D)$ is a mark $\uparrow$ or not.

Example 5.5 *Fig. 10 illustrates the ASLD derivation corresponding to the derivation with explicit ancestor annotations of Fig. 8. Sometimes, rather than writing the atoms themselves, we use the same numbers assigned to the corresponding atoms in Fig. 8. Each step has been appropriately labeled with the applied derivation rule. Although rule* external-derive *has not been presented yet, we can just assume that the code for the external predicate* `=<` *is available and has the expected behavior.*

It should be noted that, in the last state, the stack contains exactly the ancestors of `partition([1],1,L1',L2')`*, i.e., the atoms* **4** *and* **1***, since the previous calls to* `partition` *have already finished and thus their corresponding atoms have been popped off the stack. Thus, the admissibility test for* `partition([1],1,L1',L2')` *succeeds, and unfolding can proceed further without risking termination. Note that* derive *steps w.r.t. a clause which is a fact are always followed by a* pop-derive *and thus they are optimized in the figure (and in the implementation, described in Section 7) by not pushing the selected atom A_R onto the stack and not including a $\uparrow$ mark into the goal which would immediately pop A_R from the stack.*

$$\begin{array}{c}
\langle \{ \underline{\texttt{qs([1,1,1],R,[])}} \} \mid [\,] \rangle \\
\big\downarrow{}_{derive} \\
\langle \{ \mathbf{2}, \mathbf{3}, \mathbf{4}, \uparrow \} \mid [\texttt{qsort([1,1,1],R,[])}] \rangle \\
\big\downarrow{}_{derive} \\
\langle \{ \mathbf{5}, \mathbf{6}, \uparrow, \mathbf{3}, \mathbf{4}, \uparrow \} \mid [\texttt{part([1,1],1,L1,L2)}, \texttt{qs([1,1,1],R,[])}] \rangle \\
\big\downarrow{}_{external-derive} \\
\langle \{ \mathbf{6}, \uparrow, \mathbf{3}, \mathbf{4}, \uparrow \} \mid [\texttt{part([1,1],1,L1,L2)}, \texttt{qs([1,1,1],R,[])} \rangle] \\
\big\downarrow{}_{derive} \\
\langle \{ \mathbf{7}, \mathbf{8}, \uparrow, \uparrow, \mathbf{3}, \mathbf{4}, \uparrow \} \mid [\texttt{part([1],1,L,L2)}, \texttt{part([1,1],1,L1,L2)}, \texttt{qs([1,1,1],R,[])}] \rangle \\
\big\downarrow{}_{external-derive} \\
\langle \{ \mathbf{8}, \uparrow, \uparrow, \mathbf{3}, \mathbf{4}, \uparrow \} \mid [\texttt{part([1],1,L,L2)}, \texttt{part([1,1],1,L1,L2)}, \texttt{qs([1,1,1],R,[])}] \rangle \\
\big\downarrow{}_{derive,pop-derive} \\
\langle \{ \uparrow, \uparrow, \mathbf{3}, \mathbf{4}, \uparrow \} \mid [\texttt{part([1],1,L,L2)}, \texttt{part([1,1],1,L1,L2)}, \texttt{qs([1,1,1],R,[])}] \rangle \\
\big\downarrow{}_{pop-derive} \\
\langle \{ \uparrow, \mathbf{3}, \mathbf{4}, \uparrow \} \mid [\texttt{part([1,1],1,L1,L2)}, \texttt{qs([1,1,1],R,[])}] \rangle \\
\big\downarrow{}_{pop-derive} \\
\langle \{ \mathbf{3}, \mathbf{4}, \uparrow \} \mid [\texttt{qsort([1,1,1],R,[])}] \rangle \\
\big\downarrow{}_{derive,pop-derive} \\
\langle \{ \mathbf{4}, \uparrow \} \mid [\texttt{qsort([1,1,1],R,[])}] \rangle \\
\big\downarrow{}_{derive} \\
\langle \{ \texttt{part([1],1,L1',L2')}, \mathbf{10}, \mathbf{11}, \uparrow, \uparrow \} \mid [\texttt{qsort([1,1],R,[1])}, \texttt{qsort([1,1,1],R,[])}] \rangle
\end{array}$$

Figure 10: ASLD Derivation for the example

Finally, since the goals obtained by ASLD resolution may contain atoms of the form $\uparrow$, resultants are cleaned up before being transferred to the global control level or during the code generation phase by simply eliminating all atoms of the form $\uparrow$.

It is easy to see that for each ASLD derivation D_S there is a corresponding SLD derivation D with the same computed answer substitution and the same goal without the $\uparrow$ atoms. Such SLD derivation is the one obtained by performing the same *derive* steps (with exactly the same clauses) using the same computation rule and by ignoring the *pop-derive* steps since goals in SLD resolution do not contain $\uparrow$ atoms. We will use $simplify(D_S) = D$ to denote that D is the SLD derivation which corresponds to D_S.

We would now like to impose a condition on the computation rule which allows ensuring that the contents of the stack are precisely the ancestors of the atom to be selected.

Definition 5.6 (depth-preserving) *A computation rule $\mathcal{R}$ is* depth-preserving *if for each non-empty goal $G = \leftarrow A_1, \ldots, A_k$ with $A_1 \neq \uparrow$, $\mathcal{R}(G) = A_R$ and $\uparrow \notin \{A_1, \ldots, A_R\}$.*

Intuitively, a depth-preserving computation rule always returns an atom which is strictly to the left of the first (leftmost) $\uparrow$ mark. Note that $\uparrow$ is used to separate groups of atoms which are at different depth in the proof tree. Thus, the notion of depth-preserving computation rules in ASLD resolution is *equivalent* to that of local computation rules in SLD resolution.

Proposition 5.7 (ancestor stack) *Let D_S be an ASLD derivation for initial query G in program P via a* depth-preserving *computation rule. Let D be an SLD derivation such that $simplify(D_S) = D$. Let $curr\_goal(D_S) = A_1, \ldots, A_n, \uparrow, \ldots$ with $A_i \neq \uparrow$ for $i = 1, \ldots, n$. Let $curr\_ancestors(D_S) = AS$. Then,* contents*(AS) $= Ancestors(A_i, D)$ for $i = 1, \ldots, n$.*

The next theorem guarantees that we do not lose any specialization opportunities by using our stack-based implementation for ancestors instead of the more complex tree-based implementation, i.e., our proposed semantics will not stop "too early". It is a consequence of the above proposition and the results in [BSM92].

Theorem 5.8 (accuracy) *Let D be an SLD derivation for query G in a program P via a local computation rule. Let $\triangleleft$ be a structural order. If the derivation D is safe w.r.t $\triangleleft$ then there exists an ASLD derivation D_S for G and P via a depth-preserving computation rule such that $simplify(D_S) = D$.*

Note that since our semantics disables performing any further steps as soon as inadmissible sequences are detected, not all local SLD derivations have a corresponding ASLD derivation. However, if a local SLD derivation is safe, then its corresponding D_S derivation can be found.

It is interesting to note that we can allow more flexible computation rules which are not necessarily depth-preserving while still ensuring termination. For instance, consider state

$$\langle A_1, \ldots, A_n, \uparrow, A_R, \ldots \mid [P_1|P] \rangle$$

with $\uparrow \notin \{A_1, \ldots, A_n\}$ and a non depth-preserving computation rule which selects the atom A_R to the right of the $\uparrow$ mark. Then, rule *derive* will check admissibility of A_R w.r.t. all atoms in the stack $[P_1|P]$. However, the topmost atom P_1 is an ancestor only of the atoms A_i to the left of A_R but it is not an ancestor of A_R. The more $\uparrow$ marks the computation rule jumps over to select an atom, the more atoms which do not belong to the ancestors of the selected atom will be in the stack, thus, the more accuracy and efficiency we lose. In any case, the stack will always be an over-approximation of the actual set of ancestors of A_R.

In principle, our local unfolding rule based on ancestor stacks can be used within any PD framework, including Conjunctive Partial Deduction (CPD). It should be noted that some CPD examples may require the use of an unfolding rule which is not depth-preserving to obtain the optimal specialization. As we discuss above, we cannot ensure accuracy results in these cases but in turn the use of local unfolding will clearly improve the efficiency of the PD process.

6 Assertion-based Unfolding for External Predicates

Most of real-life Prolog programs use predicates which are not defined in the program (module) being developed. We will refer to such predicates as *external*. Examples of external predicates are the traditional "built-in" predicates such as arithmetic operations (e.g., `is/2`, `<`, `=<`, etc.) or basic input/output facilities. We will also consider as external predicates those defined in a different module, predicates written in another language, etc. This section deals with the difficulties which such *external* predicates pose during PD.

When an atom A, such that $pred(A) = p/n$ is an external predicate, is selected during PD, it is not possible to apply the *derive* rule in Definition 3.2 due to several reasons. First, we may not have the code defining p/n and, even if we have it, the derivation step may introduce in the residual program calls to predicates which are private to the module M where p/n is defined. In spite of this, if the executable code for the external predicate p/n is available, and under certain conditions, it can be possible to fully evaluate calls to external predicates at specialization time. We use $\mathsf{Exec}(Sys, M, A)$ to denote the execution of atom A on a logic programming system Sys (e.g., `Ciao` or Sicstus) in which the module M where the external predicate p/n is defined has been loaded. In the case of logic programs, $\mathsf{Exec}(Sys, M, A)$ can return zero, one, or several computed answers for $M \cup A$ and then execution can either terminate or loop. We will use substitution sequences [CRV02] to represent the outcome of the execution of external predicates. A *substitution sequence* is either a finite sequence of the form $\langle\theta_1, \ldots, \theta_n\rangle$, $n \geq 0$, or an incomplete sequence of the form $\langle\theta_1, \ldots, \theta_n, \bot\rangle$, $n \geq 0$, or an infinite sequence $\langle\theta_1, \ldots, \theta_i, \ldots\rangle$, $i \in I\!N^*$, where $I\!N^*$ is the set of positive natural numbers and $\bot$ indicates that the execution loops. We say that an execution *universally terminates* if $\mathsf{Exec}(Sys, M, A) = \langle\theta_1, \ldots, \theta_n\rangle$, $n \geq 0$.

In addition to producing substitution sequences, it can be the case that the execution of atoms for (external) predicates produces other outcomes such as side-effects, errors, and exceptions. Note that this precludes the evaluation of such atoms to be performed at PE time, since those effects need to be performed at run-time. We say that an expression is *evaluable* when its execution 1) universally terminates, 2) it does not produce side-effects, 3) it is sufficiently instantiated to be executed, 4) it does not issue errors and 5) it does not generate exceptions. Clearly, some of the above properties are not computable (e.g., termination is undecidable in the general case). However, it is often possible to determine some *sufficient conditions* (SC) which are *decidable* and ensure that, if an atom A satisfies such conditions, then A is evaluable. Intuitively, SC can be thought of as a traditional precondition which ensures a certain behaviour of the execution of a procedure provided they are satisfied. To formalize this, we propose to use the "*computational* assertions" which are part of the assertion language [PBH00b] of `CiaoPP` in order to express that a certain predicate is evaluable under certain conditions. The following definition introduces the notion of an eval *annotation* as (part of) a computational assertion. We use id to denote the empty substitution, i.e., $\forall\ t\ , \mathsf{id}(t) = t$.

Definition 6.1 (eval annotations) *Let p/n be an external predicate defined in module M. The assertion* `:- trust comp p(X1,...,Xn) :  SC + eval.` *in the code for M is a correct* eval annotation *for predicate p/n in a logic programming system Sys if, $\forall\theta$, the expression $\theta(SC)$ is evaluable, and*

if $\mathsf{Exec}(Sys, M, \theta(SC)) = \langle\mathsf{id}\rangle$ *then* $\theta(p(X1, ..., Xn))$ *is evaluable*

One of the advantages of using this kind of assertion is that it makes it possible to deal with new external predicates (e.g., written in other languages) in user programs or in the system libraries without having to modify the partial evaluator itself. Also, the fact that the assertions are co-located with the actual code defining the external predicate, i.e., in the module M (as opposed to being in a large table inside the PD system) makes it more difficult for the assertion to be left out of sync when a modification is made to the external predicate. We believe this to be very important to the maintainability of a real application or system library.

Example 6.2 *The computational assertions in* `CiaoPP` *for the builtin predicate $\leq$ include, among others, the following one:*

```
:- trust comp A =< B : (arithexpr(A), arithexpr(B)) + eval.
```

which states that if predicate `=</2` *is called with both arguments instantiated to a term of type* `arithexpr`*, then the call is evaluable. The type* `arithexpr` *corresponds to arithmetic expressions which, as expected, are built out of numbers and the usual arithmetic operators. The type* `arithexpr` *is expressed in Ciao as a unary regular logic program. This allows using the underlying Ciao system in order to effectively decide whether a term is an* `arithexpr` *or not.*

The following definition extends our ASLD semantics by providing a new rule, **external-derive**, for evaluating calls to external predicates. Given a sequence of substitutions $\langle\theta_1, \ldots, \theta_n\rangle$, we define $Subst(\langle\theta_1, \ldots, \theta_n\rangle) = \{\theta_1, \ldots, \theta_n\}$.

Definition 6.3 (external-derive) *Let Sys be a logic programming system. Let*

$$G = \leftarrow A_1, \ldots, A_R, \ldots, A_k$$

be a goal. Let $S = \langle G \mathbin{\mathsf{I}} AS\rangle$ be a state and AS a stack. Let $\mathcal{R}$ be a computation rule such that $\mathcal{R}(G)$ $=A_R$ with $pred(A_R) = p/n$ an external predicate from module M. Let C be a renamed apart assertion `:- trust comp p(X1,...,Xn) :  SC + eval.` *Then, $S' =$*

$\langle G' \shortmid AS' \rangle$ *is* external-derived *from* S *and* C *via* $\mathcal{R}$ *in* Sys *if: 1)* $\sigma = mgu(A_R, p(X1, ..., Xn))$*, 2)* $\mathsf{Exec}(Sys, M, \sigma(SC)) = \langle \mathsf{id} \rangle$*, 3)* $\theta \in Subst(\mathsf{Exec}(Sys, M, A_R))$*, 4)* G' *is the goal*

$$\theta(A_1, \ldots, A_{R-1}, A_{R+1}, \ldots, A_k)$$

5)$AS' =$ *AS.*

Notice that, since after computing $\mathsf{Exec}(Sys, M, A_R)$ the computation of A_R is finished, there is no need to push (a copy of) A_R into *AS* and the ancestor stack is not modified by the **external-derive** rule. This rule can be nondeterministic if the substitution sequence for the selected atom A_R contains more than one element, i.e., the execution of external predicates is not restricted to atoms which are deterministic. The fact that A_R is evaluable implies universal termination. This in turn guarantees that in any ASLD tree, given a node S in which an external atom has been selected for further resolution, only a finite number of descendants exist for S and they can be obtained in finite time.

Example 6.4 *Consider the assertion in Example 6.2 and the atoms* `5` *and* `7`*, which are of the form* `1=<1`*, in the ASLD derivation of Fig. 8. Both atoms can be evaluated because*

$$\mathsf{Exec}(ciao, arithmetic, (arithexpr(1), arithexpr(1))) = \langle \mathsf{id} \rangle$$

This is a sufficient condition for $\mathsf{Exec}(ciao, arithmetic, (1 =< 1))$ *to be evaluable. Its execution returns* $\mathsf{Exec}(ciao, arithmetic, (1 =< 1)) = \langle \mathsf{id} \rangle$.

7 Experimental Results

We have implemented in our PD system the unfolding rule we propose, together with other variations in order to evaluate the efficiency of our proposal. Our PD system has been integrated in a practical state of the art compiler which uses global analysis extensively: the `CiaoPP` preprocessor [HPBLG03b]. For the tests, the whole system has been compiled using Ciao 1.11#275 [BCC$^+$04], with the bytecode generation option. All of our experiments have been performed on a Pentium 4 at 2.4GHz and 512MB RAM running GNU Linux RH9.0. The Linux kernel used is 2.4.25.

The results in terms of execution time are presented in Table 13. The programs used as benchmarks are indicated in the **Bench** column. We have chosen a number of classical programs for the analysis and PD of logic programs as benchmarks. In order to factor out the cost of global control, we have used in our experiments initial queries which can be fully unfolded using homeomorphic embedding with ancestors. The program `advisor3` is a variation of the advisor

	Execution Times				Relative Speed Up		
Bench	**Relation**	**Trees**	**Stacks**	**MEcce**	**Relation**	**Trees**	**MEcce**
advisor3	144	192	106	1240	1.36	1.81	11.70
nrev_80	mem	106490	15040	64970	∞	7.08	4.32
nrev_38	998	2804	806	4370	1.24	3.48	5.42
permute_7	mem	5226	2800	34680	∞	1.87	12.39
permute_6	476	614	336	3530	1.42	1.83	10.51
query	166	214	116	1290	1.43	1.84	11.12
qsort_80	mem	98514	8970	71870	∞	10.98	8.01
qsort_33	686	2432	454	4580	1.51	5.36	10.09
rev_80	984	1102	960	1400	1.02	1.15	1.46
zebra	1562	2276	994	186620	1.57	2.29	187.75
Overall					mem	7.19	12.25

Table 13: Comparison of Proof Trees Vs.Ancestor Stacks (Execution Time)

program in the DPPD [Leu02] library. The programs `query` and `zebra` are classical benchmarks for program analysis. Programs `qsort_80` and `qsort_33` correspond to the quick-sort program shown in the paper with pseudo-random lists of natural numbers of length 80 and 33 respectively. `nrev_80` and `nrev_38` correspond to the well-known naive reverse with lists of 80 and 38 natural numbers. `rev_80` is a reverse program with linear complexity which uses an accumulator. The initial query is, as before, a list of 80 natural numbers. Finally, `permute` is a permutation program which uses a nondeterministic deletion predicate. It is partially evaluated w.r.t. a list of 6 and 7 elements respectively. None of `advisor3`, `query`, nor `zebra` can be fully unfolded using homeomorphic embedding over the full sequence of selected atoms. Also, `nrev` and, as seen in the running example, `qsort` are potentially not fully unfolded if the input lists contain repetitions unless ancestors are considered. In the table, the following group of columns show execution time of the unfolding process with the different implementations of unfolding:

Relation We refer to an implementation where each atom in the resolvent is annotated with the list of atoms which are in its ancestor relation, as done in the example in Figure 8.

Trees This column refers to the implementation where the ancestor relations of the different atoms are organized in a proof tree.

Stacks The column **Stacks** refers to our proposed implementation based on ancestor stacks.

MEcce We have also measured the time that it takes to process the same benchmarks using Leuschel's M-Ecce (modular Ecce [Leu02]) system, compiled with the same version of Ciao and in the same machine.

The last set of columns compare the relative measures of the different approaches w.r.t. the **Stacks** algorithm. Finally, in the last row, labeled **Overall**, we summarize the results for the different benchmarks using a weighted mean, which places more importance on those benchmarks with relatively larger unfolding figures. We use as weight for each program its actual unfolding time. We believe that this weighted mean is more informative than the arithmetic mean, as, for example, doubling the speed in which a large unfolding tree is computed is more relevant than achieving this for small trees.

Let us explain the results in Table 13. Times are in milliseconds, measuring *runtime*, and are computed as the arithmetic mean of five runs. Three entries in the **Relation** column contain the value "mem", instead of a number, to indicate that the PD system has run out of memory. For each of these three cases, we have repeated the experiment with the largest possible initial query that **Relation** can handle in our system before running out of memory. This explains that the three benchmarks are specialized w.r.t. two different initial queries. As it can be seen in the column for relative speedups, **Relation** is quite efficient in time for those benchmarks it can handle, though a bit slower than the one based on stacks. However, its memory consumption is extremely high, which makes this implementation inadmissible in practice. Regarding column **Trees**, the implementation based on proof trees has a good memory consumption but is slower than **Relation** due to the overhead of traversing the tree for retrieving the ancestors of each atom. In comparison to M-ecce, the results provide evidence that our proof tree-based implementation is indeed comparable to state of the art systems, since the execution times are similar in some cases or even better in others. The last set of columns compares the relative execution times of the different approaches w.r.t. the **Stacks** algorithm which is the fastest in all cases. Indeed, **Stacks** is even faster than the implementation based on explicitly storing all ancestors of all atoms (**Relation**) while having a memory consumption comparable to (and in fact, slightly better than) the implementation based on proof trees. The actual speedup ranges from 1.15 in the case of `rev_80` to 10.98 in the case of `qsort_80`. This variation is due to the different shapes which the proof trees can have for the (derivations in the) SLD tree. In the case of `rev`, the speedup is low since the SLD tree consists of a single derivation whose proof tree has a single branch. Thus, in this case considering the ancestor sequence is indeed equivalent to considering the whole sequence of selected atoms. But note that this only happens for binary clauses. It is also worth noticing that the speedup achieved by the `Stacks` implementation increases with the size of the SLD tree, as can be seen in the three benchmarks which have been specialized w.r.t. different queries. The overall resulting speedup of our proposed unfolding rule over other existing ones is significant: over 7 times faster than our tree-based implementation.

We have also studied the memory required by the unfolding process (for lack of space details are in [PAH05a]). As for the case of execution time, the **Stacks** algorithm presents lower consumption than any other algorithm for all programs studied. The memory required by the **Relation** algorithm precludes it from its practical usage. Regarding the **Stacks** algorithm, not only it is significantly faster than the implementation based on trees. Also it provides a relatively important reduction (1.18 overall, computed again using a weighted mean) in memory consumption over **Trees**, which already has a good memory usage.

Altogether, when the results of Table 13 and the memory figures are combined, they provide evidence that our proposed techniques allow significant speedups while at the same time requiring somewhat less memory than tree based implementations and much better memory consumptions than implementations where the ancestor relation is directly computed. This suggests that our techniques are indeed effective and can contribute to making PD a practical tool.

As for future work, we plan to provide additional solutions for the problems involved in non-leftmost unfolding for programs with extra logical predicates beyond those presented in the literature [Leu94, EGM97, AHV02, LB02]. In particular, the intensive use of static analysis techniques in this context seems particularly promising. In our case we plan to take advantage of the fact that our PD system is integrated in `CiaoPP` which includes extensive program analysis facilities.

Part IV

A Program Transformation for Backwards Analysis of Logic Programs

1 Summary

The input to backwards analysis is a program together with properties that are required to hold at given program points. The purpose of the analysis is to derive initial goals or pre-conditions that guarantee that, when the program is executed, the given properties hold. The solution for logic programs presented here is based on a transformation of the input program, which makes explicit the dependencies of the given program points on the initial goals. The transformation is derived from the *resultants* semantics of logic programs. The transformed program is then analysed using a standard abstract interpretation. The required pre-conditions on initial goals can be deduced from the analysis results without a further fixpoint computation. For the modes backwards analysis problem, this approach gives the same results as previous work, but requires only a standard abstract interpretation framework and no special properties of the abstract domain.

2 Introduction

The input to backwards analysis is a program together with properties that are required to hold at given program points. The purpose of the analysis is to derive initial goals or pre-conditions that guarantee that, when the program is executed, the given properties hold. Discussion of the motivation for backwards analysis is given by King and Lu [KL02b] and Genaim and Codish [GC01]. For example, in a logic program, it is useful to know which instantiation modes of goals will definitely not produce run-time instantiation errors caused calls to built-in predicates with insufficiently instantiated arguments [KL02b], and which calls are sufficiently instantiated to ensure termination [GC01]. By contrast, program analysis frameworks usually start with given goals, and derive properties that hold at various program points, when those goals are executed.

An essential aspect of static analysis using abstractions or approximations is that the analysis results are *safe*. Backwards analysis algorithms have distinctive characteristics in this regard. The final result, namely (a description of) the set of initial goals that guarantee the establishment of the given properties, should be an *under* approximation of the actual set of goals that satisfy the requirements. Analyses usually yield an *over* approximation, this has led investigators to develop special abstract interpretations that give an under approximation.

In this paper we develop a method for using standard abstraction and over-approximation techniques, and still obtain valid results for backwards analysis. This is achieved by analysing not the original program, but rather a transformed program that makes explicit the dependencies between the given properties and initial goals.

The method is presented in terms of (constraint) logic programs. The essential idea is to transform a given program P into another program (or rather meta-program) whose semantics is a *dependency* relation $\langle A, B \rangle$, where B is a call at some specified program point, and A is an atomic goal for P. Analysis of this transformed program yields an over-approximation of the set of dependencies between A and B, which can then be examined to find goals A that guarantee some required property of B.

2.1 Making Derivations Observable

The transformation to be presented in Section 3 makes explicit the dependencies of program points on initial goals. The transformation can be viewed as the implementation of a more expressive semantics than usual. Standard semantics (such as least Herbrand models, c-semantics, s-semantics, call and success patterns for atomic goals, and so on) do not record explicitly the relationship between initial goals and specific program points. The *resultants semantics* [GLM96, GG94] provides a sufficiently expressive framework.

2.1.1 Resultants Semantics

A *resultant* is a formula $Q_1 \leftarrow Q_2$ where Q_1, Q_2 are conjunctions of atoms[11]. If Q_1 is an atom the resultant is a *clause*. Variables occurring in Q_2 but not in Q_1 are implicitly existentially quantified. All other variables are free in the resultant.

Definition 2.1 $\mathcal{O}_L(P)$

Given a definite program P, the resultants semantics $\mathcal{O}_L(P)$ is the set of all resultants[12] $p(\bar{X})\theta \leftarrow R$ *such that* $p(\bar{X})$ *is a "most general" atom for some predicate in* P, *and* $\leftarrow p(\bar{X}), \ldots, \leftarrow R$ *is an SLD- derivation (with a computation rule selecting the leftmost atom) of* $P \cup \{\leftarrow p(\bar{X})\}$ *with computed answer* θ. *Such a resultant represents a partial computation of the goal* $p(\bar{X})$. *We include the zero-length derivations of form* $p(\bar{X}) \leftarrow p(\bar{X})$.

[11]Standard terminology and notation for logic programming is used [Llo87a].

[12]Strictly speaking $\mathcal{O}_L(P)$ contains equivalence classes of resultants with respect to variable renaming, rather than resultants themselves.

From here on the leftmost computation rule is assumed and the subscript L in $\mathcal{O}_L(P)$ is omitted. There is also a fixpoint definition of $\mathcal{O}(P)$; abstract interpretation of the resultants and related semantics was considered in [CLM01].

Other standard semantics can be derived as abstractions of $\mathcal{O}(P)$. The subset of elements $p(\bar{X})\theta \leftarrow R \in \mathcal{O}(P)$ where $R = \mathsf{true}$ is isomorphic to the s-semantics [BGLM94b], from which in turn the c-semantics [Cla79] and the least Herbrand model [Llo87a] can be derived by computing all instances and ground instances respectively. Calls generated by a given goal can also be derived from $\mathcal{O}(P)$. The set of calls that arise from a given atomic goal A in a leftmost SLD derivation is given by the set $\mathsf{calls}(P, A) = \{B_1\theta \mid H \leftarrow B_1, \ldots, B_n \in \mathcal{O}(P), \mathsf{mgu}(A, H) = \theta\}$. We assume as usual that A is standardised apart from the elements of $\mathcal{O}(P)$.

2.2 Backwards Analysis Based on the Resultants Semantics

The possibility of using the resultants semantics for backwards analysis does not seem to have been considered previously. The relation $B \in \mathsf{calls}(P, A)$ can be read backwards; given B, A is a goal that invokes a call B.

We can capture the essential information about the dependencies between calls and goals using the *downwards closure* of $\mathcal{O}(P)$, denoted $\mathcal{O}^+(P)$. That is, $\mathcal{O}^+(P)$ is $\mathcal{O}(P)$ extended with all the instances obtained by substitutions for free variables, which are variables occurring in the resultants' heads. Then define a relation $\mathcal{D}$, called the *goal dependency* relation for P.

$$\mathcal{D}(A, B) \equiv (A \leftarrow B, \ldots, B_n \in \mathcal{O}^+(P))$$

The goal dependency relation for a program is closely related to the binary clause semantics of Codish and Taboch [CT99] (but is downwards closed with respect to the free variables).

Proposition 2.2 *Let P be a program, and $\mathcal{D}$ be the goal dependency relation for P. Then (i) if $\mathcal{D}(A, B)$ then $B \in \mathsf{calls}(P, A)$, and (ii) for all goals A and $B \in \mathsf{calls}(P, A)$, there exists a substitution σ such that $\mathcal{D}(A\sigma, B)$.*

Proof 2.3 *(i). If $\mathcal{D}(A, B)$ then $\mathcal{O}(P)$ contains $A' \leftarrow B'_1, \ldots, B'_n$ such that $A \leftarrow B, \ldots, B_n$ is an instance obtained by a substitution, say θ, for the variables in A'. Hence $\mathsf{mgu}(A, A') = \theta$ and $B = B'_1\theta$, and so $B \in \mathsf{calls}(P, A)$ (ii) If $B \in \mathsf{calls}(P, A)$ then $\mathcal{O}(P)$ contains $A' \leftarrow B'_1, \ldots, B'_n$, $\mathsf{mgu}(A, A') = \sigma$ and $B = B'_1\sigma$. The instance $A\sigma \leftarrow B, \ldots, B'_n\sigma$ is thus contained in the downwards closure $\mathcal{O}^+(P)$ and hence $\mathcal{D}(A\sigma, B)$ holds.*

Definition 2.4 *Let P be a program and $\mathcal{D}$ be the goal dependency relation for P. Let Θ and Φ be properties of atoms; that is, for every atom A, $\Theta(A)$ and $\Phi(A)$ are either true or false. We* *say that a* call-dependency $\Theta \rightarrow \Phi$ *follows from* $\mathcal{D}$ *if there does not exist* $\mathcal{D}(A, B)$ *such that* $\Theta(A) \wedge \neg\Phi(B)$.

Definition 2.5 *A property* Θ *is called* downwards closed *if, whenever* $\Theta(A)$ *holds,* $\Theta(A\varphi)$ *holds for all substitutions* φ.

Proposition 2.6 *Let* P *be a program, and* $\mathcal{D}$ *be the goal dependency relation for* P. *Suppose* $\Theta \rightarrow \Phi$ *follows from* $\mathcal{D}$*, and that* Θ *is a downwards closed property. Then for all goals* A*, and* $B \in \mathsf{calls}(P, A)$, $\Theta(A) \rightarrow \Phi(B)$.

Proof 2.7 *Let* A *be a goal, such that* $\Theta(A)$ *holds. For all* $B \in \mathsf{calls}(P, A)$*, we must establish that* $\Phi(B)$ *holds. For each such* B *there exists some instance* $A\sigma$ *such that* $\mathcal{D}(A\sigma, B)$ *by Proposition 2.2.* $\Theta(A\sigma)$ *holds since* Θ *is a downwards closed property. Hence* $\Phi(B)$ *holds since* $\Theta \rightarrow \Phi$ *follows from* $\mathcal{D}$.

Proposition 2.6 establishes that we can use the goal dependency relation of a program in order to establish dependencies between goals and calls, provided that the properties on goals are downwards closed. The next proposition shows that we can use over-approximations of the goal dependency relation to deduce dependencies.

Proposition 2.8 *Let* S *be a goal dependency relation and let* S' *be a relation including* S. *Then, if the call-dependency* $\Theta \rightarrow \Phi$ *follows from* S'*, it also follows from* S.

Proof 2.9 *Suppose that* $\Theta \rightarrow \Phi$ *follows from* S'. *Then there does not exist* $\mathcal{D}(A, B) \in S'$ *such that* $\Theta(A) \wedge \neg\Phi(B)$. *Hence such a pair does not exist in* S *either, and so* $\Theta \rightarrow \Phi$ *follows from* S.

We can also explain how our approach achieves the "under-approximations" of the conditions on initial goals discussed earlier. Given a call property Φ, suppose $\Theta \rightarrow \Phi$ follows from the goal dependency relation $\mathcal{D}$. In an over-approximation of $\mathcal{D}$, we will in general be able to establish dependencies $\Theta' \rightarrow \Phi$, such that $\Theta' \rightarrow \Theta$. Put another way, the larger the approximation is, the more chance there is of finding a counterexample $\mathcal{D}(A, B)$ such that $\Theta(A) \wedge \neg\Phi(B)$. The greater the over-approximation, the more restrictive are the properties Θ' for which $\Theta' \rightarrow \Phi$ can be shown.

The backwards analysis method can now be summarised in the following way. The concrete semantics on which we define properties is the goal dependency relation $\mathcal{D}$ for a given program. Given a program P we define a transformed program containing a predicate whose logical consequences contain the goal dependency relation $\mathcal{D}$. Using abstract interpretation of the transformed program, we compute approximations of $\mathcal{D}$, which can be used to establish dependencies between goals and calls, as proved in Propositions 2.6 and 2.8.

We shall also define an even more refined transformed program, whose semantics is restricted to a subset of the goal dependency relation $\mathcal{D}$, containing tuples $\mathcal{D}(A, B)$ where B is a call occurring at one of a specified program points.

Basing our approach on a downwards closed semantics allows a straightforward approach to implementation, using for example the framework presented in [GBS95]. Our analyses are based on the c-semantics [Cla79]. Given a program P, let $\mathcal{C}(P)$ be the c-semantics of P, which contains the set of atomic logical consequences of P.

3 The Program Transformation

First, the resultants semantics is formulated as a program transformation.

3.1 Resultants Semantics by Program Transformation

A resultant $A \leftarrow Q$ is represented as a meta-predicate $\mathcal{R}(A, Q)$. Let P be a program. For each program clause $H \leftarrow D_1, \ldots, D_n$ ($n > 0$) in P we produce n clauses.

$$\begin{array}{l}\mathcal{R}(H, (Q, D_2, \ldots, D_n)) \leftarrow \mathcal{R}(D_1, Q)\\ \mathcal{R}(H, (Q, D_3, \ldots, D_n)) \leftarrow D_1, \mathcal{R}(D_2, Q)\\ \cdots\\ \mathcal{R}(H, Q) \leftarrow D_1, \ldots, D_{n-1}, \mathcal{R}(D_n, Q)\end{array}$$

For each unit clause $H \leftarrow \mathsf{true}$ produce a single clause $\mathcal{R}(H, \mathsf{true}) \leftarrow \mathsf{true}$. Finally, for each predicate p we add a clause $\mathcal{R}(p(\bar{x}), p(\bar{x}))$ where $p(\bar{x})$ is a most general call to p.

In the bodies of the clauses for $\mathcal{R}$ there are calls to the original program atoms D_1, D_2 and so on, so it is assumed that the clauses for P are included in the transformed program. These object program calls could have been written $\mathcal{R}(D_1, \mathsf{true}), \mathcal{R}(D_2, \mathsf{true})$ respectively since A is in the minimal model of the program iff there is a ground instance of a resultant $A \leftarrow \mathsf{true}$ in the resultants semantics of the program. If this modification were made, the transformation corresponds closely to the fixpoint definition of the resultants semantics [GLM96].

We denote by Res_P the collection of clauses defining the predicate $\mathcal{R}$ as shown above, together with P itself.

Proposition 3.1 *Let P be a program. Then for all resultants $A \leftarrow G \in \mathcal{O}^+(P)$, $\mathcal{R}(A, G) \in \mathcal{C}(\mathsf{Res}_P)$.*

Proof 3.2 *(Outline). A derivation corresponding to a resultant can be represented as an AND-OR proof tree. The proof is by induction on the depth of AND-OR trees.*

Note that $\mathcal{C}(\mathsf{Res}_P)$ contains more instances of resultants than does $\mathcal{O}^+(P)$. Specifically, local variables in resultants are also instantiated, as well as head variables. The transformed program thus represents an approximation of the dependency relation. In practice this is not a loss in precision, since clearly no dependencies will be derived between local variables in resultants and head variables.

3.2 From Resultants to Binary Clauses

The program above can be modified to yield (the downwards closure of) binary clauses [CT99]. Only the first call in the right-hand-side of the resultants is recorded, rather than the whole resultant. A resultant $A_1 \leftarrow A_2$ in which both A_1 and A_2 are atoms is called a *binary clause*. In the binary clause semantics, a resultant $A \leftarrow B_1, \ldots, B_n$ is abstracted to $A \leftarrow B_1$.

The transformed program corresponding to the binary clauses is as follows. A meta-predicate $\mathcal{B}(A_1, A_2)$ represents the binary resultant $A_1 \leftarrow A_2$.

$$\begin{array}{l}
\mathcal{B}(H,Q) \leftarrow \mathcal{B}(D_1,Q).\\
\mathcal{B}(H,Q) \leftarrow D_1, \mathcal{B}(D_2,Q).\\
\cdots\\
\mathcal{B}(H,Q) \leftarrow D_1, \ldots, D_{n-1}, \mathcal{B}(D_n,Q).
\end{array}$$

As before, we add a clause $\mathcal{B}(p(\bar{x}), p(\bar{x}))$ for each predicate p where $p(\bar{x})$ is a most general call to p. Note that a unit clause in P produces no clauses for $\mathcal{B}$. Let Bin_P be the transformed program consisting of P together with the clauses defining the predicate $\mathcal{B}$ as shown above.

Proposition 3.3 *Let P be a program. Then for all resultants $A \leftarrow B_1, \ldots, B_n \in \mathcal{O}^+(P)$, $\mathcal{B}(A, B_1) \in \mathcal{C}(\mathsf{Bin}_P)$.*

$\mathcal{C}(\mathsf{Bin}_P)$ is an over approximation of the goal dependency relation for P. As was the case for the resultants program Res_P, the downwards closure of local variables is included in the relation $\mathcal{B}$ in $\mathcal{C}(\mathsf{Bin}_P)$.

3.3 Transforming with Respect to Program Points

Next, a further simplification is made, when calls at specified program points are to be observed, rather than all calls. A meta-predicate $\mathsf{Dep}(A_1, A_2)$ is defined, whose meaning is that there is a clause $A_1 \leftarrow A_2$ in the binary clause semantics, and A_2 is a call at one of the specified program points to be observed.

Let $H \leftarrow B_1, \ldots, B_j, \ldots, B_n$ be a clause in a program P. Suppose that we wish to observe calls to B_j in this clause body, and determine some property of initial goals which establish some property of B_j. In the semantics, only the binary clauses of the form $A \leftarrow B_j$ are to be observed: no other calls other than those to B_j need be recorded.

To achieve this, we simply modify the binary clause transformation shown above. Specifically, instead of the clauses of form $\mathcal{B}(p(\bar{x}), p(\bar{x}))$, we create base case clauses for the given program points.

For instance, for the clause $H \leftarrow D_1, \ldots, D_j, \ldots, D_n$ with one point D_j to be observed, the following clauses for Dep are generated.

$$
\begin{array}{ll}
\mathsf{Dep}(H, D_j) \leftarrow D_1 \ldots, D_{j-1} & \mathsf{Dep}(H, Q) \leftarrow \mathsf{Dep}(D_1, Q). \\
 & \mathsf{Dep}(H, Q) \leftarrow D_1, \mathsf{Dep}(D_2, Q). \\
 & \ldots \\
 & \mathsf{Dep}(H, Q) \leftarrow D_1, \ldots, D_{n-1}, \mathsf{Dep}(D_n, Q).
\end{array}
$$

For each body atom to be observed, we add one clause similar to the one for D_j above. We can see that the only atoms that can appear in the second argument of Dep are instances of D_j. Denote by Dep_P the transformed program consisting of P together with the clauses defining Dep as shown above.

Proposition 3.4 *Let P be a program, and $\{D_{j_1}, \ldots, D_{j_k}\}$ be a set of body atoms from clauses in P. Let Dep_P be the transformed program consisting of P together with the clauses defining* Dep *as shown above. Then for all resultants $A \leftarrow D_{j_i}, \ldots \in \mathcal{O}^+(P)$, where D_{j_i} is an instance of one of the specified atoms, $\mathsf{Dep}(A, D_{j_i}) \in \mathcal{C}(\mathsf{Dep}_P)$.*

The transformation can be refined (with respect to computational efficiency) by having a separate Dep predicate corresponding to each predicate in P. That is, each occurrence of $\mathsf{Dep}(p(\bar{t}), Q)$ in the transformed program is replaced by $\mathsf{Dep}_p(\bar{t}, Q)$.

The transformation can be varied by observing in the second argument of Dep not the actual call, but simply one or more variables from the call. This is illustrated in the next example.

Example 3.5 *Let P be the "naive reverse" program. Suppose the call that we wish to observe is* `app(Ys,[X],Zs)` *in the recursive clause for* `rev` *as shown in Figure 11. For example, we suppose that we require that* `integer(X)` *holds whenever this call is encountered. However, the transformation is independent of the actual property. The transformed program, shown in Figure 11, consists of P together with the clauses defining* `drev/2` *and* `dapp/3` *(representing the meta-predicates Dep_{rev} and Dep_{app}). In place of the call* `app(Ys,[X],Zs)` *in the final argument, we observe only the variable* `X`*.*

Next, we apply standard static analysis techniques to the transformed program.

```
drev([X|Xs],Zs,X) :-
  rev(Xs,Ys).
drev([X|Xs],Zs,Q) :-
  drev(Xs,Ys,Q).
drev([X|Xs],Zs,Q) :-
  rev(Xs,Ys), dapp(Ys,[X],Zs,Q).
dapp([X|Xs],Ys,[X|Zs],Q) :-
  dapp(Xs,Ys,Zs,Q).
```

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

Figure 11: Transformed Naive Reverse Program for Backwards Analysis

3.4 Analysis of the Transformed Programs

The transformed program can be input to an abstract interpretation framework. In the experiments carried out so far, analysis was based on the c-semantics abstracted using pre-interpretations [GBS95]. A pre-interpretation is a mapping from terms into a (finite) domain D, defined by a pre-interpretation function J. For each n-ary function symbol f, J contains a function $D^n \rightarrow D$, written $J(f(d_1, \ldots, d_n)) = d$ for $d_1, \ldots, d_n, d \in D$. A mapping α is defined inductively as $\alpha(c) = d$ where $J(c) = d$, for 0-ary functions c, and $\alpha(f(t_1, \ldots, t_n)) = J(f(\alpha(t_1), \ldots, \alpha(t_n)))$ for terms with functions of arity greater than 0. An abstract "domain program" is generated by abstract compilation, in the style introduced by Codish and Demoen [CD93]. A bottom-up analysis of the domain program yields its c-semantics. Let P be a program and $\mathcal{C}(P)$ its minmal model, which is identical to the c-semantics in this case. Let P^J be the abstract domain program for some pre-interpretation J. The safety result is that for all atoms $p(t_1, \ldots, t_n) \in \mathcal{C}(P)$, $p(\alpha(t_1), \ldots, \alpha(t_n)) \in \mathcal{C}(P^J)$.

Example 3.6 *We analyse the above example where we wish to establish the property dependency of the property* `app(Ys,[X],Zs)` $\leftrightarrow$`integer(X)`. *A simple type domain could be used, consisting of the types* int, listint, other. *We construct an abstract "domain program" as described in [GBS95], based on the pre-interpretation constructed from the program's function symbols and the given types.*

$[\,] \longrightarrow \mathsf{listint}$	$[\mathsf{int} \mid \mathsf{other}] \longrightarrow \mathsf{other}$	$[\mathsf{other} \mid \mathsf{other}] \longrightarrow \mathsf{other}$
$[\mathsf{listint} \mid \mathsf{other}] \longrightarrow \mathsf{other}$	$[\mathsf{int} \mid \mathsf{int}] \longrightarrow \mathsf{other}$	$[\mathsf{listint} \mid \mathsf{int}] \longrightarrow \mathsf{other}$
$[\mathsf{other} \mid \mathsf{int}] \longrightarrow \mathsf{other}$	$[\mathsf{int} \mid \mathsf{listint}] \longrightarrow \mathsf{listint}$	$[\mathsf{listint} \mid \mathsf{listint}] \longrightarrow \mathsf{other}$
$[\mathsf{other} \mid \mathsf{listint}] \longrightarrow \mathsf{other}$		

The pre-interpretation is encoded as a predicate $\rightarrow$`/2` *corresponding to the pre-interpretation.*

```
rev(X1,X2):-
  []→X1,[]→X2.
rev(X1,X2):-
  rev(X3,X4),app(X4,X5,X2),
  [X6|X3]→X1,[]→X7,[X6|X7]→X5.
app(X1,X2,X2):-
  []→X1.
app(X1,X2,X3):-
  app(X4,X2,X5),[X6|X4]→X1,[X6|X5]→X3.
drev(X1,X2,X3):-
  rev(X4,X5),[X3|X4]→X1.
drev(X1,X2,X3):-
  rev(X4,X5),dapp(X5,X6,X2,X3),
  [X7|X4]→X1,[]→X8,[X7|X8]→X6.
drev(X1,X2,X3):-
  drev(X4,X5,X3),[X6|X4],→X1.
dapp(X1,X2,X3,X4):-
  dapp(X5,X2,X6,X4),[X7|X5]→X1,[X7|X6]→X3.
```

Figure 12: Domain Program for Backwards Analysis of Naive Reverse

app(listint,X1,X1)	rev(listint,listint)	drev(listint,X1,int)
app(listint,int,other)	rev(other,other)	drev(other,X1,int)
app(other,other,other)		drev(other,X1,listint)
app(other,int,other)		drev(other,X,other)
app(other,listint,other)		

Figure 13: Least model of program in Figure 12, over domain of simple types

The domain program is shown in Figure 12. Its least model over the pre-interpretation for the domain of simple types is shown in Figure 13.

3.5 Interpretation of the Analysis Result

Examining the results in Figure 13, we see a number of abstract facts for `drev`. (There are no results for `dapp` derived since no call to `app` affects the given program point.) The results show that whenever `rev(X,Y)` is called with `X` a list of integers, then `X` is an integer at the given program point. This is indicated by the fact that $\mathtt{drev}(\mathsf{listint}, \mathtt{X1}, \mathsf{int})$ is in the model of the abstract program, and there are no other tuples $\mathtt{drev}(\mathsf{listint}, \mathtt{X1}, \mathtt{Y})$ where $\mathtt{Y} \neq \mathsf{int}$. By contrast, there is a tuple $\mathtt{drev}(\mathsf{other}, \mathtt{X1}, \mathsf{int})$ but there is also a tuple $\mathtt{drev}(\mathsf{other}, \mathtt{X1}, \mathsf{listint})$, so although goals of the form `rev(other,Y)` *might* establish the property, they are not *guaranteed* to establish it.

In terms of the discussion in Section 2.2, the goal dependency $\Theta \rightarrow \Phi$ follows from the abstract relation, where $\Theta(\mathtt{rev}(\mathtt{X},\mathtt{Y}))$ is true if `X` is a list of integers, and $\Phi(\mathtt{app}(\mathtt{Ys},[\mathtt{X}],\mathtt{Zs}))$ is true is this call arises from the specified program point, and `X` is an integer.

Example 3.7 *Let P be the* quicksort *program, for which backwards analysis was considered in [KL02b]. Suppose we wish to check the calls to the built-in predicates $\geq$ and $<$. The intention is that these predicates require their argument to be ground when called in order to prevent run-time instantiation errors. The transformed* quicksort *program, which includes the original clauses for* quicksort, *is shown in Figure 14.*

3.6 Analysis of Quicksort

We perform groundness analysis on the program in Figure 14. A pre-interpretation over the domain elements `g` and `ng` (standing for *ground* and *non-ground*) is constructed. This is equivalent

```
qsort([],Ys,Ys).
qsort([X|Xs],Ys,Zs) :-
  partition(Xs,X,Us,Vs),
  qsort(Us,Ys,[X|Ws]),
  qsort(Vs,Ws,Zs).
partition([],Z,[],[]).
partition([X|Xs],Z,Ys,[X|Zs]) :-
  X ≥ Z, partition(Xs,Z,Ys,Zs).
partition([X|Xs],Z,[X|Ys],Zs) :-
  X < Z, partition(Xs,Z,Ys,Zs).
```

```
dqsort([X|Xs],Ys,Zs,Q) :-
  dpartition(Xs,X,Us,Vs,Q).
dqsort([X|Xs],Ys,Zs,Q) :-
  partition(Xs,X,Us,Vs),
  dqsort(Us,Ys,[X|Ws],Q).
dqsort([X|Xs],Ys,Zs,Q) :-
  partition(Xs,X,Us,Vs),
  qsort(Us,Ys,[X|Ws]),
  dqsort(Us,Ys,[X|Ws],Q).
dpartition([X|Xs],Z,Ys,[X|Zs],X ≥ Z).
dpartition([X|Xs],Z,Ys,[X|Zs],Q) :-
  X ≥ Z, dpartition(Xs,Z,Ys,Zs,Q).
dpartition([X|Xs],Z,[X|Ys],Zs,X<Z).
dpartition([X|Xs],Z,[X|Ys],Zs,Q) :-
  X < Z, dpartition(Xs,Z,Ys,Zs,Q).
```

Figure 14: Transformed Quicksort Program for Backwards Analysis

to the Pos boolean domain.

$$[] \longrightarrow \mathsf{g} \quad [\mathsf{g} \mid \mathsf{g}] \longrightarrow \mathsf{g} \quad [\mathsf{g} \mid \mathsf{ng}] \longrightarrow \mathsf{ng} \quad [\mathsf{ng} \mid \mathsf{g}] \longrightarrow \mathsf{ng} \quad [\mathsf{ng} \mid \mathsf{ng}] \longrightarrow \mathsf{ng}$$

After generating the domain program, the least model is computed and is shown in Figure 15. (When computing the minimal model we assign the success modes `g≥g` and `g<g` to the built-ins).

Examining the results via the relation `dqsort`, we see that the only calls to `qsort(X,Y,Z)` that guarantee that the required groundness properties `g≥g` and `g<g` are those in which `X` is ground. The arguments `Y` and `Z` are completely independent of the property. For `dpartition`, note that a variable `X1` occurs in both the final argument of `dpartition` and in the second argument of `partition`. This variable can be instantiated by g or ng. Thus the second argument of `partition` has to be ground to establish `g≥g` and `g<g`. In addition, the arguments of $\geq$ and $<$ are ground if either the first argument of `partition` or the third and fourth are ground. These are the same results reported by King and Lu [KL02b], summarised as $X_2 \wedge (X_1 \vee (X_3 \wedge X_4))$ in the notation of Pos, where $X_1, \ldots, X_4$ are the arguments of `partition`.

3.7 Computing the Goal Conditions

For examples such as the ones discussed above, the required properties of the input goals that guarantee the observed property were derived informally by examining the abstract tuples. We

```
partition(g,X1,g,g)                 qsort(g,X1,X1)
                                    qsort(ng,ng,g)
                                    qsort(ng,ng,ng)

dpartition(ng,X1,ng,X2,g<X1)        dqsort(ng,X1,X2,ng≥ng)
dpartition(ng,X1,g,X2,g<X1)         dqsort(ng,X1,X2,ng≥g)
dpartition(ng,X1,ng,X2,ng<X1)       dqsort(ng,X1,X2,g≥ng)
dpartition(g,X1,ng,X2,g<X1)         dqsort(ng,X1,X2,ng<ng)
dpartition(g,X1,g,X2,g<X1)          dqsort(ng,X1,X2,ng<g)
dpartition(ng,X1,X2,ng,g≥X1)        dqsort(ng,X1,X2,g<ng)
dpartition(ng,X1,X2,g,g≥X1)         dqsort(g,X1,X2,g≥g)
dpartition(ng,X1,X2,ng,ng≥X1)       dqsort(g,X1,X2,g<g)
dpartition(g,X1,X2,ng,g≥X1)         dqsort(ng,X1,X2,g≥g)
dpartition(g,X1,X2,g,g≥X1)          dqsort(ng,X1,X2,g<g)
```

Figure 15: Least model of program in Figure 14, over groundness domain

now explain how to do this systematically.

Let Dep(A, B) be the abstract dependency relation returned by the analysis, which is a finite set of tuples. Let Φ be the property required in the call; that is, we seek calls B where $\Phi(B)$ is true. Consider the set $S = \{A \mid \text{Dep}(A, B) \wedge \Phi(B)\}$. S is the set of calls that *possibly* establishes $\Phi(B)$. Now consider candidate properties Θ that hold for all elements of S. For each such property, check whether there exists Dep(A, B) such that $\Theta(A)$ and $\neg\Phi(B)$. If there is, the candidate property is eliminated. For all other candidate properties, we have established that $\Theta \rightarrow \Phi$ follows from the abstract dependency relation.

We illustrate this process for the $quicksort$ example. Consider the relation `dqsort` shown in Figure 15. The required property is that $\Phi(\text{g} \geq \text{g})$ and $\Phi(\text{g} < \text{g})$ are true and Φ is false for all other arguments of $\geq$ and $<$. The tuples in the abstract `dqsort` relation in which Φ holds are the following.

```
dqsort(g,X1,X2,g≥g)
dqsort(g,X1,X2,g<g)
dqsort(ng,X1,X2,g≥g)
dqsort(ng,X1,X2,g<g)
```

A candidate property is then that the first argument of `qsort` can be either `g` or `ng`, to establish the required property. However, we can search the relation to find a counterexample to the candidate property that the first argument is ng, such as dqsort(ng,X1,X2,ng<g). However

we can find no counterexample to the property that the first argument is `g`. Hence we have established that `qsort(g,X1,X2)` $\rightarrow \Phi$.

3.8 The Relative Pseudo-Complement

Domains which possess a relative pseudo-complement allow a more direct method. Giacobazzi and Scozzari [GS98] identified a property of abstract domains that allows analyses to be reversible. This property is central to the approach of King and Lu [KL02b, KL03]. The key property is that the domain possesses a *relative pseudo-complement* operator. We quote the definition as given by King and Lu. Let D be an abstract domain with meet and join operations $\sqcap$ and $\sqcup$. Let d_1, d_2 be elements of D. The pseudo-complement of d_1 relative to d_2, denoted $d_1 \Rightarrow d_2$ is the greatest element whose meet with d_1 is less than d_2: that is, $d_1 \Rightarrow d_2 = \sqcup\{d \in D \mid d \sqcap d_1 \sqsubseteq d_2\}$.

To take Example 3.7 again, treat g and ng as true and false respectively. The set of abstract tuples for say, `dpartition` in Figure 15, can be rewritten as the following boolean expression, in the domain Pos, which possesses a relative pseudo-complement operation (here $q(X,Y)$ means $X \geq Y \wedge X < Y$).

$$\begin{array}{l}
\mathtt{dpartition}(X_1, X_2, X_3, X_4, q(X_5, X_6)) \equiv \\
(X_2 \leftrightarrow X_6) \wedge ((\bar{X_1} \wedge \bar{X_3} \wedge X_5) \vee (\bar{X_1} \wedge X_3 \wedge X_5) \vee (\bar{X_1} \wedge \bar{X_3} \wedge \bar{X_5}) \vee \\
(X_1 \wedge \bar{X_3} \wedge X_5) \vee (X_1 \wedge X_3 \wedge X_5) \vee (\bar{X_1} \wedge \bar{X_4} \wedge X_5) \vee (\bar{X_1} \wedge X_4 \wedge X_5) \vee \\
(\bar{X_1} \wedge \bar{X_4} \wedge \bar{X_5}) \vee (X_1 \wedge \bar{X_4} \wedge X_5) \vee (X_1 \wedge X_4 \wedge X_5))
\end{array}$$

The pseudo-complement of the above boolean expression relative to the desired property $X_5 \wedge X_6$ gives $X_2 \wedge (X_1 \vee (X_3 \wedge X_4))$, which is equivalent to the result derived in Example 3.7, and the same as that reported by King and Lu [KL02b] for this predicate.

4 Related Work

The most closely related work is that of King and Lu [KL02b, KL03], who describe a method for backwards analysis of logic programs, and report results for the domain of ground and non-ground modes. Their results have all been reproduced by the technique shown above, but a formal proof of equivalence has not yet been constructed. Their approach requires the construction of an abstract interpretation which under-approximates the concrete semantics. This requires the definition of a universal projection operator, and requires a condensing domain possessing a relative pseudo-complement operator. The fixpoint computation uses a greatest fixpoint rather than the standard least fixpoint. Our approach appears to be more flexible in the sense that a wide variety of domains can be used for the analysis, not only condensing domains. The relative pseudo-complement, if it exists, can be used in our approach to extract the result from the abstract program, but is not essential.

Mesnard *et al.* [Mes96, MN01] have also performed termination inference, which is a form of backwards analysis.Their approach uses a greatest fixpoint, and in this respect seems to align more with the approach of King and Lu.

The *binary clause* semantics of Codish and Taboch [CT99] was used to make loops observable, by deriving an explicit relationship between a calls and its successor calls. The transformation presented here can be targeted to observe any program points of interest, not only loops, but the spirit of the approach is the same. In later work based on binary clause semantics, Genaim and Codish [GC01] perform termination inference which involves backwards analysis. However, they use the framework of King and Lu for the backwards analysis, rather than the binary clause semantics.

Binary clause semantics is derived from the more general and expressive resultants semantics [GLM96, GG94]. We do not know of any implemented applications of resultants semantics, apart from the present work and that of [CT99, GC01], nor any previous suggestion that resultants semantics could form the basis for backwards analysis.

The approach of transforming programs to realise non-standard semantics is also followed in the *query-answer* transformations, which include magic-set transformations and its relations [DR94, BMSU86]. There, the aim is to simulate a top-down goal-directed computation, in a bottom-up semantic framework. A related approach is advocated by Codish and Søndergaard [CS02]. Different semantics for logic programs can be represented by meta-interpreters, which are also written as logic programs. Codish and Genaim's implementation of the binary semantics [GC01] follows this style.

5 Conclusion

A method for backwards analysis of logic programs has been presented. Given a program, and one or more specified body calls, a program transformation is performed. In the transformed program, the dependencies between the selected calls and initial goals is made explicit. Analysis of the transformed program using abstract interpretation yields an over-approximation of the dependency relation, and it was proved that dependencies could safely be derived from the approximation.

In contrast to previous work on backwards analysis, our approach requires no special properties of the abstract domain, nor any non-standard operations such as universal projection, or a greatest fixpoint computation. This is put forward as an advantage of our approach, since implementations can be based on existing abstract interpretation tools.

Experimental results carried out so far indicate that this method is of similar complexity to other reported work on backwards analysis, and gives equivalent precision at least over the Boolean domain POS. A detailed analytical comparison is difficult due to the great differences between the two approaches. It is indeed quite surprising that two such different algorithms yield the same results in experiments carried out so far.

Our use of downwards closed semantics does not seem to be essential to our general approach, but does allow a simpler analysis and implementation.

Part V

Partial deduction of real-life CLP programs containing impure predicates using backwards analysis

1 Summary

Partial deduction is a program transformation technique which specializes a program w.r.t. its static data. If the program contains *impure* predicates, it is known that unfolding steps for atoms which are not leftmost is problematic. Impure predicates include those which may raise errors, exceptions or side-effects, external predicates whose definition is not available, etc. Existing proposals allow obtaining correct residual programs while still allowing non-leftmost unfolding steps, but at the cost of accuracy: bindings and failure are not propagated backwards to predicates which are classified as impure. Motivated by recent developments in the *backwards* analysis of logic programs, we propose a partial deduction algorithm which can handle impure features and non-leftmost unfolding in a more accurate way. We outline by means of examples some optimizations which are not feasible using existing partial deduction techniques. We argue that our proposal goes beyond existing ones and is a) accurate, since the classification of pure vs impure is done at the level of atoms instead of predicates, b) flexible, as the user can annotate programs using assertions, which can guide the partial deduction process, and c) automatic, since backwards analysis can be used to automatically infer the required assertions. Our approach has been implemented in the context of `CiaoPP`, the abstract interpretation-based preprocessor of the `Ciao` logic programming system.

2 Background

We assume some basic knowledge on the terminology of logic programming. See for example [Llo87b] for details. Very briefly, an *atom* A is a syntactic construction of the form $p(t_1, \ldots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \ldots, t_n$ are terms. The function $pred$ applied to atom A, i.e., $pred(A)$, returns the predicate symbol p/n for A. A *clause* is of the form $H \leftarrow B$ where its head H is an atom and its body B is a conjunction of atoms. A *definite program* is a finite set of clauses. A *goal* (or query) is a conjunction of atoms. The concept of *computation rule* is used to select an atom within a goal for its evaluation. The operational semantics of definite programs is based on derivations. Consider a program P and a goal G of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule such that $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause in program P. Then $\theta(A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k)$ is *derived* from G and C via $\mathcal{R}$ where $\theta = mgu(A_R, H)$. An *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees*. A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer for goal G. Such a derivation is called *failed* if it is not possible to perform a derivation step with G_n. We will also allow *incomplete* derivations in which, though possible, no further resolution step is performed. We refer to SLD resolution restricted to the case of leftmost unfolding as LD resolution.

Partial Deduction (PD) [LS91, Gal93] is a program transformation technique which specializes a program w.r.t. part of its known input data. Hence sometimes also known as program specialization. Informally, given an input program and a set of atoms, the PD algorithm applies an *unfolding rule* in order to compute finite (possibly incomplete) SLD trees for these atoms. This process returns a set of *resultants* (or residual rules), i.e., a residual program, associated to the root-to-leaf derivations of these trees. Each unfolding step during partial deduction can be conceptually divided into two steps. First, given a goal $\leftarrow A_1, \ldots, A_R, \ldots, A_k$ the computation rule determines the selected atom A_R. Second, it must be decided whether unfolding (or evaluation) of A_R is *profitable*. It must be noted that the unfolding process requires the introduction of this profitability test in order to guarantee that unfolding terminates. Also, unfolding usually continues as long as some evidence is found that further unfolding will improve the quality of the resultant program.

Most of real-life Prolog programs use predicates which are not defined in the program (module) being developed. We will refer to such predicates as *external*. Examples of external predicates are traditional "built-in" predicates such as arithmetic operations (e.g., `is/2`, `<`, `=<`, etc.), basic input/output facilities, and predicates defined in libraries. We will also consider as external predicates those defined in a different module, predicates written in another language, etc. The trivial computation rule which always returns the leftmost atom in a goal is interesting in that it avoids several correctness and efficiency issues in the context of PD of full Prolog programs. Such issues are discussed in depth throughout this extended abstract. When a (leftmost) atom A_R is selected during PD, with *pred*(A_R) $= p/n$ being an external predicate, it may not be possible to unfold A_R for several reasons. First, we may not have the code defining p/n and, even if we have it, unfolding A_R may introduce in the residual program calls to predicates which are private to the module where the p/n is defined. Also, it can be the case that the execution of atoms for (external) predicates produces other outcomes such as side-effects, errors, and exceptions. Note that this precludes the evaluation of such atoms to be performed at PD time, since those effects need to be performed at run-time. In spite of this, if the executable code for the external predicate p/n is available, and under certain conditions, it can be possible to fully evaluate A_R at specialization time. The notion of *evaluable* atom [PAH05b] captures the requirements which allow executing external predicates at PD time. Informally, an atom is evaluable if its execution satisfies four conditions: 1) it universally terminates, 2) it does not produce side-effects, 3) it does not issue errors and 4) it is binding insensitive. We use $\mathsf{eval}(E)$ to denote that the expression E is evaluable. We will discuss all these properties in depth in Section 4.

Since some of the above properties are not computable (e.g., termination is undecidable in the general case), [PAH05b] proposes to determine *sufficient conditions* (SC) which are *decidable* and ensure that, if the atom satisfies such conditions, then it is evaluable. To formalize this, "*computational* assertions" –which are part of the assertion language [PBH00b] of `CiaoPP` [HPBLG05]– express that a certain predicate is evaluable under certain conditions.

The following definition recalls the notion of an eval *annotation* from [PAH05b] as (part of) a computational assertion. We use $\mathsf{Exec}(Sys, M, A)$ to denote that the execution of atom A on a logic programming system Sys (e.g., `Ciao` or Sicstus) in which the module M where the external predicate p/n is defined has been loaded. In the case of logic programs, the value of $\mathsf{Exec}(Sys, M, A)$ is a pair consisting of a possibly empty set of computed answers for $M \cup A$ together with and indicator of whether the computation terminates or (possibly) loops. In particular we say that $\mathsf{Exec}(Sys, M, A)$ *trivially succeeds*, written as triv_suc, when it returns a set containing only the empty computed answer and a termination indicator.

Definition 2.1 (eval annotations) *[PAH05b] Let p/n be an external predicate defined in module M. The assertion "*`:- trust comp p(X1,...,Xn) :  SC + eval.`*" in the code for M is a correct* eval annotation *for predicate p/n in a logic programming system Sys if, $\forall$ A s.t. $A = \theta(p(X_1, \ldots, X_n))$,*

1. $\mathsf{eval}(\theta(SC))$*, and*

2. $\mathsf{Exec}(Sys, M, \theta(SC))$ *trivially succeeds* $\Rightarrow \mathsf{eval}(A)$.

3 Non-Leftmost Unfolding in Partial Deduction

It is well-known that *non-leftmost* unfolding is essential in partial deduction in some cases for the satisfactory propagation of static information (see, e.g., [LB02]). Informally, given a goal $\leftarrow A_1, \ldots, A_n$, it can happen that the profitable criterion does not hold for the leftmost atom A_1. For example, if A_1 is an atom for an internal predicate, it might not be profitable to select A_1 because 1) unfolding A_1 endangers termination (for example, A_1 may homeomorphically embed [Leu98] some selected atom in its sequence of covering ancestors), or 2) the atom A_1 unifies with several clause heads (for example, some unfolding rules do not unfold non-deterministically for atoms other than the initial query). If A_1 is an atom for an external predicate, it can happen that A_1 is not sufficiently instantiated so as to be executed at this moment. It may nevertheless be profitable to unfold atoms other than the leftmost. Therefore, it can be interesting to define a computation rule which is able to detect the above circumstances and "jump over" atoms whose profitability criterion is not satisfied in order to proceed with the specialization of another atom in the goal as long as it is correct.

3.1 Non-Leftmost Unfolding and Impure Predicates

For pure logic programs without builtins, non-leftmost unfolding is safe thanks to the independence of the computation rule (see for example [Llo87b]).[13] Unfortunately, non-leftmost unfolding poses several problems in the context of *full* Prolog programs with *impure* predicates, where such independence does not hold anymore.

For instance, `var/1` is an *impure* predicate since, under LD resolution, `var(X),X=a` succeeds with computed answer *X/a* whereas `X=a,var(X)` fails. They are not equivalent since the independence of the computation rule does not hold. Thus, given the goal $\leftarrow$ `var(X),X=a`, if we allow the non-leftmost unfolding step which binds the variable `X`, the goal will fail, either at specialization time or at run-time, whereas the initial goal succeeds in LD resolution. The above problem was early detected [Sah93] and it is known as the problem of *backpropagation of bindings*. In addition to this, it is also problematic the *backpropagation of failure* in the presence of impure predicates. There are atoms A for impure predicates such that $\leftarrow A, fail$ behaves differently from $\leftarrow fail$. For instance, we have to ensure that failure to the right of a call to `write` does not prevent the generation of the residual call to `write` nor its execution at runtime.

There are satisfactory solutions in the literature (see, e.g.,[Leu94, EGM97, AHV02, LB02]) which allow unfolding non-leftmost atoms while avoiding the backpropagation of bindings and

[13]Although safe, non-leftmost unfolding presents problems with pure programs too since it may introduce extra backtracking over the atoms to the left. We are not concerned with such efficiency issues here.

	pure			
	eval			
predicate	sideff_free	error_free	bind_ins	termin
var(X)	true	true	nonvar(X)	true
nonvar(X)	true	true	nonvar(X)	true
write(X)	false	true	ground(X)	true
assert(X)	false	nonvar(X)	ground(X)	true
A is B	true	arithexp(B)	true	true
A <= B	true	arithexp(A)$\wedge$arithexp(B)	ground(A)$\wedge$ground(B)	true
A >= B	true	arithexp(A)$\wedge$arithexp(B)	ground(A)$\wedge$ground(B)	true
ground(X)	true	true	ground(X)	true
A = B	true	true	true	true
append(A,B,C)	true	true	true	list(A)$\vee$list(C)

Table 14: Purity conditions for some predefined predicates.

failure. Basically, the common idea is to represent explicitly the bindings by using unification [Leu94] or residual case expressions [AHV02] rather than backpropagating them (and thus applying them onto leftmost atoms). This guarantees that the resulting program is correct, but it definitely introduces some inaccuracy, since bindings (and failure) generated during unfolding of non-leftmost atoms are hidden from atoms to the left of the selected one. It should be noted that preventing backpropagation by introducing equalities can be a bad idea from the performance point of view too (see, e.g., [VD88]). Thus, these solutions should be applied only when it is really necessary, since backpropagation can 1) lead to early detection of failure, which may result in important speedups and 2) make the profitability criterion for the leftmost atom to hold, which may result in more aggressive unfolding. Thus, if backpropagation is disabled, some interesting specializations can no longer be achieved.

It should also be noted that the backpropagation problem is very much related to that of *reordering* of atoms within a goal. Such reordering transformation can be of interest for achieving powerful optimizations like tupling, for effectively handling the conjunction of atoms like conjunctive PD [DSGJ$^+$99] and for the use of efficient stack-based unfolding rules [PAH05b].

4 From Impure Predicates to Impure Atoms

As mentioned in Section 3.1 above, existing techniques for PD allow the unfolding of non-leftmost atoms by combining a classification of predicates into pure and impure with techniques for avoiding backpropagation of binding and failure in the case of impure predicates. In order to classify predicates as pure or impure, existing methods [LB02] are based on simple reachability analysis. As soon as an impure predicate p can be reached from a predicate q, also q is considered impure and backpropagation is not allowed. In other words, impurity is defined at the level of predicates. Unfortunately, this notion of impurity quickly expands from a predicate to all predicates which use it.

Our work improves on existing techniques by providing a more refined notion of impurity. Rather than being defined at the level of predicates, we define purity at the level of individual atoms. This is of interest since it is often the case that some atoms for a predicate are pure whereas others are impure. As an example, the atom $var(X)$ is impure (binding sensitive), whereas the atom $var(f(X))$ is not (it is no longer binding sensitive). This allows *reducing* substantially the situations in which backpropagation has to be avoided. In the following, we characterize three different classes of impurities: binding-sensitiveness, errors and side effects.

4.1 Binding-sensitiveness

A *binding-sensitive* predicate is characterized by having a different success or failure behaviour under leftmost execution if bindings are backpropagated onto it. Examples of binding-sensitive predicates are `var/1, nonvar/1, atom/1, number/1, ground/1, ...`. However, rather than considering all atoms for such predicates as binding-sensitive, we propose to define binding sensitiveness at the atom level. The reason is that the fact that some atoms for the predicates above are indeed binding sensitive does not necessarily mean that all atoms for such predicates are. As we have seen above, the atom $var(f(X))$ is certainly not binding sensitive since its truth value is not changed by applying any substitution, i.e., the atom will not succeed in any context.

Definition 4.1 (binding insensitive atom) *An atom A is* binding insensitive*, denoted* $\mathsf{bind\_ins}(A)$*, if $\forall$ sequence of variables $\langle X_1, \ldots, X_k \rangle$ s.t. $X_i \in vars(A)$, $i = 1, \ldots, k$ and $\forall$ sequence of terms $\langle t_1, \ldots, t_k \rangle$, the goal $\leftarrow (X_1 = t_1, \ldots, X_k = t_k, A)$ succeeds in LD resolution with computed answer σ iff the goal $\leftarrow (A, X_1 = t_1, \ldots, X_k = t_k)$ also succeeds in LD resolution with computed answer σ.*

Let us note that in the definition above we are only concerned with successful derivations, which we aim at preserving. However, we are not in principle concerned about preserving infinite failure. For example, $\leftarrow (A, X = t)$ and $\leftarrow (X = t, A)$ might have the same set of answers but a different termination behaviour. In particular, the former might have an infinite derivation under LD resolution while the second may finitely fail. More on this in Section 6.

If the atom contains no variables, binding insensitiveness trivially holds. The following proposition directly follows from the definition of binding insensitive atom.

Proposition 4.2 *Let A be a ground atom. Then A is binding insensitive.*

In spite of its simplicity, Proposition 4.2 can be quite useful in practice, since it may allow considering a good number of atoms as binding insensitive even if the predicate is in principle binding sensitive. All this without the need of sophisticated analyses.

4.2 Side-effects

Predicates p for which $\theta(p(X1, ..., Xn)), fail$ and $fail$ are not equivalent in LD resolution are termed as "*side-effects*" in [Sah93].

Definition 4.3 (side-effect-free atom) *An atom A is* side-effect free, *denoted* $\mathsf{sideff\_free}(A)$, *if the run-time behaviour of $\leftarrow A, fail$ is equivalent to that of $\leftarrow fail$.*

Since side-effects have to be preserved in the residual program, we have to avoid any kind of backpropagation which can anticipate failure and, therefore, hides the existing side-effect.

4.3 Run-Time Errors

There are some predicates whose call patterns are expected to be of certain type and/or instantiation state. If an atom A does not correspond to the intended call pattern, the execution of A will issue some *run-time errors*. Since we consider such run-time errors as part of the behaviour of a program, we will require that partial deduction produces program whose behaviour w.r.t. run-time errors is identical to that of the original program, i.e., run-time errors must not be introduced to nor removed from the program.

For instance, the predefined predicate `is/2` requires its second argument to be an arithmetic expression. If that is detected not to be the case at run-time, an error is issued. Clearly, backpropagation is dangerous in the context of atoms which may issue run-time errors, since it can anticipate the failure of a call to the left of `is/2` (thus omitting the error), or it can make the call to `is/2` not to issue an error (if there is some free variable in the second argument which gets instantiated to an arithmetic expression after backpropagation). The following definition introduces the notion of *error free* atom.

Definition 4.4 (error-free atom) *An atom A is* error-free, *denoted* $\mathsf{error\_free}(A)$, *if the execution of A does not issue any error.*

Somewhat surprising this condition for PD corresponds to that used in [KL02a] for computing safe call patterns. Unfortunately, the way in which errors are issued can be implementation dependent. Some systems may write error messages and continue execution, others may write error messages and make the execution of the atom fail, others may halt the execution, others may raise exceptions, etc. Though errors are often handled using side-effects, we will make a distinction between side-effects and errors for two reasons. First, side-effects can be an expected outcome of the execution, whereas run-time errors should not occur in successful executions. Second, it is often the case that predicates which contain side-effects produce them for all (or most of) atoms for such predicate. However, predicates which can generate run-time errors can be guaranteed not to issue errors when certain preconditions about the call are satisfied, i.e., when the atom is well-moded and well-typed. A practical implication of the above distinction is that simple, reachability analysis will be used for propagating side-effects at the level of predicates, whereas a more refined, atom-based classification will be used in the case of error-freeness.

4.4 Pure and Evaluable Atoms

Given the definitions of binding insensitive, side-effect free, and error free atoms, it is useful to define aggregate properties which summarize the effect of such individual properties.

Definition 4.5 (pure atom) *An atom A is* pure, *denoted* $\mathsf{pure}(A)$, *if*

$$\mathsf{bind\_ins}(A) \wedge \mathsf{error\_free}(A) \wedge \mathsf{sideff\_free}(A)$$

In order to provide a precise definition of evaluable atom, we need to introduce first the notion of terminating atom.

Definition 4.6 (terminating atom) *An atom A is* terminating, *denoted* $\mathsf{termin}(A)$, *if the LD tree for $\leftarrow A$ is finite.*

The definition above is equivalent to *universal termination*, i.e., the search for all solutions to the atom can be performed in finite time.

Definition 4.7 (evaluable atom) *An atom A is* evaluable, *denoted* $\mathsf{eval}(A)$, *if* $\mathsf{pure}(A) \wedge \mathsf{termin}(A)$.

The notion of evaluable atoms can be extended in a natural way to boolean expressions composed of conjunction and disjunctions of atoms.

Table 14 presents sufficient conditions which guarantee that the atoms for the corresponding predicates satisfy the purity properties discussed above, where *arithexp(X)* stands for `X` being an arithmetic expression. For example, unification is pure and evaluable, whereas the library predicate `append/3` is pure but only evaluable if either the first or third argument is bound to a list skeleton.

5 Assertions about Purity of Atoms

In this section, we provide the concrete syntax of the assertions we propose to use to state the conditions under which atoms for a predicate are pure. Our assertions may include *sufficient conditions* (SC) which are *decidable* and ensure that, if the atom satisfies such conditions, then it meets the property.

We say that the execution of an atom A for p/n on a logic programming system Sys (e.g., `Ciao` or Sicstus) in which the module M (where the external predicate p/n is defined) has been loaded *trivially succeeds*, denoted by $\mathsf{triv\_suc}(Sys, M, A)$, when its execution terminates and succeeds only once with the empty computed answer, that is, it performs no bindings.

Definition 5.1 (binding insensitive assertion) *Let p/n be a predicate defined in module M. The assertion "`:- trust comp p(X1,...,Xn) : SC + bind_ins.`" in the code for M is a correct* binding insensitive assertion *for predicate p/n in a logic programming system Sys if, $\forall$ A s.t. $A = \theta(p(X_1, \ldots, X_n))$,*

1. $\mathsf{eval}(\theta(SC))$*, and*
2. $\mathsf{triv\_suc}(Sys, M, \theta(SC)) \Rightarrow \mathsf{bind\_ins}(A)$.

The fourth column in Fig. 14 comprises the information stated in several binding insensitive assertions for a few predefined builtins in `Ciao`. In particular, this column represents the sufficient conditions (SC in Def. 5.1) for the predicates in the first column ($p(X1, ..., Xn)$ in Def. 5.1). For instance, the predicate `A is B` is bind_ins if `ground(B)`.

Definition 5.2 (error-free assertion) *Let p/n be a predicate defined in module M. The assertion "`:- trust comp p(X1,...,Xn) : SC + error_free.`" in the code for M is a correct* error-free assertion *for predicate p/n in a logic programming system Sys if, $\forall$ A s.t. $A = \theta(p(X_1, \ldots, X_n))$,* *1.* $\mathsf{eval}(\theta(SC))$*, and*

2. $\mathsf{triv\_suc}(Sys, M, \theta(SC)) \Rightarrow \mathsf{error\_free}(A)$.

For instance, the SC for predicate `is/2` states that the second argument is an arithmetic expression. This condition guarantees error free calls to predicate `is/2`.

Definition 5.3 (side-effect free assertion) *Let p/n be an external predicate defined in module M. The assertion* `:- trust comp p(X1,...,Xn) + sideff_free.` *in the code for M is a correct* side-effect free assertion *for predicate p/n in a logic programming system Sys if, $\forall\theta$, the execution of $\theta(p(X1, ..., Xn))$ does not produce any side effect.*

In contrast to the two previous assertions, side-effect assertions are unconditional, i.e., their SC always takes the value true. For brevity, both in the text and in the implementation we omit the SC from them.

Example 5.4 *The following assertions are predefined in* `Ciao` *for predicate* `ground/1`*:*

```
:- trust comp ground(X) : true + error_free.
:- trust comp ground(X) + sideff_free.
:- trust comp ground(X) : ground(X) + bind_ins.
```

An important thing to note is that rather than using the overall eval assertions of [PAH05b], we prefer to have separate assertions for each of the different properties required for an atom to be evaluable. There are several reasons for this. On one hand, it will allow us the use of separate analysis for inferring each of these properties (e.g., a simple reachability analysis is sufficient for unconditional side-effects while more elaborated analysis tools are needed for error and binding sensitiveness). Also, it will allow reusing such assertions for other purposes different from partial deduction. For instance, side-effect and error free assertions are also interesting for other purposes (like, e.g., for program verification, for automatic parallelization) and are frequently required by programmers separately. Finally, eval assertions include termination which is not required for ensuring correctness w.r.t. computed answers (see Sect. 4).

6 Automatic Inference of Assertions by Backwards Analysis

Recent developments in backwards analysis of logic program [HKL04, Gal04, KL02a] have pointed out novel applications in termination analysis and inference of call patterns which are guaranteed not to produce any runtime error. In this section, we outline a new application of

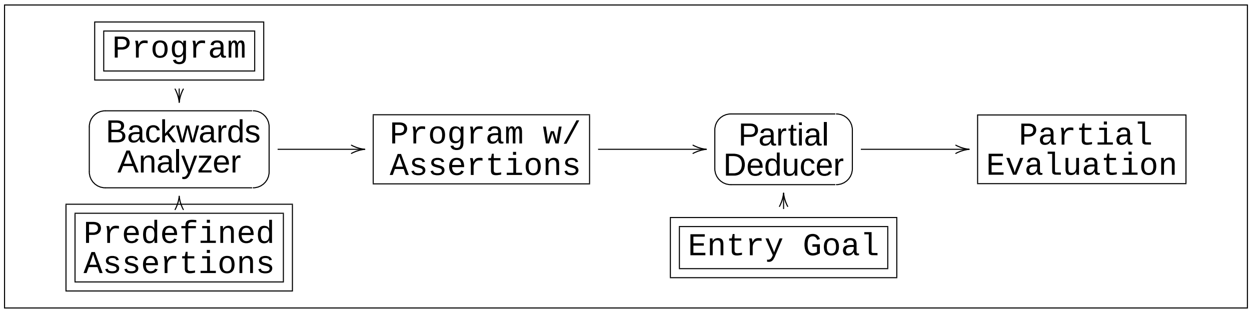

Figure 16: Backwards Analysis in Non-leftmost Partial Deduction

backwards analysis for automatically inferring binding insensitive, error free and side-effect free annotations which are useful to this purpose. Automatically figuring out when a substitution can be safely backpropagated onto a call whose execution reaches an impure predicate has been considered a difficult challenge and, to our knowledge, no accurate, satisfactory solution exists.

Fig. 16 illustrates the PD scheme based on assertions and backwards analysis that we have implemented in `CiaoPP`. Initially, given a `Program` and a set of `Predefined Assertions` for the external predicates, the Backwards Analyzer obtains a `Program w/ Assertions` which includes error_free, sideff_free and bind_ins assertions for all user predicates. Notice that this is a goal-independent process which can be started in our system regardless PD being performed or not. Afterwards, and independently from the backwards analysis process, the user can decide to partially evaluate the program. To do so, an initial call has to be provided by means of an `Entry Goal`. A Partial Deducer is executed from such program and entry with the only consideration that, whenever a non-leftmost unfolding step needs to be performed, it will take into account the information available in the generated (and predefined) assertions.

6.1 The Backwards Analyzer

Regarding the analyzer, we rely on the backwards analysis technique of [Gal04]. In this approach, the user first identifies a number of properties that are required to hold at body atoms at specific program points. A meta-program is then automatically constructed, which captures the dependencies between initial goals and the specified program points. This meta-program is based on the *resultants* semantics of logic programs, in which the meaning of a program is the set of all pairs A, R where $A = A'\theta$ and there is an LD derivation from $\leftarrow A'$ to $\leftarrow R$ with computed answer θ. An abstraction of the resultants semantics is then defined, containing all pairs $A\theta, B$ such that $A = A'\theta$ and there is an LD derivation from $\leftarrow A'$ to $\leftarrow B, B_1, \ldots, B_m$ with computed answer θ, where B corresponds to one of the specified program points. The semantics is cap-

```
:- module(main_prog,[main/2],[]).
:- use_module(comp,[long_comp/2],[]).

main(X,Y)   :- problem(X,Y), q(X).

problem(a,Y):- ground(Y),long_comp(a,Y).
problem(b,Y):- ground(Y),long_comp(b,Y).

q(a).
```

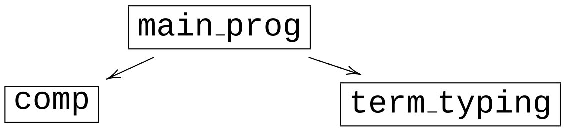

Figure 17: Example Program

tured by a meta-program defining a meta-predicate `d/2`, such that `d(A,B)` is a consequence of the meta-program whenever a pair A, B as defined above exists. Standard abstract interpretation techniques are applied to the meta-program; from the results of the analysis, conditions on initial goals can be derived which guarantee that all the given properties hold whenever the specified program points are reached.

As it appears in the figure, the analyzer starts from a program and an initial set of assertions which state the properties of interest defined in Sect. 5 for the external predicates. Essentially, the analysis algorithm propagates this information backwards in order to get the appropriate assertions for all predicates. Let us illustrate the idea by means of an example.

Example 6.1 *Consider the predefined assertions in* `Ciao` *for predicate* `ground/1` *of Ex. 5.4 and the* `Ciao` *program in Fig. 17 whose modular structure appears to the right.* `term_typing` *is the name of the module in* `Ciao` *where* `ground/1` *is defined (and thus where the assertions for* `ground/1` *are). Predicate* `long_comp/2` *is externally defined in module* `comp` *where also these predefined assertions for it are:*

```
:- trust comp long_comp(X,Y) : true + error_free.
:- trust comp long_comp(X,Y) + sideff_free.
:- trust comp long_comp(X,Y) : ground(Y) + bind_ins.
```

From the program and the available assertions (for `long_comp/2` *and* `ground/1`*), the backwards analyzer infers the following assertions for* `problem/2`*:*

```
:- trust comp problem(X,Y) : true + error_free.
:- trust comp problem(X,Y) + sideff_free.
:- trust comp problem(X,Y) : ground(Y) + bind_ins.
```

Backwards analysis of the above program, with analysis over a simple domain with elements `ground` *and* `nonground`*, yields the following dependencies, represented using the meta-predicate* `d(A,B)` *described above.*

```
d(problem(X,ground), long_comp(ground,ground)).
d(problem(X,nonground), long_comp(ground,nonground)).
```

These facts imply that whenever a call `problem(X,Y)` *is made where* `Y` *is ground, any subsequent assertions concerning binding insensitivity are satisfied; specifically, calls to* `long_comp(X,Y)` *satisfy the sufficient condition for being binding insensitive, i.e.,* `ground(Y)`*. Hence the last assertion (binding insensitivity) on* `problem(X,Y)` *is established. The analysis results for* `d/2` *also clearly establish first two assertions on* `problem(X,Y)`*, with condition* `true`*, since any call to* `problem(X,Y)` *is guaranteed to satisfy all the (trivial) error-freeness and side-effect-freeness assertions.*

The last assertion indicates that calls performed to `problem(X,Y)` *with the second argument being ground are binding insensitive. This will be very useful information for the specializer.*

6.2 The Partial Deducer

In our system, we use a standard partial deducer based on an *observable-preserving unfolding rule* defined as follows.

Definition 6.2 (observable-preserving unfolding rule) *We say that an unfolding rule is* observable-preserving *if, for any goal* $\leftarrow A_1, \ldots, A_R, \ldots, A_n$*, it always selects an atom* A_R *for unfolding such that all atoms* $A_1, \ldots, A_{R-1}$ *are pure.*

The fact that our system relies on the assertions for purity defined in Section 5 allow us to ensure that our PD scheme is *correct* in the sense that the partially evaluated program preserves the runtime behaviour (or observables) of the original one w.r.t. the predefined assertions. Of course, when it is not possible to perform an observable preserving selection, our implementation resorts to the usual solution which consists in hiding bindings and failure instead of performing backpropagation.

Let us see an example.

Example 6.3 *Consider a deterministic unfolding rule (i.e., an unfolding rule which cannot perform non-deterministic steps other than the first one). Given the program of Ex. 6.1 and the entry goal: "* `:- entry main(X,a).` *" The unfolding rule performs an initial step and* *derives the goal* `problem(X,a),q(X)`*. Now, it cannot select the atom* `problem(X,a)` *because its execution performs a non deterministic step. Fortunately, the assertions inferred for* `problem(X,Y)` *in Ex. 6.1 allow us to jump over this atom and specialize first* `q(X)`*. In particular, the first two assertions do not pose any restriction because their conditions are* `true`*, thus, there is no problem related to errors or side-effects. From the last assertion, we know that the above call is binding insensitive, since the condition "*`ground(a)`*" trivially succeeds.*

If atom `q(X)` *is evaluated first, then variable* `X` *gets instantiated to* `a`*. Now, the unfolding rule already can select the deterministic atom* `problem(a,a)` *and obtain the clause "*`main(a,a):-long_comp(a,a).`*" as partially evaluated program. Note that the residual call* `long_comp(a,a)` *is not evaluated at PD time because the given assertions for this external predicate do not guarantee that such evaluation terminates. The interesting point to note is that, without the use of assertions, the derivation is stopped when the atom* `problem(X,a)` *is selected because any call to* `problem` *is considered potentially dangerous since its execution reaches a binding sensitive predicate. The specialized program in this case is:*

```
main(A,a) :-  problem_1(A,a), q_1(A).

problem_1(a,a) :- long_comp(a,a).
problem_1(b,a) :- long_comp(b,a).

q_1(a).
```

Intuitively, this residual program is less efficient than our specialization since the execution of a call such as `main(b,a)` *would immediately fail in our specialized program whereas it would nevertheless execute the call to* `long_comp(b,a)` *in the above program which is bound to fail afterwards.*

As already mentioned in Section 4.2, our safety conditions for non-leftmost unfolding preserve computed answers, but has the well-known implication that an infinite failure can be transformed into a finite failure. However, in our framework this will only happen for predicates which do not have side-effects, since non-leftmost unfolding is only allowed in the presence of pure atoms. Nevertheless, our framework can be easily extended to preserve also infinite failure by including termination as an additional property that non-leftmost unfolding has to take into account, i.e. this implies requiring that all atoms to the left of the selected atom should be evaluable and not only pure (see Section 4.4).

The following theorem states that our PD scheme is *correct* in the sense that the partially evaluated program preserves the runtime behaviour of the original one w.r.t. the predefined assertions. We assume a correct partial evaluator implementing the traditional PD algorithm (like the one in [Gal93]) with the only modification of using a safe unfolding rule for performing non-leftmost unfolding steps as defined in Def. 6.2.

Theorem 6.4 (correctness) *Let AS be a set of correct assertions. Let P be a program and G be a goal. Let PE be a partial evaluator based on a safe unfolding rule U. Then, $PE(P, G)$ preserves the runtime behaviour of P w.r.t. AS.*

7 Conclusions

In the case of leftmost unfolding, eval assertions can be used in order to determine whether evaluation of atoms for external predicates can be fully evaluated at specialization time or not. Such eval assertions should be present whenever possible for all library (including builtin) predicates. Though the presence of such assertions is not required, as the lack of assertions is interpreted as the predicate not being evaluable under any circumstances, the more eval assertions are present for external predicates, the more profitable partial deduction will be. Ideally, eval assertions can be provided by the system developers and the user does not need to add any eval assertion.

If non-leftmost unfolding is allowed, the following conditions are required: given a goal $\leftarrow A_1, \ldots, A_R, \ldots, A_n$, backpropagation of bindings and failure for the execution of A_R is only allowed if $\mathsf{pure}(A_1) \wedge \ldots \wedge \mathsf{pure}(A_{R-1})$. An important distinction w.r.t. the case of leftmost unfolding above is that pure assertions are of interest not only for external predicates but also for internal, i.e., user-defined predicates. As already mentioned, the lack of pure assertions must be interpreted as the predicate not being pure, since impure atoms can be reached from them. Thus, for non-leftmost unfolding to be able to "jump over" internal predicates, it is required that such pure assertions are available not only for external predicates, but also for predicates internal to the module. Such assertions can be manually added by the user or, much more interestingly, as our system does, by backwards analysis. Indeed, we believe that manual introduction of assertions about purity of goals is too much of a burden for the user. Therefore, accurate non-leftmost unfolding becomes a realistic possibility only thanks to the availability of backwards analysis.

Part VI

Set-Sharing is not always redundant for Pair-Sharing

1 Summary

Sharing among program variables is vital information when analyzing logic programs. This information is often expressed either as sets or as pairs of program variables that (may) share. That is, either as set-sharing or as pair-sharing. It has been recently argued that (a) set-sharing is interesting not as an observable property in itself, but as an encoding for accurate pair-sharing, and that (b) such an encoding is in fact redundant and can be significantly simplified without loss of pair-sharing accuracy. We show that this is not the case when set-sharing is combined with other kinds of information, such as the popular freeness and in the presence of certain builtins.

2 Introduction

Program analysis is the process of inferring at compile–time inferring information about run–time properties of programs. In logic programs one of the most studied run-time properties is *sharing* among program variables. Two program variables share in a given run-time store if the terms to which they are bound have at least one run-time variable in common. A set of program variables share if they all have at least one run-time variable in common. The former kind of sharing is called *pair-sharing* while the latter is called *set-sharing*. Any of the two may be target observables of an analysis.

The importance (and hence popularity) of sharing comes from two sources. First, sharing information is in itself vital for several applications such as exploitation of independent AND-parallelism [JL92, BdlBH99], occurs check reduction [Pla84, Son86], and compile-time garbage collection [MWB90]. And second, sharing can be used to accurately keep track of other interesting run-time properties such as *freeness* (a program variable is free in a run-time store if it is either unbound or bound to a run-time variable).

Sharing analysis has therefore raised an enormous amount of interest in our research community, with many different analysis domains being proposed in the literature (see e.g., [Son86, JL89, MH91, BC93, KS94]). Two of the best known sharing analysis domains are `ASub` defined by Søndergaard [Son86] and `Sharing` defined by Jacobs and Langen [JL89, JL92]. The main difference between these two domains is the way in which they represent sharing information: while `ASub` keeps track of *pairs* of program variables that possibly share, `Sharing` keeps track of *sets* of program variables that possibly share certain variable occurrences.

These differences have subtle consequences. On the one hand, the pair sharing encoding in `ASub` allows it to keep track of linear program variables (a program variable is *linear* in a run-time store if it is bound to a term which does not have multiple occurrences of the same run-time variable). Linearity information, in turn, allows `ASub` to improve the accuracy of the abstract sharing operations. On the other hand, the set sharing encoding in `Sharing` allows it to represent several other kinds of information (such as groundness and sharing dependencies) which also result in more accurate abstract operations. In fact, when combined with linearity, `Sharing` is strictly more accurate than `ASub`. In practice, this accuracy improvement has proved to be significant [CMB$^+$95].

As a result, `Sharing` became the standard choice for sharing analysis, usually combined with other kinds of information such as freeness or structural information, even though its complexity can have significant impact on efficiency. However, the benefits of using set sharing for sharing analysis have been recently questioned (see [CFW94, BHZ97, BHZ02]). As a paradigm of the case, we cite the title of a paper by Bagnara, Hill, and Zaffanella: "Set-Sharing is redundant for Pair-Sharing" [BHZ97, BHZ02]. In this paper, the authors state the following

> **Assumption**: The goal of sharing analysis for logic programs is to detect which *pairs* of variables are definitely independent (namely they cannot be bound to terms having one or more variables in common).
>
> As far as we know this assumption is true. In the literature we can find no reference to the "independence of a *set* of variables". All the proposed applications of sharing analysis (compile-time optimizations, occur-check reduction and so on) are based on information about the independence of *pairs* of variables.

Based on the above assumption, the authors focus on defining a simpler version of `Sharing` which is however as precise as far as pair-sharing is concerned. This new simpler domain, referred to in the future as SS^ρ, is obtained by eliminating from `Sharing` information which is considered "redundant" w.r.t. the pair-sharing property. This elimination allows further simplification of the abstract operations in SS^ρ which can significantly improve its efficiency.

The popularity of the `Sharing` domain combined with the great accuracy and efficiency results obtained for SS^ρ (and the clarity with which the authors explained the intricacies of the `Sharing` domain), ensured the paper had a significant impact on the community, with many researchers now accepting that set-sharing is indeed redundant for pair-sharing (see, e.g., [KSH99, CSS99, LS00, LS02]).

The aim of this paper is to prove that this is not always the case. In particular, we will show that: (1) There exist applications which use set-sharing analysis (combined with freeness) to infer properties other than sharing between pairs of variables; and (2) When combined with information capable of distinguishing among the different variable occurrences represented by `Sharing`, this domain can yield results not obtainable with SS^ρ, *including better pair-sharing*. Such a combination is found in at least two common situations: when `Sharing` is used as a carrier for other analyses (such as freeness), and when the analysis process is improved with extra information (such as in-lined knowledge of the semantics of some predicates, for example builtins). Possible approaches to combine SS^ρ with other kinds of information without losing accuracy are also suggested.

We believe our insights will contribute to the better understanding of an abstract domain which, while being one of the most popular and more intensively studied abstract domains ever defined, remains somewhat misunderstood.

3 Preliminaries

Let us start by introducing our notation as well as the basics of the `Sharing` domain [JL89, JL92]. In doing this we will mainly follow the extremely clear summary presented in [BHZ97]. Given a set S, $\wp(S)$ denotes the powerset of S, and $\wp_f(S)$ denotes the set of all the finite subsets of S. $\mathcal{V}$ denotes a denumerable set of variables. $Var \in \wp_f(\mathcal{V})$ denotes a finite set of variables, called the *variables of interest* (e.g., the variables of a program). The set of variables in a syntactic object o is denoted $vars(o)$. $\mathcal{T}_\mathcal{V}$ is the set of first order terms over $\mathcal{V}$. A substitution θ is a mapping $\theta : \mathcal{V} \rightarrow \mathcal{T}_\mathcal{V}$, whose application to variable x is denoted by $x\theta$. Substitutions are denoted by the set of their bindings: $\theta = \{x \mapsto x\theta \mid x\theta \neq x\}$. We define the image of a substitution θ as the set $img(\theta) \stackrel{\text{def}}{=} \bigcup\{vars(x\theta) \mid x \in Var\}$.

The `Sharing` domain is formally defined as follows. Let $SH \stackrel{\text{def}}{=} \wp(SG)$, where $SG \stackrel{\text{def}}{=} \{S \subseteq Var \mid S \neq \emptyset\}$. Each element $S \in SG$ is called a *sharing set*. We will write sharing sets as strings with the variables that belong to it, e.g., sharing set $\{x, y, z\}$ will be denoted xyz. A sharing set of size 2 is called a *sharing pair*.

The function $occ(\theta, v)$ obtains a sharing set that represents the occurrence of variable v through the variables of interest as per the substitution θ.

$$occ(\theta, v) \stackrel{\text{def}}{=} \{x \in Var \mid v \in vars(x\theta)\}$$

The abstraction of a substitution θ is obtained by computing all relevant sharing sets:

$$\alpha(\theta) \stackrel{\text{def}}{=} \{occ(\theta, v) \mid v \in img(\theta)\}.$$

Abstract element $sh \in SH$ approximates substitution θ iff $\alpha(\theta) \subseteq sh$. Conversely, the concretization of $sh \in SH$ is the set of all substitutions approximated by sh. Projection over a set $V \subseteq Var$ is given by

$$proj(sh, V) \stackrel{\text{def}}{=} \{S \cap V \mid S \in sh[V]\}$$

where, for any syntactic object o and abstraction $sh \in SH$,

$$sh[o] \stackrel{\text{def}}{=} \{S \in sh \mid S \cap vars(o) \neq \emptyset\}.$$

The pairwise (or binary) union of two abstractions is defined as:

$$sh_1 \uplus sh_2 \stackrel{\text{def}}{=} \{S_1 \cup S_2 \mid S_1 \in sh_1, S_2 \in sh_2\}.$$

The closure under (or star) union of an abstract element sh is defined as the least set sh^* that satisfies:

$$sh^* = sh \cup \{S_1 \cup S_2 \mid S_1, S_2 \in sh^*\}.$$

Abstract unification for a substitution θ is given by extending to the set of bindings of θ the following abstract unification operation for a binding:

$$amgu(sh, x \mapsto t) = (sh \setminus (sh[x] \cup sh[t])) \cup (sh[x]^* \uplus sh[t]^*).$$

The set-sharing lattice is thus given by the set

$$SS \stackrel{\text{def}}{=} \{(sh, U) | sh \in SH, U \subseteq Var, \forall S \in sh : S \subseteq U\} \cup \{\bot, \top\}$$

which is a complete lattice ordered by $\leq_{SS}$ defined as follows. For elements $\{d, (sh_1, U_1), (sh_2, U_2)\} \subseteq SS$:

$$\begin{array}{l} \bot \leq_{SS} d \\ d \leq_{SS} \top \\ (sh_1, U_1) \leq_{SS} (sh_2, U_2) \text{ iff } U_1 = U_2 \text{ and } sh_1 \subseteq sh_2. \end{array}$$

The lifting of $\cup$, $proj$, and $amgu$ defined over SH to define the abstract operations $\sqcup$, $Proj$, and $Amgu$ over SS is straightforward.

Example 1. Let $Var = \{x, y, z\}$ be the set of variables of interest and consider the substitutions $\theta_1 = \{x \mapsto f(u, u, v), y \mapsto g(u, v, w, o), z \mapsto h(u)\}$ and $\theta_2 = \{x \mapsto u, y \mapsto u, z \mapsto 1\}$. Then, $sh_1 = \alpha(\theta_1) = \{xy, xyz, y\}$, where sharing set xyz represents the occurrence of variable u in x, y and z, sharing set xy represents the occurrence of variable v in x and y, and sharing set y represents the occurrence of variables w and o in y. Similarly, we have that $sh_2 = \alpha(\theta_2) = \{xy\}$ where sharing set xy represents the occurrence of variable u in x and y. Let $U = Var$. We then have that $(sh_2, U) \leq_{SS} (sh_1, U)$ and thus $(sh_1, U) \sqcup (sh_2, U) = (sh_1, U)$. Finally, let $V = \{x, y\}$, $Proj((sh_1, U), V) = (\{xy, y\}, V)$. Note that the sharing set xy in the projected abstraction represents not only the occurrence of variable u but also that of v. $\diamond$

4 Eliminating redundancy from `Sharing`

One of the main insights in [CFW94, BHZ97] regarding the `Sharing` domain is the detection of sets which are redundant (and can thus be safely eliminated or not produced) as far as pair-sharing is concerned. Given an element sh of SH, sharing set $S \in sh$ is *redundant* w.r.t. pair sharing if and only if all its sharing pairs can be extracted from other subsets of S which also appear in sh. Formally, let $pairs(S) \stackrel{\text{def}}{=} \{xy \mid x, y \in S, x \neq y\}$. Then, S is redundant iff

$$pairs(S) = \bigcup\{pairs(T) \mid T \in sh, T \subset S\}$$

Example 2. Consider the abstraction $sh = \{xy, xz, yz, xyz\}$ defined over $Var = \{x, y, z\}$. It is easy to see that set $xyz \in sh$ is redundant w.r.t. pair sharing. $\diamond$

Based on this insight, a closure operator, $\rho : SH \rightarrow SH$, is defined in [BHZ97] to add to each $sh \in SH$ the set of elements which are redundant for sh. Formally:

$$\rho(sh) \stackrel{\text{def}}{=} \{S \in SG \mid \forall x \in S : S \in sh[x]^*\}.$$

This function is then used to define a new domain SS^ρ which is the quotient of SS w.r.t. the new equivalence relation induced by ρ: elements d_1 and d_2 are equivalent iff $\rho(d_1) = \rho(d_2)$. The authors prove that (a) the addition of redundant elements does not cause any precision loss as far as pair-sharing is concerned, i.e., that SS^ρ is as good as SS at representing pair-sharing, and that (b) ρ is a congruence w.r.t. the abstract operations $Amgu$, $\sqcup$ and $Proj$. Thus, they conclude that SS^ρ is as good as SS also for propagating pair-sharing through the analysis process.

The above insight is used by [BHZ97] to perform two major changes to the `Sharing` domain. Firstly, redundant elements can be eliminated (although experimental results suggest that this is not always advantageous). And secondly, addition of redundant elements can be avoided by replacing the star union with the binary union operation without loss of accuracy. This is a very important change since it can have significant impact on efficiency by simplifying one of the most expensive abstract operations in `Sharing`.

The results obtained in [BHZ97] are indeed interesting and can be very useful in some contexts. However, there are situations in which the lack of redundant sets can lead to loss of accuracy w.r.t. pair sharing, and even incorrect results if the full expressive power of `Sharing` is assumed to be still present in SS^ρ.

Example 3. Consider the abstractions $sh_1 = \{x, y, z, xy, xz, yz\}$ and $sh_2 = \{x, y, z, xy, xz, yz, xyz\}$ defined over $Var = \{x, y, z\}$, and note that $\rho(sh_1) = sh_2$, i.e., the sharing set xyz is redundant for sh_2.

Consider the Prolog builtin $x == y$ which succeeds if program variables x and y are bound at run-time to identical terms. A sophisticated implementation of the `Sharing` domain (such as that of [BdlBH94]) could take advantage of this information and eliminate every single sharing set in which the program variables x and y appear but not together (since all variables which occur in x must also occur in y, and vice versa). Thus, correct and precise abstractions of a situation in which the builtin was successfully executed in stores represented by sh_1 and sh_2, will become $sh'_1 = \{z, xy\}$ and $sh'_2 = \{z, xy, xyz\}$, respectively. However, it is easy to see that $pairs(sh'_1) \neq pairs(sh'_2)$, since z is definitely independent of both x and y in sh'_1 while it might still share with them in sh'_2. $\diamond$

The above example shows that `Sharing` can make use of the information provided by other sources in order to improve the pair-sharing accuracy of its elements, while the same action might lead to incorrect results for elements of SS^ρ if redundant sharing sets had actually been eliminated from those elements. As we will see in the following sections, this can happen when using information coming not only from builtins, but also from other domains (such as freeness) which are usually combined with set-sharing. Furthermore, useful information other than sharing can be inferred from combinations of `Sharing` and other sources which are not possible with SS^ρ.

5 When redundant sets are no longer redundant

The problem illustrated in the previous example is rooted in the always surprising complexity of the information encoded by elements of SH. As indicated by [BHZ97, BHZ02], elements of SH can encode definite groundness (e.g., x is ground), groundness dependencies (e.g., if x becomes ground then y is ground), and sharing dependencies.[14] However, as we will see in this section, these are only by-products of the main property represented by elements of SH: the different variable occurrences shared by each set of program variables.

The groundness of variable x, and the sharing independence between variables x and y (i.e., the fact that x and y are known not to share) can be expressed by an element $sh \in SH$ as follows:

$$ground(x) \text{ iff } \forall S \in sh : \; x \notin S$$
$$indep(x, y) \text{ iff } \forall S \in sh : xy \not\subseteq S$$

where $ground(x)$ represents the fact that variable x is ground in all substitutions abstracted by sh, and $indep(x, y)$ represents the fact that variables x and y do not share in any substitution abstracted by $sh \in SH$.

[14]The fact that it also encodes independence (e.g., x does not share with y) was probably obviated because this is also encoded by pair-sharing.

Groundness dependencies in $sh \in SH$ can be easily obtained from the above statements in the following way. Let us assume that x is known to be ground. We can then modify sh by enforcing $\forall S \in sh: \; x \notin S$ to hold, i.e., by eliminating every $S \in sh$ such that $x \in S$. If we can then prove that the same statement holds for some other variable y, we would then know that the implication $ground(x) \rightarrow ground(y)$ holds for sh. This simply illustrates the well known result that `Sharing` subsumes the groundness dependency domain Def. The same method can be used for obtaining other dependencies for elements sh of SH. The following were used in [BdlBH99] for simplifying parallelization tests:

1. $ground(x_1) \wedge \ldots \wedge ground(x_n) \rightarrow ground(y)$ if
   $\forall S \in sh: \;$ if $y \in S$ then $\{x_1, \ldots, x_n\} \cap S \neq \emptyset$

2. $ground(x_1) \wedge \ldots \wedge ground(x_n) \rightarrow indep(y, z)$ if
   $\forall S \in sh: \;$ if $\{y, z\} \subseteq S$ then $\{x_1, \ldots, x_n\} \cap S \neq \emptyset$

3. $indep(x_1, y_1) \wedge \ldots \wedge indep(x_n, y_n) \rightarrow ground(z)$ if
   $\forall S \in sh: \;$ if $z \in S$ then $\exists j \in [1, n], \; \{x_j, y_j\} \subseteq S$

4. $indep(x_1, y_1) \wedge \ldots \wedge indep(x_n, y_n) \rightarrow indep(w, z)$ if
   $\forall S \in sh: \;$ if $\{w, z\} \subseteq S$ then $\exists j \in [1, n], \; \{x_j, y_j\} \subseteq S$

Let us now characterize in a similar way the (non-symmetrical) property $covers(x, y)$ expressed by an element $sh \in SH$ as follows:

$$covers(x, y) \text{ iff } \forall S \in sh: \; \text{if } y \in S \text{ then } x \in S$$

where $covers(x, y)$ indicates that variable y shares all its variables with variable x and, therefore, every sharing set in which y appears must also contain x. We can now derive other sharing dependencies for any $sh \in SH$, such as:

5. $covers(x_1, y_1) \wedge \ldots \wedge covers(x_n, y_n) \rightarrow ground(z)$ if
   $\forall S \in sh: \;$ if $z \in S$ then $\exists j \in [1, n], \; y_j \in S, \; x_j \notin S$

6. $covers(x_1, y_1) \wedge \ldots \wedge covers(x_n, y_n) \rightarrow indep(w, z)$ if
   $\forall S \in sh: \;$ if $\{w, z\} \subseteq S$ then $\exists j \in [1, n], \; y_j \in S, \; x_j \notin S$

7. $covers(x_1, y_1) \wedge \ldots \wedge covers(x_n, y_n) \rightarrow covers(w, z)$ if
   $\forall S \in sh: \;$ if $z \in S, w \notin S$ then $\exists j \in [1, n], \; y_j \in S, \; x_j \notin S$

It is important to note that while the expressions with only $ground(x)$ and $indep(x, y)$ elements can also hold for any element of SS^ρ, this is not true for the expressions with coverage information.

Example 4. Consider again the abstractions introduced by Example 3, $sh_1 = \{x, y, z, xy, xz, yz\}$ and $sh_2 = \{x, y, z, xy, xz, yz, xyz\}$ which are defined over $Var = \{x, y, z\}$. Let us assume that both abstractions belong to `Sharing`. While implication $covers(x, y) \wedge covers(y, x) \rightarrow indep(x, z)$ holds for sh_1, it does not hold for sh_2. If we now consider the SS^ρ domain, both abstractions would be represented by the element sh_1. Therefore, the implication should not hold for sh_1 in SS^ρ. $\diamond$

In order to understand why, consider the differences between the expressions $ground(x)$ iff $\forall S \in sh : x \notin S$, and $indep(x, y)$ iff $\forall S \in sh : xy \not\subseteq S$, and the expression $covers(x, y)$ iff $\forall S \in sh$: if $y \in S$ then $x \in S$. While in the first two the sharing sets which violate the right hand side of the expressions would always include the redundant set (if any), those which violate the last expression would not. Thus, to assume coverage might result in the subset of a redundant set being eliminated without the redundant set itself being eliminated. In this way sharing sets which are considered redundant at some point, might become non redundant once coverage information is added and, therefore, their elimination (or non generation) can lead to incorrect information. For example, consider the substitution $sh = \{xyz, xy, xz, yz\}$. While the problematic sets for $ground(x)$ and $indep(x, y)$ in sh are xyz, xy, xz and xyz, xy, respectively, the only one for $covers(x, y)$ is yz. But once yz is removed from sh, xyz is no longer redundant: it is the only sharing set able (when x covers y) to represent the possible sharing between x and y.

As a result, sharing sets initially redundant for pair-sharing can prove useful whenever combined with other sources of information (coming from builtins, other analysis domains, etc.) capable of distinguishing between the variable occurrences represented by the redundant sharing sets and the variable occurrences represented by their subsets, so that, once the extra information is added, a sharing set previously identified as redundant will no longer be so.

6 Combining `Sharing` with freeness

In this section we will use the popular combination of `Sharing` with freeness information to illustrate two points. First, that very common sources of information (such as freeness) can distinguish between variable occurrences, an ability which can be exploited in ways that can make a redundant set no longer redundant. Thus, it can be advantageous not to eliminate them. And second, that the goal of sharing analysis for logic programs is not only to detect which pairs of variables are definitely independent, but also to detect (or propagate) many other kinds of information.

In order to illustrate these points we will use the notion of *active* sharing sets [CH94]. A sharing set $S \in sh$ is said to be *active* for store $c \in \gamma(sh)$ iff $S \in \alpha(c)$. All sharing sets $\{S_1, \cdots, S_n\} \subseteq sh$ are said to be active *at the same time* if there exists a store $c \in \gamma(sh)$ such that $\forall 1 \leq i \leq n, S_i \in \alpha(c)$. If only the information in `Sharing` is taken into account, then all sharing sets in any $sh \in SH$ can be active at the same time.

Example 5. Consider the set-sharing abstraction $sh = \{x, xy, yz\}$ defined over $Var = \{x, y, z\}$. All sets in sh can be active at the same time since there exists a store, say $\theta = \{x = f(u, v), y = f(v, w), z = f(w)\}$, such that $\alpha(\theta) = sh$. In particular, u is the variable represented by sharing set x, v is represented by xy, and w is represented by yz. $\diamond$

However, this is not always the case when considering information outside the scope of `Sharing`. In some cases, two or more sharing sets cannot be active at the same time since, thanks to some extra information, we can determine that these sharing sets must represent the same variable(s) occurrence.

Example 6. Consider again the set-sharing abstraction $sh = \{x, xy, yz\}$ defined over $Var = \{x, y, z\}$, and let us now assume y and z are known to be free variables. As pointed out in [CH94], since each sharing set in an abstraction represents a different occurrence of one or more variables, no two sharing sets containing the same free variable can be active at the same time (the same variable cannot be a different occurrence). In our example, xy and yz cannot be active at the same time since there is no concrete store with both y and z free, such that both share a variable not shared with anyone else (sharing set yz) and y also shares a different variable with x (sharing set xy). $\diamond$

Knowing which sharing sets in abstraction sh can be active at the same time according to Ω is useful because we can use thois notion to divide sh into $\{sh_1, \cdots, sh_n\}$ such that $sh = sh_1 \cup \ldots \cup sh_n$, $\forall i, 1 \leq i \leq n$ all sets in sh_i can be active at the same time, and $\neg\exists j, 1 \leq j \leq n : j \neq i, sh_j \subseteq sh_i$.

Example 7. Consider again the abstraction $sh = \{x, xy, yz\}$ defined over $Var = \{x, y, z\}$. If y and z are known to be free variables, sh can be divided into two different sets, $\{x, xy\}$ and $\{x, yz\}$, whose sharing sets can all be active at the same time. The former represents the concrete stores in which x definitely shares a variable with y (which is actually known to be y itself), and x might also have some variable which is not shared with anyone else. The latter represents the stores in which the free variables y and z are aliased and x might have some variables which are not shared with anyone else. $\diamond$

Note that the different sh_i together with Ω describe disjoints sets of concrete stores. Furthermore, even though $(\bigcup_i \gamma(sh_i)) \cap \gamma(\Omega)$ is still equivalent to $\gamma(sh) \cap \gamma(\Omega)$ (which justifies the correctness of dividing sh into the different sh_i in the presence of Ω), it is often the case that $\bigcup_i \gamma(sh_i) \subset \gamma(sh)$, as it happens in the above example. As a result, it is generally easier to understand the concretization of sh and Ω by means of the concretization of each sh_i and Ω. Let us use this to show how the direct-product domain [CC79] of `Sharing` and freeness can be used to improve pair-sharing.

Example 8. Consider the abstraction $sh = \{xy, xz, yz, xyz\}$ defined over program variables x, y and z. If we knew that x, y, and z are free we could divide sh into the sets $sh_1 = \{xy\}, sh_2 = \{xz\}, sh_3 = \{yz\}$ and $sh_4 = \{xyz\}$. Now, sh_1 represents stores in which z is known to be ground, which is not true according to our freeness information. Thus, its sharing sets (xy) can be eliminated from sh. The same reasoning applies to sh_2 and sh_3. Thus, sh can be simplified to $\{xyz\}$ indicating that all variables definitely share (which of course also implies their definite pair-sharing dependencies). Note that if the set xyz did not belong to the abstraction, the concretization of sh in the context of freeness would be empty (indicating a failure in the program). $\diamond$

The above example shows how the direct-product domain of SS^ρ and freeness might be incorrect if the full power of set-sharing is assumed to be still present in SS^ρ. This occurs whenever a redundant set is known to contain a free variable, since it would then appear in an sh_i without one or more of its subsets. Thus, the set would no longer be redundant for sh_i. A simple solution would be to behave as if redundant sets containing free variables were present in the SS^ρ abstractions even if they do not appear explicitly in them. It would be easy to think that such solution does not lose accuracy w.r.t. pair sharing. This is, however, not true.

Example 9. Consider the set-sharing abstraction $sh = \{xy, xz, yz\}$ defined over $Var = \{x, y, z\}$. If we knew that y and z were free, we could divide sh into the sets $sh_1 = \{xy, xz\}$ and $sh_2 = \{yz\}$, respectively representing the concrete stores in which x shares with y and z, which do not share among them, and those in which x does not share with anyone and y shares with z. Note that these two situations are mutually exclusive. This allow us to prove (among others) that:

$$indep(y, z) \text{ iff } \neg indep(x, y) \text{ and } indep(y, z) \text{ iff } \neg indep(x, z).$$

This is crucial pair-sharing information (e.g., for automatic AND-parallelization, as we will see in the next section). If the redundant set xyz could have been eliminated from sh, the above expression might not hold, since the variables might then be aliased to the same free variable, thus capturing also the case in which all of them are definitely dependent of each other. $\diamond$

Let us now show how combining `Sharing` and freeness information, as done for example in `Sharing+Freeness` [MH91], yields interesting kinds of information other than the sharing itself, information which is the goal of such analyses for several applications.

Example 10. Consider again the set-sharing abstraction $sh = \{xy, xz, yz\}$ defined over $Var = \{x, y, z\}$. As mentioned above, if we knew that y and z were free, we could divide sh into the sets $sh_1 = \{xy, xz\}$ and $sh_2 = \{yz\}$. The concrete stores represented by these sets can in fact be described much more accurately than we did in the previous example: While sh_1 represents stores in which x is bound to a term with two (and only two) non-aliased free variables (y and z), sh_2 represents those stores in which x is ground, and y and z are free aliased variables. As a result, we can be sure sh only represents stores in which x is bound to a non-variable term. $\diamond$

Definite information about non-variable bindings is used, for example, to determine whether dynamic scheduled goals waiting for a program variable to become non-variable can be woken up, as performed by [dlBMS95]. However, such information cannot be obtained if redundant sets containing free variables are eliminated.

Example 11. Consider the set-sharing abstractions $sh = \{xy, xz, yz\}$ above and $sh' = sh \cup \{xyz\}$ where y and z are known to be free, we could divide sh' into the sets $sh_1 = \{xy, xz\}$ and $sh_2 = \{yz\}$ and $sh_3 = \{xyz\}$. The first two are as above, while the third represents stores in which all x, y and z share the same variables (with x possibly being a free variable). Thus, sh' does not only represent stores in which x is bound to a non-variable term. $\diamond$

Definite knowledge about non-variable bindings is not the only kind of useful information that can be inferred from combining `Sharing` and freeness. The combination can also be used to detect new bindings added by some body literal.

Example 12. Consider again the set-sharing abstraction $sh = \{xy, xz, yz\}$ where y and z are known to be free. Let us assume that sh is the abstract call for body literal $p(x, y, z)$ (i.e., the abstraction at the program point right before executing the literal) and that $sh' = \{xy, xz, yz, xyz\}$ is the abstract answer for $p(x, y, z)$ (i.e., the abstraction at the program point right after executing the literal) with y and z still known to be free. The addition of sharing set xyz means that a new binding aliasing y and z might have been introduced by $p(x, y, z)$. However, if the abstract answer is found to be identical to the call sh, we can be sure that none of the three program variables has been further instantiated (since they are still known to be free) nor any new aliasing introduced among them. $\diamond$

The above kind of information is used, for example, for detecting non-strict independence [CH94] as we will see in the next section. As shown in the above example, this information cannot be inferred if redundant sets might have been eliminated (or not produced).

7 When independence among sets is relevant

This section uses the well-known application of automatic parallelization within the independent AND-parallelism model [Con83] to illustrate how some applications (a) require independence among sets (as opposed to pairs) of variables, and (b) can benefit from combining `Sharing` with freeness information in ways which would not be possible with SS^{ρ}. The relevance of this application comes from the fact that it is not only one of the best known applications of sharing information, but also the one for which the `Sharing` domain was developed.

In the independent AND-parallelism model goals g_1 and g_2 in the sequence g_1, g_2 can be run in parallel in constraint store c if g_2 is independent of g_1 for store c. In this context, independence refers to the conditions that the run-time behavior of these goals must satisfy in order to guarantee the correctness and efficiency of their parallelization w.r.t. their sequential execution. This can be expressed as follows: goal g_2 is independent of goal g_1 for store c iff the execution of g_2 in c has the same number of computation steps, cost, and answers as that of g_2 in any store c' obtained from executing g_1 in c.

Note that the general independence condition introduced above is thus neither symmetric nor established between pairs of variables, as assumed by [BHZ97, BHZ02]. However, this general notion of independence is indeed rarely used. Instead, sufficient (and thus simpler) conditions are generally used to ensure independence. These conditions can be divided into two main groups: a priori and a posteriori. A priori conditions can always be checked prior to the execution of the goals involved, while a posteriori conditions can be based on the actual behaviour of the goals to be run in parallel.

A priori conditions are more popular even though they can be less accurate. The reasons are twofold. First, they can only be based on the characteristics of the store c and the variables belonging to the goals to be run in parallel. Thus, they are relatively simple. And second, they can be used as run-time tests without actually running the goals themselves. This is useful whenever the conditions cannot be proved correct at compile-time. Note that a priori conditions must be symmetric: goals g_1 and g_2 are independent for c iff g_1 is independent of g_2 for c *and* g_2 is independent of g_1 for c.

The most general a priori condition, called *projection independence*, was defined in [dlBHM00] as follows: goals g_1 and g_2 are independent for c if for any variable $x \in vars(g_1) \cap vars(g_2)$, x is uniquely defined by c (i.e., ground), and the constraint obtained by conjoining the projection of c over $vars(g_1)$ and the projection of c over $vars(g_2)$ entails (i.e., logically implies) the constraint obtained by projecting c over $vars(g_1) \cup vars(g_2)$.

Example 13. Consider the literals $p(x), q(y), r(z)$ and constraint $c \equiv \{x = y + z\}$. The projection of c over the sets of variables containing either one or two variables from $\{x, y, z\}$ is the empty constraint $true$. Thus, we can ensure that every pair of literals, say $p(x)$ and $q(y)$, can run in parallel. However, no literal can run in parallel with the goal formed by the conjunction of the other two literals, e.g., $p(x)$ cannot run in parallel with goal $q(y), r(z)$, since the projection of c over $\{x, y, z\}$ is c itself, which is indeed not entailed by $true$. $\diamond$

Therefore, as mentioned in both [MBdlBH99] and [dlBBH96], in general projection independence does indeed rely on the independence of a pair of *sets* of variables. However, for the Herbrand case projection independence is equivalent to the better known a priori condition called *strict independence,* which was introduced in [Con83, DeG87] and formally defined and proved correct in [HR95]. It states that goals g_1 and g_2 are strictly independent for substitution θ iff $vars(g_1)$ do not share with $vars(g_2)$ for θ, i.e., iff $vars(g_1\theta) \cap vars(g_2\theta) = \emptyset$. It is easy to prove that this is equivalent to requiring that for every pair of variables $xy, x \in vars(g_1), y \in vars(g_2)$, x and y do not share.

Therefore, only for a priori conditions and the Herbrand domain, is parallelization based on the independence of pairs of variables. And even in this case, the `Sharing` domain is more powerful than SS^ρ when combined with other kinds of information.

Example 14. Consider again the abstractions $sh = \{xy, xz, yz, xyz\}$ and $sh' = \{xy, xz, yz\}$ defined over $Var = \{x, y, z\}$. Example 9 illustrated how the formula

$$indep(y, z) \text{ iff } \neg indep(x, y) \;\text{ and }\; indep(y, z) \text{ iff } \neg indep(x, z)$$

holds for sh' but not for sh when y and z are known to be free.

Consider the automatic parallelization of sequential goal `p(y),q(z),r(x)` for the usual case of the a priori condition strict independence and the Herbrand domain. In the absence of any information regarding the state of the store occurring right before the sequential goal is executed, the compiler could rewrite the sequential goal into the following parallel goal (leftmost column):

```
(  indep(y,z) ->             | (  indep(y,z) ->      | (  indep(y,z) ->
   (  indep(x,y) ->          |                       |
      (  indep(x,z) ->       |                       |
         p(y)&q(z)&r(x)      |                       |
      ;  p(y)&(q(z),r(x))    |                       |
      )                      |                       |
   ;  (p(y)&q(z)),r(x)       |    (p(y)&q(z)),r(x)   |    (p(y)&q(z)),r(x)
   )                         |                       |
;  indep(x,z) ->             | ;                     | ;  indep(x,z) ->
   p(y),(q(z)&r(x))          |    p(y),(q(z)&r(x))   |    p(y),(q(z)&r(x))
;  p(y),q(z),r(x)            |                       | ;  p(y),q(z),r(x)
)                            | )                     | )
```

where the operator & represents parallel execution of two goals, and the run-time test `indep(x,y)` succeeds if the two variables do not share at run-time. The middle and right columns represent the simplifications that can be performed to the parallel goal in the context of sh' and sh, respectively. This is because while test `indep(x,y)` is known to fail if `indep(y,z)` succeeds for both sh and sh', test `indep(x,z)` is known to succeed if `indep(y,z)` fails for sh' but not for sh. Thus, `indep(x,z)` still needs to be tested at run-time with the resulting loss of efficiency. $\diamond$

The assumption is also incorrect when considering a posteriori conditions, even those associated to the Herbrand domain. In particular, strict independence has been generalised to several different [HR95] a posteriori notions of *non-strict independence*. These notions allow goals that share variables to run in parallel as long as the bindings established for those shared variables satisfy certain conditions. For example, one of the simpler notions only allows g_1 to instantiate a shared variable and does not allow any aliasing (of different shared variables) to be created during the execution of g_1 that might affect goals to the right. Thus, for this notion, the conditions are established between the *bindings* introduced by the two goals over their respective set of variables, and cannot be expressed using only sharing between pairs of variables.

There has been at least one attempt [CH94] at inferring non-strict independence at compile-time using the abstract domain `Sharing+Freeness`. The inference is based on two conditions. The first ensures that (C1) no shared variables are further instantiated by g_1. This is done by requiring that (a) all shared variables share through variables known to be free in the abstract call of g_1 (all sharing sets in the abstract call containing shared variables also contain a free variable), and (b) all these variables must remain free in the abstract answer of g_1 (all such sharing sets still contain a free variable after the analysis of g_1). This first condition can be detected in the SS^ρ domain since the existence of a free variable in every sharing pair ensures the existence of a free variable in the "redundant" sharing set. Thus, the absence of such sharing set is not a problem.

This is not however the case for the second condition, which ensures that no aliasing is introduced among shared variables by requiring C1 and, additionally, that (C2) there is no introduction in the abstract answer of any sharing set resulting from the union of several sets such that none contain the same free variable, and at least two contain variables belonging to both goals.

Example 15. Consider again the set-sharing abstraction $sh = \{xy, xz, yz\}$ where y and z are known to be free. Let us assume that sh is the abstract call for body $p(x, y, z), q(x, y, z)$ and that $sh' = \{xy, xz, yz, xyz\}$ is the abstract answer for $p(x, y, z)$ with y and z still known to be free. All sharing sets in sh containing variables from both literals contain a free variable which remains free in sh'. Thus, C1 is satisfied. However, there exists a set xyz in sh' which can be obtained by unioning at least two sets xy and xz in sh which contain variables from both literals and have no variable in common known to be free in sh. The appearance of such a set represents the possible aliasing of y and z by $p(x, y, z)$. This appearance violates C2 and thus the goals cannot run in parallel. Note that if the abstract answer was found to be identical to sh (i.e., if the redundant set xyz was absent), we would have been able to ensure that none of the three program variables had been further instantiated nor any new aliasing introduced among them. Therefore, we could have ensured that g_2 is independent of g_1 for the stores represented by sh and the associated freeness information, thus allowing their parallel execution. $\diamond$

The above example illustrates the fact that an equivalent inference cannot be performed in the SS^ρ domain augmented with freeness *unless care is taken when considering redundant sharing sets which include program variables known to be free*. This is because the inference strongly depends on distinguishing between the different bindings introduced during execution of the goals to be run in parallel, and as a result, on distinguishing between the different shared variables represented by the abstractions in the domain. Thus, elimination of redundant sets can render the method incorrect. One possible solution is to always assume that redundant sets containing free variables are present when combining SS^ρ with freeness information. However, as shown in Example 9, this might be imprecise. Another, more accurate solution, is to only eliminate redundant sets which do not contain variables known to be free.

8 Conclusion

We have shown that the power of set-sharing does not come from representing sets of variables that share, but from representing different variable occurrences. As a result, eliminating from `Sharing` information which is considered "redundant" w.r.t. the pair-sharing property as performed in SS^ρ can have unexpected consequences. In particular, when `Sharing` is combined with some other kinds of information capable of distinguishing among variable occurrences in a way that can make a redundant set no longer redundant, it can yield results not obtainable with SS^ρ, *including better pair-sharing*. Furthermore, there exist applications which use `Sharing` analysis (combined with freeness) to infer properties other than sharing between pairs of variables and which cannot be inferred if SS^ρ is used instead. We have proposed some possible solutions to this problem.

References

[AHV02] E. Albert, M. Hanus, and G. Vidal. A practical partial evaluation scheme for multi-paradigm declarative languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BC93] M. Bruynooghe and M. Codish. Freeness, sharing, linearity and correctness – all at once. In *Proc. Third International Workshop on Static Analysis*, pages 153–164. Springer LNCS 724, 1993.

[BCC$^+$04] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.10). Technical Report CLIP3/97.1.10(04), School of Computer Science (UPM), August 2004. Available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

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

[BdlBH94] F. Bueno, M. García de la Banda, and M. Hermenegildo. The PLAI Abstract Interpretation System. Technical Report CLIP2/94.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, February 1994.

[BdlBH99] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[BdlBH$^+$01] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[BGLM94a] A. Bossi, M. Gabbrieli, G. Levi, and M.C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122(1,2):3–47, 1994.

[BGLM94b] Annalisa Bossi, Maurizio Gabbrielli, Giorgio Levi, and Maurizio Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19/20:149–197, 1994.

[BHZ97] R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. In *Static Analysis Symposium*, pages 53–67. Springer-Verlag, 1997.

[BHZ02] R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 277(1-2):3–46, 2002.

[BJ03] F. Besson and T. Jensen. Modular class analysis with datalog. In *10th International Symposium on Static Analysis, SAS 2003*, number 2694 in LNCS. Springer, 2003.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986.

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

[BSM92] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding during Partial Deduction. *New Generation Computing*, 1(11):47–79, 1992.

[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.

[CC79] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.

[CD93] M. Codish and B. Demoen. Analysing logic programs using “Prop”-ositional logic programs and a magic wand. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*. MIT Press, 1993.

[CDG93] M. Codish, S.K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proc. POPL’93*, 1993.

[CFW94] A. Cortesi, G. Filé, and W. Winsborough. The quotient of an abstract interpretation for comparing static analyses. In *GULP-PRODE'94 Joint Conference on Declarative Programming*, pages 372–397, 1994.

[CH94] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.

[CH00] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[Cla79] K. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, London, Department of Computing, 1979.

[CLM01] Marco Comini, Giorgio Levi, and Maria Chiara Meo. A theory of observables for logic programs. *Information and Computation*, 169(1):23–80, 2001.

[CMB$^+$95] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, January 1995.

[Con83] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[CRV02] B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence Based Abstract Interpretation of Prolog. *Theory and Practice of Logic Programming*, 2(1):25–84, 2002.

[CS02] Michael Codish and Harald Søndergaard. Meta-circular abstract interpretation in prolog. In Torben Mogensen, David Schmidt, and I. Hal Sudburough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 109–134. Springer-Verlag, 2002.

[CSS99] Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.

[CT99] Michael Codish and Cohavit Taboch. A semantic basic for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[CV94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[DeG87] D. DeGroot. A Technique for Compiling Execution Graph Expressions for Restricted AND-parallelism in Logic Programs. In *Int'l Supercomputing Conference*, pages 80–89, Athens, 1987. Springer Verlag.

[dlBBH96] M. García de la Banda, F. Bueno, and M. Hermenegildo. Towards Independent And-Parallelism in CLP. In *Programming Languages: Implementation, Logics, and Programs*, number 1140 in LNCS, pages 77–91, Aachen, Germany, September 1996. Springer-Verlag.

[dlBHM00] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems*, 22(2):269–339, March 2000.

[dlBMS95] M. García de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. In *1995 International Logic Programming Symposium*, pages 417–431, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.

[DR94] S. Debray and R. Ramakrishnan. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming*, 18:149–176, 1994.

[DSGJ$^+$99] Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.

[EGM97] S. Etalle, M. Gabbrielli, and E. Marchiori. A Transformation System for CLP with Dynamic Scheduling and CCP. In *Proc. of the ACM Sigplan PEPM'97*, pages 137–150. ACM Press, New York, 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.

[Gal04] J. Gallagher. A Program Transformation for Backwards Analysis of Logic Programs. In *Logic Based Program Synthesis and Transformation: 13th International Symposium, LOPSTR 2003*, number 3018 in LNCS, pages 92–105. Springer-Verlag, 2004.

[GBS95] J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In J. W. Lloyd, editor, *Proc. of International Logic Programming Symposium*, pages 351–365, 1995.

[GC01] S. Genaim and M. Codish. Inferring termination conditions of logic programs by backwards analysis. In *International Conference on Logic for Programming, Artificial intelligence and reasoning*, volume 2250 of *Springer Lecture Notes in Artificial Intelligence*, pages 681–690, 2001.

[GDMS02] María J. García de la Banda, Bart Demoen, Kim Marriott, and Peter J. Stuckey. To the Gates of HAL: A HAL Tutorial. In *International Symposium on Functional and Logic Programming*, pages 47–66, 2002.

[GG94] Maurizio Gabbrielli and Roberto Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the 1994 ACM Symposium on Applied Computing, SAC 1994*, pages 394 – 399, 1994.

[GLM96] Maurizio Gabbrielli, Giorgio Levi, and Maria Chiara Meo. Resultants semantics for Prolog. *Journal of Logic and Computation*, 6(4):491–521, 1996.

[GS98] R. Giacobazzi and F Scozzari. A logical model for relational abstract domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.

[HKL04] Jacob M. Howe, Andy King, and Lunjin Lu. Analysing Logic Programs by Reasoning Backwards. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, LNCS, pages 380–393. Springer-Verlag, May 2004.

[HPBLG03a] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[HPBLG03b] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *Proc. of SAS'03,* pages 127–152. Springer LNCS 2694, 2003.

[HPBLG05] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming,* (2694), 2005.

[HPMS00] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[HR95] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[JL89] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[JL92] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.

[KL02a] A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, page 32, July 2002. (Theory and Practice of Logic Programming was formally known as The Journal of Logic Programming).

[KL02b] Andy King and Lunjin Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming,* 2(4-5):514–547, 2002.

[KL03] Andy King and Lunjin Lu. Forward versus backward verification of logic programs. In *ICLP'2003 (to appear)*, 2003.

[KMM$^+$98] A. Kelly, A. Macdonald, K. Marriott, H. Søndergaard, and P.J. Stuckey. Optimizing compilation for CLP($\mathcal{R}$). *ACM Transactions on Programming Languages and Systems*, 20(6):1223–1250, 1998.

[Kru60] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[KS94] A. King and P. Soper. Depth-k Sharing and Freeness. In *International Conference on Logic Programming*. MIT Press, June 1994.

[KSH99] Andy King, Jan-Georg Smaus, and Patricia M. Hill. Quotienting share for dependency analysis. In *European Symposium on Programming*, pages 59–73, 1999.

[LB02] Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[Leu94] Michael Leuschel. Partial evaluation of the "real thing". In Laurent Fribourg and Franco Turini, editors, Logic Program Synthesis and Transformation — Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, Lecture Notes in Computer Science 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.

[Leu98] Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[Leu02] Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.

[Llo87a] J. W. Lloyd. *Logic Programming*. Springer-Verlag, 1987.

[Llo87b] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[LMDS98] Michael Leuschel, Bern Martens, and Danny De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[LS91] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[LS00] Giorgio Levi and Fausto Spoto. Non pair-sharing and freeness analysis through linear refinement. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 52–61, 2000.

[LS02] Vitaly Lagoon and Peter Stuckey. Precise pair-sharing analysis of logic programs. In *Principles and Practice of Declarative Programming*, pages 99–108. ACM Press, 2002.

[MBdlBH99] K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.

[MD96] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.

[Mes96] F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In M. J. Maher, editor, *Joint International Conference and Symposium on Logic Programming*, pages 7–21. MIT Press, 1996.

[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.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

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

[MN01] F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In *Static Analysis Symposium*, volume 2126 of *LNCS*, pages 93–110, 2001.

[MWB90] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, Jerusalem, Israel, June 1990. MIT Press.

[Net02] Nicholas Nethercote. The Analysis System of HAL. Master's thesis, Monash University, 2002.

[PAH05a] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. Technical Report CLIP2/2005.0, Technical University of Madrid, February 2005.

[PAH05b] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *14th International Symposium on Logic-based Program Synthesis and Transformation*, LNCS. Springer-Verlag, 2005. To appear.

[PBH00a] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[PBH00b] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.

[PCH$^+$04] G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, Heidelberg, Germany, August 2004.

[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.

[PH00] G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[PH03] G. Puebla and M. Hermenegildo. Abstract Specialization and its Applications. In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003. Invited talk.

[Pla84] D. A. Plaisted. The occur-check problem in prolog. In *International Symposium on Logic Programming*, pages 272–281, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[Pro02] Christian W. Probst. Modular Control Flow Analysis for Libraries. In *Static Analysis Symposium, SAS'02*, volume 2477 of *LNCS*, pages 165–179. Springer-Verlag, 2002.

[RRL99] A. Rountev, B.G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *ESEC/FSE'99*, volume 1687 of *LNCS*, pages 235–252. Springer-Verlag, 1999.

[RS97] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages: Word Language Grammar*, volume 1. Springer-Verlag, 1997.

[Sah93] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[SG95] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479. The MIT Press, 1995.

[Son86] H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[TJ94] Y. M. Tang and P. Jouvelot. Separate abstract interpretation for control-flow analysis. In *Theoretical Aspects of Computer Software (TACS '94)*, number 789 in LNCS. Springer, 1994.

[VB00] W. Vanhoof and M. Bruynooghe. Towards modular binding-time analysis for first-order mercury. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[VD88] R. Venken and B. Demoen. A partial evaluation system for prolog: some practical considerations. *New Generation Computing*, 6:279–290, 1988.