PhD Thesis

Advanced Compilation Techniques based on Abstract Interpretation and Program Transformation

presented at the
Facultad de Informática
Universidad Politécnica de Madrid
in partial fulfillment of the degree of
Doctor en Informática

Author: A. Germán Puebla Sánchez
Master in Computer Science
Universidad Politécnica de Madrid – UPM

Advisor: Manuel de Hermenegildo y Salinas

Madrid, November, 1997

To my parents

Acknowledgements

Many people have helped me in different ways in order to accomplish this work. First, my family, without whose help and support I would not have been able to make research into my most important endeavor during these years.

I am also very grateful to my supervisor, Manuel Hermenegildo, whose optimism has many times helped me to achieve goals I would have found too hard otherwise. His enthusiasm and almost never-ending capacity to work has given me strength and has encouraged me to keep on working even in the hardest situations.

I have been very lucky to perform most of my work in the environment of the CLIP group at UPM whose incomparable union and predisposition to help others I have found almost nowhere else. Many thanks are due to Francisco Bueno, María García de la Banda, Manuel Carro, Pedro López, and Daniel Cabeza.

I also have to thank many other researchers who have participated in different parts of the work presented in this PhD thesis: Peter Stuckey, Kim Marriott, John Gallagher, Jan Małuszyński, Wlodek Drabent, and Pierre Deransart.

Also, during these years I have had the opportunity to meet many researchers from all over the world such as Will Winsborough, Michael Leuschel, Laura Lafave, Estrella Pulido, and many others I do not have the space to name here. I have also had the chance to visit other universities, such as the University of Bristol, the University of Linköping, and the Ecole Normale Supérieure.

Finally, I am also deeply grateful to the institutions and projects which have funded my postgraduate research activities, such as Universidad Politécnica de Madrid, Comunidad Autónoma de Madrid, Ministerio de Educación y Cultura, the ESPRIT projects ParForCE, ACCLAIM, and DISCIPL, the CICYT projects IPL-D and ELLA, and the HCM network Logic Program Synthesis and Transformation. Without such financial support this thesis would not have been possible.

Abstract

Static program analysis is becoming more relevant each day in current compilers, as it allows obtaining information at compile-time about the program behaviour. Such information can then be used to check whether the program satisfies the given specifications and also to optimize the program. Some techniques for efficient incremental analysis are presented, which allow reusing analysis information when a program which has already been analyzed is modified. On the other hand it is also studied how to correctly analyze all the features of real-life programming languages.

The program optimization techniques presented can be formalized in the general framework of program specialization. The resulting program after specialization is valid for the particular case being considered while being more efficient than the original program. The specialization proposed is abstract in the sense that the program is specialized not w.r.t. concrete values, but rather w.r.t. abstract values, which can also be seen as (possibly infinite) sets of concrete values. We further study the relationship between abstract specialization and traditional partial evaluation and how to integrate them in a novel framework with the advantages of both of them.

An assertion language is presented which allows the user to express requirements (specifications) about the program and which is also valid to express analysis results. This language enables the communication between the user and the different tools which may exist in a program development environment, and among the tools themselves. We present a scheme for run-time checking of assertions. Also, we study the possibility of compile-time checking (i.e., to prove statically that an assertion does or does not hold) based on analysis information.

Contents

Chapter 1

Introduction

Powerful computers are available nowadays in the marketplace with high computing and storage capabilities. Thus, it seems possible to make use of such machines in order to solve highly complex problems in the field of *Artificial Intelligence* [MMRS55, RN95], where high symbolic processing and search performance is often required. Unfortunately, the process of developing programs capable of tackling such problems is still a slow and costly task. Therefore, the study of techniques which allow obtaining correct and efficient programs in a reasonable time is of high interest.

The *Logic Programming* paradigm [Kow74, Kow80, Col87] has intensively been applied in the context of Artificial Intelligence due to its appropriateness for knowledge representation and for the implementation of typical applications in this field, such as expert systems, knowledge bases, etc. Such applications are in general complex and with a strong symbolic component. In addition, some recent extensions to Logic Programming, such as the *Constraint Logic Programming* paradigm [JM94] greatly facilitate the implementation and efficient execution of some kinds of tasks which are also highly relevant in Artificial Intelligence, such as planning, optimization, etc. For these reasons, in this work (Constraint) Logic Programming is taken as the programming paradigm under consideration. However, most of the techniques developed will be general and thus equally applicable to other programming paradigms. In addition, many of the techniques proposed will be of use also in improving the development process of applications which are not specific to Artificial Intelligence.

High level languages are characterized by allowing the programmer to write programs not in terms of the particular machine being used but rather in terms of the tasks the programs must perform. Thus, the programmer does not have to worry about the specifics of the machine. This results in a less time-consuming and error-prone developing process. Programs written in such high level languages are automatically translated into the language of a particular machine by another program referred to as *compiler*. An important kind of high level languages are the so-called *declarative languages*. They are called declarative in contrast to the traditional high level languages such as FORTRAN, C, Pascal, etc., which are generally referred to as *imperative languages*. The main difference between declarative languages, a good example of them being logic programming in its pure form, and imperative languages, is that in the former the program only needs to express *what* the program should compute. However, in imperative languages it is also required to express *how* to compute it by explicitly specifying in the program the control flow.

One of the main difficulties for the practical application of higher level languages is the relative performance loss introduced as a result of having written the program in a higher level language. There is a trade-off between a reduced development cost and an efficiency loss when using high level languages. Much of the efficiency loss mentioned above is due to the compilation phase, i.e., the translation from a high level language into the machine language. Such translation is in most compilers merely syntactic: a systematic translation scheme is applied to each high level program construct which produces a set of instructions in the low level language. As the translation scheme must be valid for any expression in the considered class, the low level code generated will not be in general as efficient as the code which could have been generated for each particular case. Therefore, obtaining efficient low level code from a program written in a high level language is only possible if additional information to that provided by the (isolated) syntax of a program fragment alone is available.

Another major problem that programmers must face when developing a program is the lack of automatic tools for program verification. As a result of this, much of the development time is spent in tests and manual debugging (in many cases by tracing program execution) until reasonably satisfied with the program behaviour. Thus, the development of automatic tools which allow proving that a program is correct or otherwise help to easily identify the part(s) of the program responsible for an error is another prime objective. Such tools will on one hand allow an important decrease in the program development time and on the other hand allow guaranteeing that the program satisfies the given specifications and does not contain bugs still not discovered. Verification is especially relevant in the Artificial Intelligence area as the tasks considered are usually complex, dealing with incomplete knowledge is often required, and test cases are often scarce. Particular examples are validating knowledge bases or otherwise helping in finding where inconsistencies lie.

1.1 Thesis Objectives

The final objective of the work presented in this thesis is the development, implementation, and experimental evaluation of a set of novel compilation techniques for (constraint) logic programs which contribute to improve the state of the art of this research area by facilitating the development process of programs, especially for those aimed at solving problems related to artificial intelligence. Advances in compilation technology do not only allow obtaining more efficient programs, but also they contribute in a direct way to allowing the practical use of languages of even higher levels (with the associated decrease in development time) as they contribute to make the efficiency loss associated to high level languages smaller.

The compilation techniques mentioned above focus, on one hand, in obtaining more efficient programs, and on the other hand in the study of automatic techniques for program debugging and verification.

The main technique proposed in order to accomplish the objectives above is the use of static program analysis techniques with the objective of inferring information at compile-time on the run-time behaviour of the program. Such information will then be used to both optimize and verify or diagnose the program. Even though the area of static program analysis has received considerable attention, the existing techniques are not always satisfactory in order to achieve the proposed research goals. Thus, it is also an objective of this thesis to study improvements to existing analysis techniques when currently existing ones are found lacking.

1.2 Structure of the Work

This thesis consists of three parts. Each one of them concentrates respectively on the improvement of the existing techniques for: program analysis, program optimization, and program debugging and validation. Next, each of these parts is described in detail.

1.2.1 Part I: Advanced Techniques for Program Analysis

As mentioned before, the improvement of program analysis techniques is instrumental for the other tasks which are objectives of this thesis, i.e., obtaining more efficient programs in an automatic way, and guaranteeing their correctness or otherwise to detect existing errors.

The most interesting alternative for obtaining *semantic* information, i.e., about the meaning of the program, is automatic program analysis. *Abstract Interpretation*, proposed by P. and R. Cousot [CC77], is at this point arguably the most successful formal technique for the automatic analysis of programs at compile-time [Deb89a, Bru91, MH92, Deb92, MSJ94, CV94].

Abstract interpretation has been successfully used in different systems in order to obtain information at compile-time about the run-time behaviour of programs. The fundamental idea is to simulate program execution, but rather than with the actual values which variables will contain at run-time, with an abstract (i.e., symbolic) version of such values which is simpler. These abstract values allow dealing with incomplete information and with infinite sets of values. The strong mathematical basis of this technique guarantees the correctness of the results obtained. Its successful implementation on a good number of compilers shows its practicality [HWD92, VD92, MH92, SCWY91, BGH94b].

There are, however, two practical reasons why abstract interpretation is not yet included as a usual technique in commercial compilers. On one hand, even though its efficiency is reasonable, this technique is still perceived by many as too costly. On the other hand, most of the analyzers developed to date have as target "pure" languages, in the sense that they usually do not consider the problematic features of the language which often appear in practice, such as program code which is missing or meta-programming. Thus, the analysis techniques presented in this work are aimed at:

- improving analysis efficiency, by means of:
  - the development of incremental analysis techniques
  - the usage of more efficient fixpoint algorithms, and at
- extending analysis techniques to all the features present in real-life programming languages.

In the following paragraph we explain our approach for tackling these issues.

Incremental Analysis Algorithms: Traditional global analyzers [MH92, CV94] analyze whole programs at once in a non-incremental way. There are, however, many circumstances in which it is required to reanalyze (parts of) a program which has already been analyzed. In such cases, the simple non-incremental model is inefficient as it requires analysis to be started again from scratch. A promising alternative is to perform the analysis in an incremental way, i.e., reusing as much information as possible from previous analyses. In Chapter 2 we study the different kinds of modifications which may be performed to a program and we propose four types of algorithms for incremental analysis which are capable of dealing with all usual cases of incremental change: addition, deletion, local change and arbitrary change. We also quantify experimentally the performance improvements achievable by using incremental analysis.

Optimized Fixpoint Algorithms: The algorithms used in abstract interpretation and in program analysis in general require global information about the program. As a result, the effect of the computation performed in a part of the program must be propagated to other parts in the program. This is usually done by means of an algorithm which computes a fixpoint of the analysis functions, i.e., iterations are performed until propagation does not modify the analysis information obtained in the previous iteration [MH90a, MH92]. It is thus very important to have an analysis algorithm which is capable of reaching a fixpoint in the least number of iterations possible. The usage of algorithms with detailed dependency information allows critically reducing the time required by analysis. In Chapter 3 we propose some optimized fixpoint algorithms [PH96c, PH96d] for analysis which are applicable and efficient both for the incremental and for the non-incremental (traditional) case.

Analysis of Full Languages: Even though in the implementation of an analyzer for a given programming language all the problems which may appear when analyzing any possible construct in such programming language should be addressed in one way or the other, few proposals for analysis of full languages exist which are effective. Most of them considerably restrict the class of programs which may analyzed [Deb89b]. Thus, it seems mandatory to develop analysis techniques which are capable of improving such situation, if we want analysis to be able to deal with generic programs which make use of any of the features of the language, may they be pure or not. In Chapter 4 we propose a set of techniques (some novel, some well known) which when simultaneously applied allow analyzing any program written in ISO standard Prolog.

1.2.2 Part II: Program Specialization based on Abstract Interpretation

Compilers often use static knowledge regarding invariants in the execution state of the program in order to optimize the program for such particular cases [AU77]. A good number of optimizations can be seen as special cases of *program specialization*, a formal and automatic tool for program optimization. The main objective of program specialization is to automatically overcome losses in performance which are due to general purpose algorithms by *specializing* the program for known values of the inputs. This generalizes traditional compilation techniques by identifying that specialization is the essence and common idea behind such techniques. One of the most relevant techniques for program specialization is *Partial Evaluation* [JGS93, CD93, GB90, GCS88, JLW90, LS91, Kom92, DGT96] which consists in a source to source program transformation: a program `pgm` together with a set `s` of partial input data which is known at compile-time is transformed into another program $\texttt{pgm}_{\texttt{s}}$, obtained by computing the parts of `pgm` which only depend on `s`. The gains in run-time obtained are due to computing at compile time the parts of the program which do not depend on unknown input values, and thus reducing the amount of computation which must be performed at run-time.

Regarding program optimization, three kinds of source to source program transformations will be studied. All of them rely on information obtained by means of abstract interpretation:

- abstract multiple specialization,
- integration of partial evaluation into abstract specialization,
- optimization of dynamic scheduling.

Abstract Multiple Specialization: *Multiple Specialization* [JLW90, GH91, Win92, KMM+95, KMM+96] is a technique capable of automatically generating several versions of a program procedure for different uses of such procedure. This allows further optimization as each version can be specialized independently of the rest. Even though multiple specialization has received considerable theoretical attention, it has been never integrated into a compiler nor its effects have been empirically quantified. In Chapter 5 a novel technique for multiple specialization is studied which will use the analysis information obtained by an abstract interpreter, and which requires little modification of existing abstract interpreters. We also present the implementation of such specialization technique in the context of a parallelizing compiler [BGH94b, Con83, CC94, HR95] and we study the practical relevance of such technique.

Integration of Partial Evaluation in Abstract Multiple Specialization: The relationship between abstract interpretation and partial evaluation has been foreseen in several works. However, no work to date has clearly identified the similarities and differences between these two frameworks. In Chapter 6 we compare such frameworks and we study the specialization capabilities which abstract specialization introduces and which are not possible by means of traditional partial evaluation. We then propose several alternatives for achieving all the power of partial evaluation in the framework of abstract multiple specialization.

Optimization of Dynamic Scheduling: Most second-generation logic programming languages allow an execution strategy usually referred to as *Dynamic Scheduling* which is more flexible than the traditional fixed left-to-right strategy of Prolog. Dynamic scheduling allows delaying the execution of goals which are not instantiated enough for (efficient) execution. Dynamic scheduling increases the expressive power of logic languages but it also introduces additional overhead.

The development of static analysis techniques capable of dealing with this kind of programs [MGH94, GMS95] allows the use of automatic optimization techniques in order to reduce the overhead of dynamic scheduling. In Chapter 7 we study some program transformation techniques designed with the aim of reducing such additional overhead while ensuring that the operational semantics of the program is preserved. This is crucial so as not to obtain optimized programs which are less efficient than the original program.

1.2.3 Part III: Program Debugging

It is often the case that much of the program development time is spent in program debugging and testing until we are reasonably satisfied with the program behaviour. It seems very desirable to improve this situation by developing novel techniques which help in program validation and debugging. This last part of the thesis concentrates on this objective. An important advantage of high level languages such as Prolog or constraint logic languages is that the language used to write programs is, in many cases, valid for expressing specifications about the expected behaviour of the program. This is an important factor which may contribute to encouraging the programmer to write and actually use (partial) specifications of the program. Such specifications may then be used in order to automate the validation and debugging process as much as possible.

Assertion Language: Assertions are syntactic constructions which allow expressing properties of programs. Assertions may be used to allow the communication between the user and the system and among different modules of the system. They have been used in order to replace the oracle [DNTM88] in declarative debugging [Sha82] and for the communication with the analyzer in [BCHP96].

In Chapter 8 we define an assertion language [PBH97] which is general enough so as to serve as a communication vehicle among the different modules and tools which may co-exist in an integrated development and debugging environment. It is also important that the assertion language be as simple as possible. The proposed assertion language allows expressing properties both of what the program does (for example, in order to express analysis results) and of what the program should do (requirements).

Semantic Approximations in Program Debugging: During program validation and debugging we are interested in finding out whether the program we have satisfies the requirements given for it. Often, requirements are partial specifications (approximations of the actual specifications). Moreover, the behaviour of the program may be complex and abstract interpretation techniques may be used in order to approximate the program behaviour. In Chapter 9 we study the role of semantic approximations for program validation and diagnosis. For example, they may be used at compile-time in order to prove that the program satisfies the (approximate) requirements or to show that it does not satisfy them [Bou93, CLMV96b]. In the latter case, the program is "incorrect" and diagnosis should be performed for it.

1.3 Main Contributions

The main contributions of this thesis are enumerated below. Some of these results have already been published and presented in international forums, in which case the relevant publication(s) is(are) mentioned explicitly. Also, some of these contributions have been made in collaboration with other researchers in addition to the thesis supervisor. This is also explicitly mentioned below.

- The fixpoint algorithm used in generic analysis systems has been augmented in order to allow incremental analysis. This allows reusing the (parts of the) information from previous analyses which is still valid when analyzing modified versions of a program which has already been analyzed. We have identified a set of possible changes which may be performed to a program which covers the usual modifications performed in practice. For each type of change, one or several algorithms are given which allow performing analysis in an incremental way. The proposed algorithms have been implemented in the PLAI system and the experimental results obtained show important improvements in the efficiency of the analysis. This work has been performed in collaboration with Kim Marriott, from Monash University, and with Peter Stuckey, from the University of Melbourne. These studies have been published in the International Conference on Logic Programming (ICLP) in 1995 [HPMS95].

- The requirements imposed by incremental analysis on the fixpoint algorithm to be used have been identified. Also, we have identified an important class of analysis strategies which satisfy the requirements mentioned above while at the same time providing very efficient analysis performance even in the non-incremental case. "A priori" conditions have been provided which ensure that the analysis strategies considered belong in the class mentioned above. We have proposed, implemented, and evaluated experimentally a novel algorithm for incremental analysis based on such ideas. The experimental results obtained show that such algorithm is very efficient in the incremental case and is also comparable to, and in many cases considerably more efficient than, other advanced algorithms (developed for the non-incremental case) even for non-incremental analysis. This work has been published at the Sixth International Symposium on Static Analysis (SAS) in 1996 [PH96c] and a short version has been presented at the Workshop on Static Analysis [PH96d] associated to JICSLP'96.

- Program analysis techniques based on abstract interpretation have received considerable attention. Both general analysis frameworks and abstract domains are comparatively well understood. However, most of the analysis techniques proposed to date restrict in one way or the other the class of programs which can be analyzed. We have proposed the first set of analysis techniques (some known, some novel) which allow analyzing in a correct way any program written using any of the features of ISO standard Prolog. This work has been performed in collaboration with Francisco Bueno and Daniel Cabeza, both from the CLIP group the Technical University of Madrid (UPM). This work has been presented at the Workshop on Static Analysis [BCHP95] associated to the International Conference of Logic Programming (ICLP) in 1995 and an improved version published at the European Symposium on Programming (ESOP) in 1996 [BCHP96].

- Multiple specialization has been studied in depth from a theoretical point of view, but is has never been implemented nor evaluated experimentally. We have developed a multiple specialization framework which is equivalent to the most powerful existing one. We have implemented and integrated such framework into a parallelizing compiler. The experimental results show that multiple specialization is a relevant technique in practice. Parts of this work have been presented at the ACM Symposium on Partial Evaluation and Semantic Program Manipulation (PEPM) in 1995 [PH95], at the Dagstuhl Seminar on Partial Evaluation [HP96], at the International Workshop on Synthesis and Transformation of Logic Programs (LoPSTr) in 1996 [PH97b], and in the fifth International Workshop on Meta-Programming and Meta-Reasoning in Logic (META) in 1996 [PH96a]. A combination of such works is to appear in the Journal of Logic Programming (JLP) [PH97a].

- Important contributions have been performed towards the integration of partial evaluation in the abstract multiple specialization framework mentioned previously. We have proposed an algorithm which allows obtaining a specialized program from an analysis graph obtained by means of abstract interpretation. We have compared the specialization capabilities of this method with those achievable by traditional partial evaluation techniques. We have explained how abstract interpretation helps in solving two of the problems of partial evaluation: that the resulting program is "covered", and that the specialization process terminates. We have identified for the first time how the problems of local and global control usually considered in partial evaluation of logic programs appear in the setting of abstract interpretation. This work has been performed in collaboration with John Gallagher, from Bristol University and has been presented at the Workshop on Specialization of Declarative Programs [PGH97] associated to the International Symposium on Logic Programming (ILPS) in 1997.

- Two kinds of program transformation techniques are presented which aim at reducing the additional cost introduced by dynamic scheduling while preserving the semantics of the original program. Previous optimization techniques mainly aimed at eliminating dynamic scheduling, but as they did not ensure that the operational semantics of the program is preserved, it may be the case that the optimized program is less efficient than the original one. The proposed techniques have been implemented using information obtained by static analysis. The experimental results obtained show that the proposed techniques may produce significant performance improvements. Preliminary results have been published as a poster at the

International Symposium on Programming, Implementation, Logics, and Programs (PLILP) in 1996 [PH96b]. A more thorough study, performed in collaboration with María García de la Banda and Kim Marriott from Monash University and Peter Stuckey from the University of Melbourne, has been presented at the Workshop on Abstract Interpretation of Logic Languages (WAILL) in 1997 [PGH^{+}96] and published at the International Conference on Logic Programming (ICLP) in 1997 [PGMS97].

- We have defined an assertion language which allows the communication among the different tools which may exist in an advanced environment for the development and debugging of (constraint) logic programs. The assertion language is parametric w.r.t. the constraint domain, the specific implementation of the language, and the kind of properties which each tool makes use of. We have studied the possibility of checking assertions at compile-time by means of static analysis and we have given an example framework for run-time checking of assertions. This work has been performed in collaboration with Francisco Bueno and has been presented at the Workshop on Tools and Environments for (Constraint) Logic Programming [PBH97] associated to the International Symposium on Logic Programming (ILPS) in 1997.

- We have performed a detailed study of the role of semantic approximations in program debugging. During program development, a usual question is whether the program satisfies the properties expected from it (requirements) or not. Usually, requirements are incomplete and they are often given my means of approximations. Thus, program validation and debugging must be able to deal with approximations of both the actual and intended semantics of the program. We have studied the kind of approximations which may be used in order to obtain conclusions about the usual questions which appear in the different tools involved in program validation and debugging. This work has been performed in collaboration with Francisco Bueno, from the Technical University of Madrid (UPM); Pierre Deransart, from INRIA Rocquencourt; Wlodek Drabent and Jan Małuszyński, from Linköping University; and with Gerard Ferrand from the University of Orleans. This study has been presented at the International Workshop on Automated and

Algorithmic Debugging (AADEBUG) in 1997 [BDD$^+$97] as an invited talk.

Part I

Advanced Techniques for Program Analysis

Chapter 2

Incremental Analysis

Global analyzers traditionally read and analyze the entire program at once, in a non-incremental way. However, there are many situations which are not well suited to this simple model and which instead require reanalysis of certain parts of a program which has already been analyzed. In these cases, it appears inefficient to perform the analysis of the program again from scratch, as needs to be done with current systems. We describe how the fixpoint algorithms used in current generic analysis engines can be extended to support incremental analysis. The possible changes to a program are classified into three types: addition, deletion, and arbitrary change. For each one of these, we provide one or more algorithms for identifying the parts of the analysis that must be recomputed and for performing the actual recomputation. The potential benefits and drawbacks of these algorithms are discussed. Finally, we present some experimental results obtained with an implementation of the algorithms in the PLAI generic abstract interpretation framework. The results show significant benefits when using the proposed incremental analysis algorithms.

2.1 Introduction

Global program analysis is becoming a practical tool in (constraint) logic program compilation in which information about calls, answers, and substitutions at different program points is computed statically [HWD92, VD92, MH92, SCWY91, BGH94b, Deb89a, Bru91, Deb92, MSJ94, CV94]. The underlying theory, formalized in terms of abstract interpretation [CC77], and the related implementation techniques are well understood for several general types of analysis and, in particular, for top-down analysis of Prolog [Deb89a, Bru91, MH92, Deb92, MSJ94, CV94].

Several generic analysis engines, such as PLAI [MH92, MH90a], GAIA [CV94] and the CLP($\mathcal{R}$) analyser described in [KMSSar], facilitate construction of such top-down analyzers. These generic engines have the description domain and functions on this domain as parameters. Different domains give analyzers which provide different types of information and degrees of accuracy. The core of each generic engine is an algorithm for efficient fixpoint computation [MH90a, MH92, CDMV93]. Efficiency is obtained by keeping track of which parts of a program must be reexamined when a success pattern is updated. Current generic analysis engines are non-incremental—the entire program is read, analyzed and the analysis results written out.

Despite the obvious progress made in global program analysis, most LP and CLP compilers still perform only local analysis (the &-Prolog [HG91], Aquarius [VD92], Andorra-I [SCWY91] and CLP($\mathcal{R}$) [KMM+95, KMM+96] systems are notable exceptions). We believe that an important contributing factor to this is the simple, non-incremental model supported by global analysis systems, which is unsatisfactory for at least four reasons. The first reason is that optimizations are often source-to-source transformations, and so optimization consists of an analyze, perform optimization then reanalyze cycle. This is inefficient if the analysis starts from scratch each time. Such analyze-optimize cycles may occur for example when program optimization and multivariant specialization are combined [Win92, PH95] (see Chapter 5 for details). This is used, for instance, in program parallelization, where an initial analysis is used to introduce specialized predicate definitions with run-time parallelization tests, and then these new definitions are analyzed and redundant tests removed. This is also the case in optimization of CLP($\mathcal{R}$) in which specialized predicate definitions are reordered and then reanalyzed. The second reason is that incremental analysis supports incremental run-time compilation during the test-debug cycle. Again, for efficiency only those parts of the program which are affected by the changes should be reanalyzed. Incremental compilation is important in the context of logic programs as traditional environments have been interpretive, allowing the rapid generation of prototypes. The third reason is to handle correctly and accurately the optimization of programs in which clauses are asserted or retracted at run-time. The fourth reason is to support incremental compilation of programs broken into modules.

In this chapter we describe how the fixpoint algorithm in the generic analysis engines can be extended to support incremental analysis. Guided by the applications mentioned above, we consider algorithms for different types of incrementality. The first, and simplest type of incrementality is when program clauses are added to the original program. The second type of incrementality is clause deletion. We give several algorithms to handle deletion. These capture different tradeoffs between efficiency and accuracy. The algorithms for deletion can be easily extended to handle the third and most general type of incrementality, arbitrary change, in which program clauses can be deleted or modified in any way. Finally, we consider a restricted type of arbitrary change: local change in which clauses are modified, but the answers to the clauses are unchanged for the calling patterns they are used with. This case occurs in program optimization as correctness of the optimization usually amounts to requiring this property. Local change means that changes to the analysis are essentially restricted to recomputing the new call patterns which these clauses generate. We give an algorithm which handles this type of incrementality. Finally we describe an implementation of the algorithms in the PLAI system and give a preliminary empirical evaluation. We argue that the experimental results show that our algorithms are practically important.

Surprisingly, there has been little research into incremental analysis for (constraint) logic programs. Several researchers have looked at compositional analysis of modules in (constraint) logic programs [CDG93]. There has been much research into incremental analysis for other programming paradigms (see for example the bibliography of Ramalingam and Reps [RR93]). However, to our knowledge this is the first work to identify the different types of incremental change which are useful in logic program analysis and to give practical algorithms which handle these types of incremental change. Another contribution of this work is a generalization of the non-incremental fixpoint algorithms used in generic analysis engines. We formalize the analysis as a graph traversal and couch the algorithm in terms of priority queues. Different priorities correspond to different traversals of the program analysis graph. This simple formalization greatly facilitates the description of our incremental algorithms and their proofs of correctness.

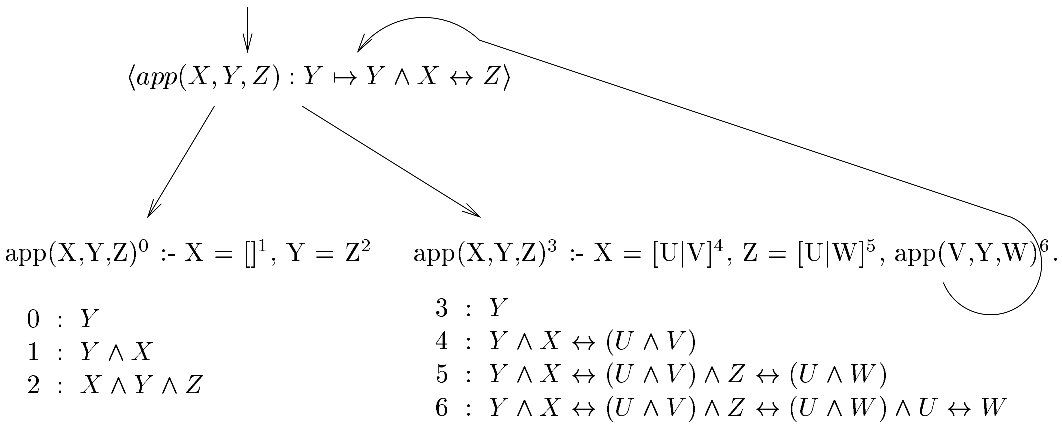

Figure 2.1: Example Program Analysis Graph

2.2 A Generic Analysis Algorithm

As mentioned above, we start by providing a formalization of a fixpoint algorithm which generalizes those used in a good number of the existing generic analysis engines. The purpose of our presentation is not so much to present a practical algorithm for performing program analysis but rather to capture the core behaviour of the standard algorithms (whose exact description would be too involved for the space available), and then use this stylized algorithm to present our proposals regarding how to make them incremental.

The aim of the kind of (goal oriented) program analysis performed by the above mentioned engines is, for a particular description domain, to take a program and a set of initial calling 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 calling patterns.

To illustrate this, we now develop an example. The description domain, as in all of our examples, will be the *definite Boolean functions* [AMSS94]. The key idea in this description is to use implication to capture groundness dependencies. The reading of the function $x \rightarrow y$ is: “if the program variable x is (becomes) ground, so is (does) program variable y.” For example, the best description of the constraint $f(X,Y) = f(a, g(U,V))$ is $X \wedge Y \leftrightarrow (U \wedge V)$.

Now consider the program for appending lists:

```
app(X,Y,Z) :- X=[], Y=Z.
```

```
app(X,Y,Z) :- X=[U|V], Z=[U|W], app(V,Y,W).
```

Assume that we are interested in analyzing the program for the call $app(X, Y, Z)$ with initial calling pattern Y indicating that we wish to analyze it for any call to app with the second argument definitely ground. In essence the analyzer must produce the *program analysis graph* given in Figure 2.1, which can be viewed as a finite representation (through a "widening") of the set of AND-OR trees explored by the concrete execution [Bru91]. The graph has two sorts of nodes: those belonging to rules (also called "AND-nodes") and those belonging to atoms (also called "OR-nodes"). For example, the atom node $\langle app(X, Y, Z) : Y \mapsto Y \wedge (X \leftrightarrow Z)\rangle$ indicates that the calling pattern Y for the atom $append(X, Y, Z)$ has answer pattern $Y \wedge (X \leftrightarrow Z)$. This answer pattern depends on the two rules defining *app* which are attached by arcs to the node. These rules are annotated by descriptions at each program point of the constraint store when the rule is executed from the calling pattern of the node connected to the rules. The program points are entry to the rule, the point between each two literals, and return from the call. Atoms in the rule body have arcs to OR-nodes with the corresponding calling pattern. If such a node is already in the tree it becomes a recursive call. Thus, the analysis graph in Figure 2.1 has a recursive call to the calling pattern $app(X, Y, Z) : Y$.

A program analysis graph is defined in terms of an initial set of calling patterns, a program, and five abstract operations on the description domain. The abstract operations are:

- $Aproject(CP, V)$ which performs the abstract restriction of a calling pattern CP to the variables in the set V;
- $Aextend(CP, V)$ which extends the description CP to the variables in the set V;
- $Aadd(C, CP)$ which performs the abstract operation of conjoining the actual constraint C with the description CP;
- $Acombine(CP_1, CP_2)$ which performs the abstract conjunction of two descriptions;
- $Alub(CP_1, CP_2)$ which performs the abstract disjunction of two descriptions.

For a given program and calling pattern there may be many different analysis graphs. However, for a given set of initial calling patterns, a program and abstract operations on the descriptions, there is a unique *least analysis graph* which gives the most precise information possible. This analysis graph corresponds to the least fixpoint of the abstract semantic equations.

We now give an algorithm which computes the least analysis graph. We first introduce some notation. CP, possibly subscripted, stands for a calling pattern (in the abstract domain). AP, possibly subscripted, stands for an answer pattern (in the abstract domain). Each literal in the program is subscripted with an identifier or pair of identifiers. $A : CP$ stands for an atom (unsubscripted) together with a calling pattern. $A_k : CP$ or $A_{k,i} : CP$ stands for subscripted literals together with a calling pattern. Rules are assumed to be normalized and each rule for a predicate p has identical sets of variables $p(x_{p_1}, \ldots x_{p_n})$ in the head atom. Call this the *base form* of p. Rules in the program are written with a unique subscript attached to the head atom (the rule number), and dual subscript (rule number, body position) attached to each body atom (and constraint redundantly) e.g. $H_k \leftarrow B_{k,1}, \ldots, B_{k,n_k}$ where $B_{k,i}$ is a subscripted atom or constraint. The rule may also be referred to as rule k, the subscript of the head atom. E.g. the append program is written:

$\texttt{app}_1\texttt{(X,Y,Z) :- X=[]}_{1,1}\texttt{, Y=Z}_{1,2}\texttt{.}$

$\texttt{app}_2\texttt{(X,Y,Z) :- X=[U|V]}_{2,1}\texttt{, Z=[U|W]}_{2,2}\texttt{, app}_{2,3}\texttt{(V,Y,W).}$

The program analysis graph is implicitly represented in the algorithm by means of two data structures, the *answer table* and the *dependency arc table*. The answer table contains entries of the form $A : CP \mapsto AP$. A is always a base form. This represents a node in the analysis graph of the form $\langle A : CP \mapsto AP \rangle$. It is interpreted as the answer pattern for calls of the form CP to A is AP. A dependency arc is of the form $H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$. This is interpreted as that if the rule with H_k as head is called with calling pattern CP_0 then this causes literal $B_{k,i}$ to be called with calling pattern CP_2. The remaining part CP_1 is the program annotation just before $B_{k,i}$ is reached and contains information about all variables in rule k. CP_1 is not really necessary, but is included for efficiency. Dependency arcs represent the arcs in the program analysis graph from atoms in a rule body to an atom node. E.g. the program analysis graph in Figure 2.1 is represented by *answer table:* $\quad$ `app(X, Y, Z)` $: Y \mapsto Y \wedge (X \leftrightarrow Z)$

dependency arc table:

$$\texttt{app}_2\texttt{(X,Y,Z)}: Y \Rightarrow [Y \wedge X \leftrightarrow (U \wedge V) \wedge Z \leftrightarrow (U \wedge W)] \;\; \texttt{app}_{2,3}\texttt{(V,Y,W)}: Y$$

Intuitively, the analysis algorithm is just a graph traversal algorithm which places entries in the answer table and dependency arc table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixpoint algorithms, we use a priority queue. Thus, the third, and final structure used in our algorithms is a *prioritized event queue*. The priority mechanism for the queue is left as a parameter of the algorithm. Events are of three forms:

- *updated*$(A : CP)$ which indicates that the answer pattern to atom A with calling pattern CP has been changed.
- *arc*(R) which indicates that the rule referred to in R needs to be (re)computed from the position indicated.
- *newcall*$(A : CP)$ which indicates that a new call has been encountered.

The generic analysis algorithm is given in Figure 2.2. Apart from the parametric description domain dependent functions, the algorithm has several other undefined functions. The function $vars$ returns the variables appearing in some program part. The functions add_event and next_event respectively add an event to the priority queue and return (and delete) the event of highest priority. When an event being added to the priority queue is already in the priority queue, a single event with the maximum of the priorities is kept in the queue. When an arc $H_k : CP \Rightarrow [CP'']B_{k,i} : CP'$ is added to the dependency arc table, it overwrites any other arcs of the form $H_k : CP \Rightarrow [\_]B_{k,i} : \_$ in the table and in the priority queue. The function initial_guess returns an initial guess for the answer to a new calling pattern. The default value is $\perp$ but if the calling pattern is more general than an already computed call then its current value may be returned. The procedure remove_useless_calls traverses the dependency graph given by the dependency arcs from the initial calls S and marks those entries in the dependency arc and answer table which are reachable. The remainder are removed.

```
analyze(S)
  foreach A : CP ∈ S
    add_event(newcall(A : CP))
  main_loop()

main_loop()
  while E := next_event()
    if (E == newcall(A : CP))
      new_calling_pattern(A : CP)
    elseif (E == updated(A : CP))
      add_dependent_rules(A : CP)
    elseif (E == arc(R))
      process_arc(R)
  endwhile
  remove_useless_calls(S)

new_calling_pattern(A : CP)
  foreach rule A_k ← B_{k,1}, ..., B_{k,n_k}
    CP_0 :=
      Aextend(CP, vars(B_{k,1}, ..., B_{k,n_k}))
    CP_1 := Aproject(CP_0, vars(B_{k,1}))
    add_event(arc(
      A_k : CP ⇒ [CP_0] B_{k,1} : CP_1))
  AP := initial_guess(A : CP)
  if (AP <> ⊥)
    add_event(updated(A : CP))
  add A : CP ↦ AP to answer_table

add_dependent_rules(A : CP)
  foreach arc of the form
      H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2
      in graph
    where there exists renaming σ
      s.t. A : CP = (B_{k,i} : CP_2)σ
    add_event(arc(
      H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2))

process_arc(H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2)
  if (B_{k,i} is not a constraint)
    add H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2
    to dependency_arc_table
  AP_0 := get_answer(B_{k,i} : CP_2)
  CP_3 := Acombine(CP_1, AP_0)
  if (CP_3 <> ⊥ and i <> n_k)
    CP_4 := Aproject(CP_3, vars(B_{k,i+1}))
    add_event( arc(
      H_k : CP_0 ⇒ [CP_3] B_{k,i+1} : CP_4))
  elseif (CP_3 <> ⊥ and i == n_k)
    AP_1 := Aproject(CP_3, vars(H_k))
    insert_answer_info(H : CP_0 ↦ AP_1)

get_answer(L : CP)
  if (L is a constraint)
    return Aadd(L, CP)
  else  return lookup_answer(L, CP)

lookup_answer(A : CP)
  if (there exists a renaming σ s.t.
    σ(A : CP) ↦ AP in answer_table)
    return σ^{-1}(AP)
  else
    add_event(newcall(σ(A : CP)))
    where σ is a renaming s.t.
    σ(A) is in base form
    return ⊥

insert_answer_info(H : CP ↦ AP)
  AP_0 := lookup_answer(H : CP)
  AP_1 := Alub(AP, AP_0)
  if (AP_0 <> AP_1)
    add (H : CP ↦ AP_1) to answer_table
    add_event(updated(H : CP))
```

Figure 2.2: A generic analysis algorithm

2.2.1 Example Execution of the Generic Algorithm

The generic algorithm presented in Figure 2.2 essentially represents standard fixpoint algorithms, and as such is given in this work only the amount of description needed to understand the algorithms for incremental analysis. The following example, illustrates how the `app` program would be analyzed.

The abstract operations for the description domain Def of definite Boolean functions are defined as follows. The abstraction operation α_{Def} gives the best description of a constraint. It is defined as: $\alpha_{Def}(X = f(Y_1, \ldots, Y_n)) = X \leftrightarrow (Y_1 \wedge \cdots \wedge Y_n)$, and $\alpha_{Def}(e_1 \wedge \cdots \wedge e_k) = \alpha_{Def}(e_1) \wedge \cdots \wedge \alpha_{Def}(e_k)$ where $e_1, \ldots, e_k$ are term equations. The remaining operations are defined as follows:

$$\begin{aligned}
\mathsf{Aproject}(CP, V) &= \exists_{-V} CP \\
\mathsf{Aextend}(CP, V) &= CP \\
\mathsf{Aadd}(C, CP) &= \alpha_{Def}(C) \wedge CP \\
\mathsf{Acombine}(CP_1, CP_2) &= CP_1 \wedge CP_2 \\
\mathsf{Alub}(CP_1, CP_2) &= CP_1 \vee CP_2
\end{aligned}$$

Example 2.2.1 Analysis begins from an initial set S of calls. In our example S contains `app(X,Y,Z)`:Y. The first step in the algorithm is to add the initial calls as new calls to the priority queue. After this the priority queue contains

`newcall( app(X,Y,Z)`:Y`)`

and the answer and dependency arc tables are empty. The *newcall* event is taken from the event queue and processed as follows. For each rule defining `app`, an arc is added to the priority queue which indicates that the rule body must be processed from the initial literal. An entry for the new call is added to the answer table with an initial guess of $\perp$ as the answer. The data structures are now:

priority queue:	`arc( app`$_1$`(X, Y, Z)` : $Y \Rightarrow [Y]$ `X=[]`$_{1,1}$: *true*)
	`arc( app`$_2$`(X, Y, Z)` : $Y \Rightarrow [Y]$ `X=[U|V]`$_{2,1}$: *true*)
answer table:	`app(X, Y, Z)` : $Y \mapsto \perp$
dependency arc table:	no entries

An arc on the event queue is now selected for processing, say the first. The routine `get_answer` is called to find the answer pattern to the literal `X=[]` with calling pattern *true*. As the literal is a constraint, the parametric routine *Aadd* is used. It returns the answer pattern X. This is combined using *Acombine* with the initial annotation Y to give $X \wedge Y$, which is the next annotation in the rule body. A new arc is added to the priority queue which indicates that the second literal in the rule body must be processed. The priority queue is now:

$$\texttt{arc(app}_1\texttt{(X, Y, Z)} : Y \Rightarrow [X \wedge Y]\ \texttt{X=Z}_{1,2} : X)$$
$$\texttt{arc(app}_2\texttt{(X, Y, Z)} : Y \Rightarrow [Y]\ \texttt{X=[U|V]}_{2,1} : \ true).$$

The answer and dependency arc table remain the same.

Again, an arc on the event queue is selected for processing, say the first. As before, `get_answer`, *Aadd* and *Acombine* are called to obtain the next annotation $X \wedge Y \wedge Z$. This time, as there are no more literals in the body, the answer table entry for $app(X, Y, Z) : Y$ is updated. *Alub* is used to lub the new answer $X \wedge Y \wedge Z$ with the old answer $\perp$. This gives $X \wedge Y \wedge Z$. The entry in the answer table is updated and an *updated* event is placed on the priority queue. The data structures are now:

priority queue:	$\texttt{updated(app(X, Y, Z)} : Y)$
	$\texttt{arc(app}_2\texttt{(X, Y, Z)} : Y \Rightarrow [Y]\ \texttt{X=[U\|V]}_{2,1} : \ true)$
answer table:	$\texttt{app(X, Y, Z)} : Y \mapsto X \wedge Y \wedge Z$
dependency arc table:	no entries

The *updated* event can now be processed. As there are no entries in the dependency arc table, nothing in the current program analysis graph depends on the answer to this call, so nothing needs to be recomputed. The priority queue now contains

$$\texttt{arc(app}_2\texttt{(X, Y, Z)} : Y \Rightarrow [Y]\ \texttt{X=[U|V]}_{2,1} : true).$$

The answer and dependency arc table remain the same.

Similarly to before we process the arc, giving rise to the new priority queue

$$\texttt{arc(app}_2\texttt{(X, Y, Z)} : Y \Rightarrow [Y \wedge X \leftrightarrow (U \wedge V)]\ \texttt{Z=[U|W]}_{2,2} : true).$$

The arc, is processed to give the priority queue

$$\texttt{arc(app}_2\texttt{(X,Y,Z)}\text{:}\ Y \Rightarrow [Y \wedge X \leftrightarrow (U \wedge V) \wedge Z \leftrightarrow (U \wedge W)]\ \texttt{app}_{2,3}\texttt{(V,Y,W)}\text{:}\ Y)$$

This time, because $\texttt{app}_{2,3}\texttt{(V,Y,W)}$ is an atom, the arc is added to the arc dependency table. The answer table is looked up to find the answer and, appropriately renamed, the next annotation is $Y \wedge V \wedge W \wedge X \leftrightarrow U \wedge Z \leftrightarrow U$. As this is the last literal in the body, the new answer $Y \wedge X \leftrightarrow Z$ is obtained. This is lubbed with the old answer in the table, giving $Y \wedge X \leftrightarrow Z$. As the answer has changed an *updated* event is added to the priority queue. The data structures are now:

priority queue: `updated( app(X, Y, Z)` $: Y)$

answer table: `app(X, Y, Z)` $: Y \mapsto Y \wedge (X \leftrightarrow Z)$

dependency arc table: $\texttt{app}_2\texttt{(X, Y, Z)} : Y \Rightarrow$
$[Y \wedge X \leftrightarrow (U \wedge V) \wedge Z \leftrightarrow (U \wedge W)]$ $\texttt{app}_{2,3}\texttt{(V,Y,W)}$: Y

The updated event is processed by looking in the dependency arc table for all arcs which have a body literal which is a variant of $app(X, Y, Z) : Y$ and adding these arcs to the priority queue to be reprocessed. We obtain the new priority queue

$\texttt{arc(app}_2\texttt{(X,Y,Z)}$: $Y \Rightarrow [Y \wedge X \leftrightarrow (U \wedge V) \wedge Z \leftrightarrow (U \wedge W)]$ $\texttt{app}_{2,3}\texttt{(V,Y,W)}$: $Y)$

This arc is reprocessed, and gives rise to the answer $Y \wedge X \leftrightarrow Z$. This is lubbed with the old answer, and the result is identical to the old answer. Thus, no *updated* event is added to the priority queue. As there are no events on the priority queue, the analysis terminates with the desired answer and dependency arc table.

It is also important to remember the purpose of this algorithm. It is not intended as a practical algorithm for computing a program analysis graph, as the overhead of event handling is too high. Rather it is intended to capture the behaviour of several algorithms which are used for computing the program analysis graph. Different algorithms correspond to different event processing strategies. In addition, practical algorithms incorporate a series of optimizations. For example, one strategy would be to always perform *newcall* events first, to process non-recursive rules before recursive rules, and to finish processing a rule before starting another. This strategy would produce an algorithm which is quite close to the one used in PLAI or GAIA (the differences between the proposed algorithm and that used in PLAI are presented in more detail in Section 2.7).

Theorem 2.2.2 For a program P and call patterns S, the generic analysis algorithm returns an answer table and dependency arc table which represents the least program analysis graph of P and S.

```
incremental_addition(R)
    foreach rule A_k ← B_{k,1}, ..., B_{k,n_k} ∈ R
        foreach entry A : CP ↦ AP in the answer_table
            CP_0 := Aextend(CP, vars(B_{k,1}, ..., B_{k,n_k}))
            CP_1 := Aproject(CP_0, vars(B_{k,1}))
            add_event(arc(A_k : CP ⇒ [CP_0] B_{k,1} : CP_1))
    main_loop()
```

Figure 2.3: Incremental Addition Algorithm

The corollary of this is that the priority strategy does not involve correctness of the analysis. This corollary will be vital when arguing correctness of the incremental algorithms in the following sections.

Corollary 2.2.3 The result of the generic analysis algorithm does not depend on the strategy used to prioritize events.

2.3 Incremental Addition

Since the answer and dependency arc tables are incrementally extended in the generic analysis of a program, incremental addition of new rules and new calling patterns does not place extra demands on the generic analysis algorithm. If the analysis is required for new calling patterns, then the routine `analyze(S)`, where `S` is the set of new calling patterns may be repeatedly called.

The new routine for analysis of programs in which rules are added incrementally is given in Figure 2.3. The routine takes as input the set of new rules R. If these define a calling pattern of interest, then requests to process the rule are placed on the priority queue. Subsequent processing is exactly as for the non-incremental case.

Example 2.3.1 As an example, we begin with the program for naive reverse, `rev`, already analyzed for the calling pattern *true* but without a definition of the append, `app`, predicate. The initial program is

[A] $\texttt{rev}_1\texttt{(X,Y) :- X = []}_{1,1}\texttt{, Y = []}_{1,2}\texttt{.}$
[B] $\texttt{rev}_2\texttt{(X,Y) :- X = [U|V]}_{2,1}\texttt{, rev}_{2,2}\texttt{(V,W), T = [U]}_{2,3}\texttt{, app}_{2,4}\texttt{(W,T,Y).}$

The answer table and dependency arc tables are (**State 1**):

answer table: $\texttt{rev(X, Y)} : \mathit{true} \mapsto X \wedge Y$

$\texttt{app(X, Y, Z)} : X \mapsto \bot$

dependency arc table:

$\texttt{rev}_2\texttt{(X,Y)} : \mathit{true} \Rightarrow [X \leftrightarrow (U \wedge V)]\ \ \texttt{rev}_{2,2}\texttt{(V,W)} : \mathit{true}$

$\texttt{rev}_2\texttt{(X,Y)} : \mathit{true} \Rightarrow [(X \leftrightarrow (U \wedge V)) \wedge V \wedge W \wedge (T \leftrightarrow U)]\ \ \texttt{app}_{2,4}\texttt{(W,T,Y)} : W$

We now add the clauses for `app` one at a time. The first clause to be added is

[C] $\texttt{app}_3\texttt{(X, Y, Z) :- X = []}_{3,1}\texttt{, Y = Z}_{3,2}\texttt{.}$

The incremental analysis begins by looking for entries referring to *app* in the answer table. It finds the entry $\mathit{app}(X, Y, Z) : X$ so the arc $\texttt{app}_3\texttt{(X,Y,Z)} : X \Rightarrow [X]\ \ \texttt{X = []}_{3,1} : X$ is put in the priority queue. After processing this rule, the new answer $X \wedge (Y \leftrightarrow Z)$ for $\texttt{app(X,Y,Z)}$: X is obtained. This is lubbed with the current answer to obtain $X \wedge (Y \leftrightarrow Z)$ and the answer table entry is updated (causing an *updated*($\texttt{app(X,Y,Z)} : X$) event). Examining the dependency arc table, the algorithm recognizes that the answer from clause [B] must now be recomputed. This gives rise to the new answer $(X \leftrightarrow (U \wedge V) \wedge V \wedge W \wedge (U \leftrightarrow Y))$ which simplifies to $X \leftrightarrow Y$. Lubbing this with the current answer $X \wedge Y$ gives $X \leftrightarrow Y$. The memo table entry for $\texttt{rev(X, Y)} : \mathit{true}$ is updated appropriately, and an *updated* event is placed on the queue. Again the answer to clause [B] must be recomputed. First we obtain a new calling pattern for $\texttt{app}_{2,4}\texttt{(X, Y, Z)}$ of true. This means that the dependency arc

$$\texttt{rev}_2\texttt{(X,Y)} : \mathit{true} \Rightarrow [(X \leftrightarrow (U \wedge V)) \wedge V \wedge W \wedge (T \leftrightarrow U)]\ \texttt{app}_{2,4}\texttt{(W,T,Y)} : W$$

in the dependency arc table is replaced by

$$\texttt{rev}_2\texttt{(X,Y)} : \mathit{true} \Rightarrow [(X \leftrightarrow (U \wedge V) \wedge (V \leftrightarrow W) \wedge (T \leftrightarrow U)]\ \texttt{app}_{2,4}\texttt{(W,T,Y)} : \mathit{true}$$

This sets up a new call $\texttt{app(X, Y, Z)} : \mathit{true}$. The current answer for the old call $\texttt{app(X, Y, Z)} : X$ can be used as an initial guess to the new, more general, call. The algorithm examines clause [C] for the new calling pattern. It obtains the same answer $X \wedge (Y \leftrightarrow Z)$.

This leads to a new answer for $\texttt{rev}_2\texttt{(X, Y)}$, $(X \leftrightarrow (U \wedge V)) \wedge (V \leftrightarrow W) \wedge W \wedge (T \leftrightarrow U) \wedge (T \leftrightarrow Y)$, which simplifies to $X \leftrightarrow Y$. This does not change the current answer so the main loop of the analysis is finished. The reachability analysis removes the entry `app(X, Y, Z)` : $X \mapsto X \wedge (Y \leftrightarrow Z)$ from the answer table. The resulting answer and dependency arc table entries are (**State 2**):

answer table: `rev(X, Y)` : $true \mapsto X \leftrightarrow Y$
`app(X, Y, Z)` : $true \mapsto X \wedge (Y \leftrightarrow Z)$

dep. arc table:

$\texttt{rev}_2\texttt{(X,Y)}$: $true \Rightarrow [X \leftrightarrow (U \wedge V)]$ $\texttt{rev}_{2,2}\texttt{(V,W)}$: $true$

$\texttt{rev}_2\texttt{(X,Y)}$: $true \Rightarrow [X \leftrightarrow (U \wedge V) \wedge (V \leftrightarrow W) \wedge (T \leftrightarrow U)]$ $\texttt{app}_{2,4}\texttt{(W,T,Y)}$: $true$

If the second clause for `app`

[D] $\texttt{app}_4\texttt{(X,Y,Z) :- X = [U|V]}_{4,1}\texttt{, Z = [U|W]}_{4,2}\texttt{, app}_{4,3}\texttt{(V,Y,W).}$

is added, the analysis proceeds similarly. The final memo and dependency arc table entries are (**State 3**):

answer table: `rev(X, Y)` : $true \mapsto X \leftrightarrow Y$
`app(X, Y, Z)` : $true \mapsto (X \wedge Y) \leftrightarrow Z$

dep. arc table:

(1) $\texttt{rev}_2\texttt{(X, Y)}$: $true \Rightarrow [X \leftrightarrow (U \wedge V)]$ $\texttt{rev}_{2,2}\texttt{(V,W)}$: $true$

(2) $\texttt{rev}_2\texttt{(X, Y)}$: $true \Rightarrow$
$[X \leftrightarrow (U \wedge V) \wedge (V \leftrightarrow W) \wedge (T \leftrightarrow U)]$ $\texttt{app}_{2,4}\texttt{(W,T,Y)}$: $true$

(3) $\texttt{app}_4\texttt{(X,Y,Z)}$: $true \Rightarrow [X \leftrightarrow (U \wedge V) \wedge Z \leftrightarrow (U \wedge W)]$ $\texttt{app}_{4,3}\texttt{(V,Y,W)}$: $true$

Correctness of the incremental addition algorithm follows from correctness of the original generic algorithm. Execution of the incremental addition algorithm corresponds to executing the generic algorithm with all rules but with the new rules having the lowest priority for processing.

Theorem 2.3.2 If the rules in a program are analyzed incrementally with the incremental addition algorithm, the same answer and dependency arc tables will be obtained as when all rules are analyzed at once by the generic algorithm.

In a sense, therefore, the cost of performing the analysis incrementally can be no worse than performing the analysis all at once, as the generic analysis could have used a priority strategy which has the same cost as the incremental strategy.

We will now formalize this intuition. Our cost measure will be the number of calls to the underlying parametric functions. This is a fairly simplistic measure, but our results will continue to hold for reasonable measures.

Let $C_{noninc}(\bar{F}, R, S)$ be the worst case number of calls to the parametric functions $\bar{F}$ when analyzing the rules R and call patterns S for all possible priority strategies with the generic analysis algorithm.

Let $C_{add}(\bar{F}, R, R', S)$ be the worst case number of calls to the parametric functions $\bar{F}$ when analyzing the new rules R' for all possible priority strategies with the incremental addition algorithm after already analyzing the program R for call patterns S.

Theorem 2.3.3 Let the set of rules R be partitioned into $R_1, ..., R_n$ rule sets. For any call patterns S and parametric functions $\bar{F}$,

$$C_{noninc}(\bar{F}, R, S) \geq \sum_{i=1}^{n} C_{add}(\bar{F}, (\bigcup_{j=1}^{j<i} R_j), R_i, S).$$

2.4 Incremental Deletion

In this section we consider deletion of clauses from an already analyzed program and how to incrementally update the analysis information. The first thing to note is that we need not change the analysis results at all. The current approximation is trivially guaranteed to be correct. This approach is obviously inaccurate but simple. More accuracy can be obtained by applying a *narrowing* like strategy. Starting the analysis from scratch will often give a more accurate result. We first discuss a narrowing-like strategy and then present in detail two algorithms which are incremental yet are as accurate as the non-incremental analysis.

2.4.1 Narrowing-like Strategy

More accuracy can be obtained by applying a *narrowing*-like strategy. The current approximation in the answer table is greater than the least fixpoint of the semantic equations. Thus applying the analysis engine as usual except taking the greatest lower bound of new answers with the old rather than the least upper bound is guaranteed to produce a correct, albeit perhaps imprecise result. Again we let this process be guided from the initial changes using the dependency graph information. Care must be taken with treating new calling patterns that arise in this process correctly.

Example 2.4.1 Consider the program in Example 2.3.1 after both additions. The current memo table and dependency graph entries are given by **State 3**. Deleting clause [D] results in the following process.

First we delete any dependency arcs which correspond to deleted clauses. In this case we remove the arc $\texttt{app}_4\texttt{(X, Y, Z)} : true \Rightarrow [\_]\ \texttt{app}_{4,1}\texttt{(V, Y, W)} : true$. In general we may subsequently delete other dependency arcs which are no longer required.

We recompute the answer information for all (remaining) clauses for `app(X, Y, Z)` for all calling patterns of interest using the current answer information. We obtain `app(X, Y, Z)` $: true \mapsto X \wedge (Y \leftrightarrow Z)$.

Because this information has changed we now need to consider recomputing answer information for any calling patterns that depend on `app(X, Y, Z)` $: true$, in this case `rev(X, Y)` $: true$. Recomputing using clauses [A] and [B] obtains the same answer information $X \leftrightarrow Y$. The result is **State 2** (with the useless entry for `app(X, Y, Z)`: X removed).

Deleting clause [C] subsequently leads back to **State 1** as expected. In contrast removing clause [C] from [A,B,C,D] does not result in recovering **State 1** as might be expected. This highlights the possible inaccuracy of the narrowing-like strategy. In this case clause [D] prevents more accurate answer information from being acquired.

The above example does not illustrate how new calling patterns are handled by the narrowing method. New calling patterns are guaranteed to be more informative than the patterns before deletion, hence we can correctly use the join of the new call pattern with the answer pattern for a more general call to abstract any new calling patterns.

Example 2.4.2 Consider the program

$\texttt{q}_1$ `:-` $\texttt{p}_{1,1}$`(X, Y),` $\texttt{r}_{1,2}$`(X, Y, Z).`
$\texttt{p}_2$`(X, Y) :- X = a, Y = b.`
$\texttt{p}_3$`(X, Y) :- X = Y.`
$\texttt{r}_4$`(X, Y, Z) :- X = Z.`
$\texttt{r}_5$`(X, Y, Z) :- Y = Z.`

The state of the completed analysis is

State 4
$\mathtt{q}$: $true \mapsto true$
$\mathtt{p(X,\ Y)}$: $true \mapsto X \leftrightarrow Y$
$\mathtt{r(X,\ Y,\ Z)}$: $X \leftrightarrow Y \mapsto X \leftrightarrow Y \leftrightarrow Z$
$\mathtt{q}_1$: $true \Rightarrow [true]$ $\mathtt{p}_{1,1}\mathtt{(X,\ Y)}$: $true$
$\mathtt{q}_1$: $true \Rightarrow [X \leftrightarrow Y]$ $\mathtt{r}_{1,2}\mathtt{(X,\ Y,\ Z)}$: $X \leftrightarrow Y$

Deleting the clause $\mathtt{p}_3\mathtt{(X,\ Y)}$ `:- X = Y` results in the new dependency arc $\mathtt{q}_1$: $true \Rightarrow [X \wedge Y]$ $\mathtt{r}_{1,2}\mathtt{(X,\ Y,\ Z)}$: $X \wedge Y$. In the addition process the initial answer pattern for $\mathtt{r}_{1,1}\mathtt{(X,\ Y,\ Z)}$: $X \wedge Y$ is $\perp$ but this may lead to incorrect results (say if `r` was recursive). In this case we let the initial answer be $(X \leftrightarrow Y \leftrightarrow Z) \sqcap (X \wedge Y)$ so the entry is $\mathtt{r(X,\ Y,\ Z)}$: $X \wedge Y \mapsto X \wedge Y \wedge Z$. This is also the final answer after incremental reanalysis.

2.4.2 "Top-Down" Deletion Algorithm

As mentioned before, the main disadvantage of narrowing-like strategies is the possibly inaccurate result w.r.t. restarting analysis from scratch. In this section and in Section 2.4.3 below we present two algorithms for incremental deletion which are as accurate as restarting analysis from scratch.

The first method for incremental analysis of programs after deletion is to remove all information in the answer and dependency arc tables which depends on the rules which have been deleted and then to restart the analysis. Not only will removal of rules change the answers in the answer table, it will also mean that subsequent calling patterns may change. Thus we must also remove entries for calling patterns which may no longer exist.

Information in the dependency arc table allows us to find these no longer valid call and answer patterns. Consider the dependency arcs of the analyzed program. Let D be the set of deleted rules. Let H be the set of atoms which occur as the head of a deleted rule. Let $up(H)$ be the set of atom/calling pattern pairs whose answers depended on an atom in H. That is, all of the atom/calling pattern pairs that can reach a pair $p : CP$ in the dependency graph where $p \in H$. After entries concerning these now invalid atom/calling pattern pairs are deleted, the usual

```
top_down_delete(D, S)
    H := {A|(A ← B) ∈ D}
    T := up(H)
    foreach A : CP ∈ T
        delete entry A : CP ↦ AP from answer_table
        delete each arc A_k : CP ⇒ [CP_1]B_{k,j} : CP_2 from dependency_ arc_table
    foreach A : CP ∈ S ∩ T
        add_event(newcall(A : CP))
    main_loop()
```

Figure 2.4: Top-down Incremental Deletion Algorithm

generic analysis is performed. The routine for top-down rule deletion is given in Figure 2.4. It is called with the set of deleted rules D and a set of initial calls S.

Example 2.4.3 Consider the program

$\mathtt{q}_1$ `:- ` $\mathtt{p}_{1,1}$`(X, Y), `$\mathtt{r}_{1,2}$`(X, Y, Z), `$\mathtt{s}_{1,3}$`(Y, Z).`
$\mathtt{p}_2$`(X, Y) :- X = `$\mathtt{a}_{2,1}$`, Y = `$\mathtt{b}_{2,2}$`.`
$\mathtt{p}_3$`(X, Y) :- X = `$\mathtt{Y}_{3,1}$`.`
$\mathtt{r}_4$`(X, Y, Z) :- X = `$\mathtt{Z}_{4,1}$`.`
$\mathtt{r}_5$`(X, Y, Z) :- Y = `$\mathtt{Z}_{5,1}$`.`
$\mathtt{s}_6$`(Y, Z) :- Y = `$\mathtt{c}_{6,1}$`.`

After program analysis we obtain (**State 5**):

answer table:	`q` : $true \mapsto true$
	`p(X, Y)` : $true \mapsto X \leftrightarrow Y$
	`r(X, Y, Z)` : $X \leftrightarrow Y \mapsto X \leftrightarrow Y \leftrightarrow Z$
	`s(Y, Z)` : $Y \leftrightarrow Z \mapsto Y \wedge Z$
dependency arc table:	(A) $\mathtt{q}_1$: $true \Rightarrow [true]$ $\mathtt{p}_{1,1}$`(X, Y)` : $true$
	(B) $\mathtt{q}_1$: $true \Rightarrow [X \leftrightarrow Y]$ $\mathtt{r}_{1,2}$`(X, Y, Z)` : $X \leftrightarrow Y$
	(C) $\mathtt{q}_1$: $true \Rightarrow [X \leftrightarrow Y]$ $\mathtt{s}_{1,3}$`(Y, Z)` : $Y \leftrightarrow Z$

Now consider the deletion of rule $\mathtt{r}_5$. The initial state when we start the main loop has

$\texttt{p(X, Y)} : true \mapsto X \leftrightarrow Y$
$\texttt{s(Y, Z)} : Y \leftrightarrow Z \mapsto Y \wedge Z$

in the answer table and the dependency arc table is empty. The priority queue entry is $newcall(\texttt{q} : true)$. We start a new answer entry for $\texttt{q} : true \mapsto \perp$ and add event $arc(A)$. This is selected, arc A is re-added to the dependency arc table, and $arc(B)$ is placed on the priority queue. This is selected and arc B is placed back in the dependency arc table and the event $newcall(\texttt{r(X, Y, Z)} : X \leftrightarrow Y)$ is placed on the queue. This generates an answer entry $\texttt{r(X, Y, Z)} : X \leftrightarrow Y \mapsto \perp$ and the event arc $\texttt{r}_4\texttt{(X, Y, Z)} : X \leftrightarrow Y \Rightarrow [X \leftrightarrow Y]\ \texttt{X = Z} : true$ is added to the priority queue. This in turn generates new answer information $X \leftrightarrow Y \leftrightarrow Z$ and the event $updated(\texttt{r(X, Y, Z)} : X \leftrightarrow Y)$. This is replaced with $arc(B)$, which is replaced with $arc(C)$, which results in arc C being re-added to the dependency graph and new answer info $\texttt{q} : true \mapsto true$ and an event $updated(\texttt{q} : true)$ which is removed with no effect. The resulting state is identical to the starting state.

Example 2.4.4 Consider again the `rev` and `app` program from Example 2.3.1. After analysis of the entire program we are in **State 3**. Now consider the deletion of clause [C] from [A,B,C,D]. $T = up(app(X, Y, Z))$ is all the calling patterns so, all the answer table and dependency arc table are emptied. Reanalysis is complete starting from the initial call $\texttt{rev}(\texttt{X}, \texttt{Y}) : true$ and results in **State 1** as expected.

Correctness of the incremental top-down deletion algorithm follows from correctness of the generic algorithm. Execution of the top-down deletion algorithm is identical to that of the generic algorithm except that information about the answers to some call patterns which do not depend on the deleted rules is already in the data structures.

Theorem 2.4.5 If a program P is first analyzed and then rules R are deleted from the program and the remaining rules are reanalyzed with the top-down deletion algorithm, the same answer and dependency arc tables will be obtained as when the rules $P \setminus R$ are analyzed by the generic algorithm.

The cost of performing the actual analysis incrementally can be no worse than performing the analysis all at once. Let $C_{del-td}(\bar{F}, R, R', S)$ be the worst case number of calls to the parametric functions $\bar{F}$ when analyzing the program R

with rules R' deleted for all possible priority strategies with the top-down deletion algorithm after already analyzing the program R for call patterns S.

Theorem 2.4.6 Let R and R' be sets of rules such that $R' \subseteq R$. For any call patterns S and parametric functions $\bar{F}$,

$$C_{noninc}(\bar{F}, R \setminus R', S) \geq C_{del-td}(\bar{F}, R, R', S).$$

2.4.3 "Bottom-up" Deletion Algorithm

The last theorem shows that the top-down deletion algorithm is never worse than starting the analysis from scratch. However, in practice it is unlikely to be that much better, as on average deleting a single rule will mean that half of the dependency arcs and answers are deleted in the first phase of the algorithm. The reason is that the top-down algorithm is very pessimistic—deleting everything unless it is sure that it will be useful. For this reason we now consider a more optimistic algorithm. The algorithm assumes that calling patterns to changed predicate definitions are still likely to be useful. In the worst case it may spend a large amount of time reanalyzing calling patterns that end up being useless. But in the best case we do not need to reexamine large parts of the program above changes when no actual effect is felt. The algorithm proceeds by computing new answers for calls to the lowest strongly connected component[1] (SCC) in the program call graph which is affected by the rule deletion, and then moving upwards to higher SCCs. At each stage the algorithm recomputes or verifies the current answers to the calls to the SCC without considering dependency arcs from SCC in higher levels. This is possible because if the answer changes, the *arc* events they would generate are computed anyway. If the answers are unchanged then the algorithm stops, otherwise it examines the SCCs which depend on the changed answers (using the dependency arcs). For obvious reasons we call the algorithm Bottom-Up Deletion. It is shown in Figure 2.5.

Example 2.4.7 Consider the same deletion as in Example 2.4.3. Initially H is `r(X,Y,Z)` $: X \leftrightarrow Y$. Its answer table entry `r(X,Y,Z)` $: X \leftrightarrow Y \mapsto X \leftrightarrow Y \leftrightarrow Z$ is moved to the old_table. The dependency arc (B) is moved

[1]A *strongly connected component* (SCC) in a directed graph is a set of nodes S such that $\forall n_1, n_2 \in S$ there is a path from n_1 to n_2.

```
bottom_up_delete(S, D)
    H := ∅
    foreach rule A_k ← B_{k,1}, ..., B_{k,n_k} ∈ D
        foreach A : CP ↦ AP in table
            H := H ∪ (A : CP)
    while H is not empty
        let B : _ ∈ H be such that B is of minimum predicate SCC level
        T := calling patterns in dependency graph for predicates in
                same predicate SCC as B
        foreach A : CP ∈ T
            delete each arc A_k : CP ⇒ [CP_1]B_{k,j} : CP_2 from dependency_arc_table
        foreach A : CP ∈ external_calls(T, S)
            move entry A : CP ↦ AP from answer_table to old_table
            foreach arc B_k : CP_0 ⇒ [CP_1] B_{k,j} : CP_2 in dependency_arc_table
                where there exists renaming σ s.t. (A : CP) = (B_{k,j} : CP_2)σ
                move B_k : CP_0 ⇒ [CP_1] B_{k,j} : CP_2 to old_dependency_table
            add_event(newcall(A : CP))
        main_loop()
        foreach A : CP ∈ external_calls(T, S)
            foreach arc B_k : CP_0 ⇒ [CP_1] B_{k,j} : CP_2 in old_dependency_table
                where there exists renaming σ s.t. (A : CP) = (B_{k,j} : CP_2)σ
                if answer pattern for A : CP in old_table and answer_table agree
                    move B_k : CP_0 ⇒ [CP_1] B_{k,j} : CP_2 to dependency_arc_table
                else
                    H := H ∪ (B : CP_0)
        H := H − T
        delete old_table

external_calls(T, S)
    U := ∅
    foreach A : CP ∈ T
        where exists arc B_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2
        and B : CP_0 ∉ T
        and there exists renaming σ s.t. (A : CP) = (B_{k,i} : CP_2)σ
        %% this means there is an external call
        U = U ∪ (A : CP)
    return U ∪ (T ∩ S)
```

Figure 2.5: Bottom-up Incremental Deletion Algorithm

to old_dependency_table. The event *newcall*($\mathtt{r(X,Y,Z)}$: $X \leftrightarrow Y$) is placed on the queue. This (re)generates new answer information $\mathtt{r(X,Y,Z)}$: $X \leftrightarrow Y \mapsto X \leftrightarrow Y \leftrightarrow Z$ and an event *updated*($\mathtt{r(X,Y,Z)}$: $X \leftrightarrow Y$). As the dependency arc table has no arc that need be recomputed we stop. Because the answer for $\mathtt{r(X,Y,Z)} : X \leftrightarrow Y$ is unchanged the arc (B) is moved to the dependency arc table and the algorithm terminates, without recomputing $\mathtt{q} : \mathit{true}$.

Example 2.4.8 Consider the `rev` and `app` program. After analysis of the entire program we are in **State 3**. Now consider the deletion of clause [C] from [A,B,C,D]. H is initially $\{\mathtt{app(X,Y,Z)} : \mathit{true}\}$. So is T. We remove the arc (3). We move the answer pattern for $\mathtt{app(X,Y,Z)}$: true and the arc (2) to the old_dependency_table. The event *newcall*($\mathtt{app(X,Y,Z)}$: true) is placed in the queue. The analysis proceeds to compute answer $\mathtt{app(X,Y,Z)}$: $\mathit{true} \mapsto \bot$. Since this has changed $\mathtt{rev(X,Y)}$: true is added to H. $\mathtt{app(X,Y,Z)}$: true is removed from H. In the next iteration S equals H. The answer pattern for $\mathtt{rev(X,Y)}$: true is moved to old_table, and the arc (1) is removed. Reanalysis proceeds as before including building a new call to $\mathtt{app(X,Y,Z)}$: X. This gives the answer $\mathtt{rev(X,Y)} : \mathit{true} \mapsto X \wedge Y$. The resulting state is **State 1** as expected. Note that the reanalysis of $\mathtt{app(X,Y,Z)}$: true was unnecessary for computing the answers to the call to `rev` (this was avoided by the top-down deletion).

Proving correctness of the incremental bottom-up deletion algorithm requires an inductive proof on the SCCs. Correctness of the algorithm for each SCC follows from correctness of the generic algorithm.

Theorem 2.4.9 If a program P is first analyzed for calls S and then rules R are deleted from the program and the remaining rules are reanalyzed with the bottom-up deletion algorithm, the same answer and dependency arc tables will be obtained as when the rules $P \setminus R$ are analyzed by the generic algorithm for S.

Unfortunately, in the worst case, reanalysis with the bottom-up deletion algorithm may take longer than reanalyzing the program from scratch using the generic algorithm. This is because the bottom-up algorithm may do a lot of work recomputing the answer patterns to calls in the lower SCCs which are no longer made. In practice, however, if the changes are few and have local extent, the bottom-up algorithm will be faster than the top-down.

2.5 Arbitrary Change

Given the above algorithms for addition and deletion of clauses we can handle any possible change of a set of clauses by first deleting the original and then adding the revised version. This is inefficient since the revision may not involve very far reaching changes while the deletion and addition together do. Moreover we compute two fixpoints rather than one.

In fact, the bottom-up and top-down deletion algorithms of the previous subsections can handle arbitrary change with only minor modification. Care must be taken to ensure that we reset enough answer information to $\perp$ to guarantee correctness. In particular the call dependency graph may have been altered after the change, so we must recompute the SCCs.

Example 2.5.1 Consider the following program

$\texttt{q}_1\texttt{(X, Y) :- p}_{1,1}\texttt{(X, Y).}$
$\texttt{p}_2\texttt{(a, b).}$

The complete analysis information for the initial call $\texttt{q(X, Y)}: X$ is

answer table:	$\texttt{q(X, Y)} : X \mapsto X \wedge Y$
	$\texttt{p(X, Y)} : X \mapsto X \wedge Y$
dependency arc table:	$\texttt{q}_1\texttt{(X, Y)} : X \Rightarrow \texttt{p}_{1,1}\texttt{(X, Y)} : X$

Consider replacing the clause $\texttt{p}_2\texttt{(a, b)}$ by $\texttt{p}_3\texttt{(X, Y) :- U = a, q}_2\texttt{(U, X)}$. Clearly we must set the answer for $\texttt{p(X, Y)} : X$ to $\perp$ but in the analysis of the (modified) clause we determine a dependency arc $\texttt{p}_3\texttt{(X, Y)} : X \Rightarrow \texttt{q}_2\texttt{(X, Y)} : X$. Hence because $\texttt{q(X, Y)} : X$ is now mutually recursive with $\texttt{p(X, Y)} : X$ is must also have its answer information set to $\perp$.

2.6 Local Change

One common reason for incremental modification to a program is optimizing compilation. Changes from optimization are special in the sense that usually the answers to the modified clause do not change. This means that the changes caused by the modification are local in that they cannot affect SCCs above the

```
local_change(S, R)
    let R be of the form A_k ← D_{k,1}, ..., D_{k,n_k}
    T := ∅
    foreach A : CP ↦ AP in answer table
        T := T ∪ (A : CP)
    T := T plus all B : CP_0 in same SCCs of dependency graph
    delete each arc of the form A_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2 from graph
    foreach A : CP ∈ external_calls(T, S)
        CP_0 := Aextend(CP, vars(D_{k,1}, ..., D_{k,n_k}))
        CP_1 := Aproject(CP_0, vars(D_{k,1}))
        add_event(arc(A_k : CP ⇒ [CP_0] D_{k,1} : CP_1))
    main_loop()
```

Figure 2.6: Local Change Algorithm

change. Thus, changes to the analysis are essentially restricted to computing the new call patterns that these clauses generate. This allows us to obtain an algorithm for local change (related to bottom-up deletion) which is more efficient than arbitrary change.

The algorithm for local change is given in Figure 2.6. It takes as arguments the original calling patterns S and a modified rule R, which we assume has the same number as the rule it replaces.

Correctness of the local change algorithm essentially follows from correctness of the bottom-up deletion algorithm.

Let $A \leftarrow B$ and $A \leftarrow B'$ be two rules. They are *local variants* with respect to the calls S and program P if for each call pattern in S the program $P \cup \{A \leftarrow B\}$ has the same answer patterns as $P \cup \{A \leftarrow B'\}$.

Theorem 2.6.1 Let P be a program analyzed for the initial call patterns S. Let R be a rule in P which in the analysis is called with call patterns S' and let R' be a local variant of R with respect to S' and $P \setminus \{R\}$. If the program P is reanalyzed with the routine `local_change(`S`,`R'`)` the same answer and dependency arc tables will be obtained as when the rules $P \cup \{R'\} \setminus \{R\}$ are analyzed by the generic algorithm.

The cost of performing the actual analysis incrementally can be no worse than performing the analysis all at once. Let $C_{local}(\bar{F}, P, R, R', S)$ be the worst case number of calls to the parametric functions $\bar{F}$ when analyzing the program P with rule R changed to R' for all possible priority strategies with the local change algorithm after already analyzing the program P for call patterns S.

Theorem 2.6.2 Let P be a program analyzed for the initial call patterns S. Let R be a rule in P which in the analysis is called with call patterns S' and let R' be a local variant of R with respect to S' and $P \setminus \{R\}$. For any parametric functions $\bar{F}$,

$$C_{noninc}(\bar{F}, P \cup \{R'\} \setminus \{R\}, S) \geq C_{local}(\bar{F}, P, R, R', S).$$

2.7 Experimental Results

We have conducted a number of experiments using the PLAI generic abstract interpretation system in order to assess the practicality of the techniques proposed in the previous sections. As mentioned in Section 2.2 the original fixpoint used in PLAI uses the concrete strategy of always performing *newcall* events first, processing non-recursive rules before recursive rules, and finishing processing a rule before starting another. Prior to the invocation of the fixpoint algorithm a step is performed in which the set of predicates in the program is split into the SCCs based on the call graph of the program found using Tarjan's algorithm [Tar72]. This information is used among other things to determine which predicates and which clauses of a predicate are (can be) recursive, used as mentioned above.

PLAI also incorporates some additional optimizations such as dealing directly with non-normalized programs and filtering out non-eligible clauses using concrete unification (or constraint solving) when possible. Also, instead of explicitly storing the annotation and call-pattern in the dependency arcs, it is recomputed from the head of the rule. In one way, however, PLAI is somewhat simpler than the generic algorithm given in Section 2.2: in order to simplify the implementation, the original fixpoint algorithm does not keep track of dependencies at the level of literals, but rather, in a coarser way, at the level of clauses.

Since relatively detailed dependencies, as described in the previous sections, seem quite useful in incremental analysis, we first introduced support for such more detailed dependencies in PLAI's fixpoint, as well as the quite small amount of additional code required to handle incremental addition.

Bench.	**Av**	**Mv**	**Ps**	**Cl**	**S**	**M**	**Gs**
aiakl	4.58	9	7	12	57	0	9
ann	3.17	14	65	170	20	36	99
bid	2.20	7	19	50	31	0	39
boyer	2.36	7	26	133	3	23	49
browse	2.63	5	8	29	62	25	9
deriv	3.70	5	1	10	100	0	1
fib	2.00	6	1	3	100	0	2
grammar	2.13	6	6	15	0	0	7
hanoiapp	4.25	9	2	4	100	0	3
mmatrix	3.17	7	3	6	100	0	4
occur	3.12	6	4	8	75	0	5
peephole	3.15	7	26	134	7	46	39
progeom	3.59	9	9	18	66	0	13
qplan	3.18	16	46	148	32	28	56
qsortapp	3.29	7	3	7	100	0	13
query	0.19	6	4	52	0	0	4
read	4.20	13	24	54	12	33	75
rdtok	3.07	7	22	88	27	40	37
serialize	4.18	7	5	12	80	0	8
tak	7.00	10	1	2	100	0	3
warplan	2.47	7	29	101	31	17	64
witt	4.57	18	77	160	35	22	102
zebra	2.06	25	6	18	33	0	11

Table 2.1: Summary of benchmark statistics

A relatively wide range of programs has been used as benchmarks. Some statistics on their size and complexity is given in Table 2.1. **Av**, **Mv** are respectively the average and maximum number of variables in each clause analyzed (dead code is not considered); **Ps** and **Cl** are, respectively, the total number of predicates and clauses analyzed; **S**, and **M** are, respectively, the percentage of simply and mutually recursive predicates; **Gs** is the total number of different goals solved in analyzing the program, i.e., the total number of syntactically different calls.

Bench.	Strd	Incr	I_cl	NI_cl	I SD	NI SD	I SU
aiakl	3746	3859	4050	6153	1.05	1.59	1.52
ann	7882	7746	50830	643609	6.56	83.09	12.66
bid	916	962	4436	16937	4.61	17.61	3.82
boyer	3625	2808	20853	271043	7.43	96.53	13.00
browse	495	516	1703	9560	3.30	18.53	5.61
deriv	766	492	3199	1736	6.50	3.53	0.54
fib	46	52	53	89	1.02	1.71	1.68
grammar	155	175	496	1193	2.83	6.82	2.41
hanoiapp	613	629	1036	1419	1.65	2.26	1.37
mmatrix	306	329	733	1003	2.23	3.05	1.37
occur	342	335	396	523	1.18	1.56	1.32
peephole	7256	6605	65546	567572	9.92	85.93	8.66
progeom	240	256	406	1066	1.59	4.16	2.63
qplan	1973	2036	41912	154500	20.59	75.88	3.69
qsortapp	346	372	646	1169	1.74	3.14	1.81
query	176	185	2079	4626	11.24	25.01	2.23
rdtok	2032	2793	23606	39193	8.45	14.03	1.66
read	36416	47899	187512	1044112	3.91	21.80	5.57
serialize	569	733	1596	3556	2.18	4.85	2.23
tak	109	123	127	166	1.03	1.35	1.31
warplan	4682	3966	45592	122562	11.50	30.90	2.69
witt	2543	2526	21143	65109	8.37	25.78	3.08
zebra	4068	4146	9312	45340	2.25	10.94	4.87
				Average	5.44	33.53	6.16

Table 2.2: Incremental vs. Non-incremental Addition

All execution times presented in the tables are milliseconds on a Sparc 10. . In all the experiments, the description (abstract) domain uses is the *sharing+freeness* domain [MH91]. In Table 2.2 **Strd** is the time taken by PLAI's original fixpoint in order to analyze the whole program as one block. **Incr** is the equivalent figure for the slightly modified fixpoint including the more detailed dependencies. It turns out that the additional cost of keeping track of more detailed dependencies is essentially offset by some improvement in the convergence of the fixpoint algorithm. In any case, the differences are due to the difference in dependency tracking rather than the cost of incrementality. Thus, the fully incremental algorithm shows no real disadvantage when analyzing programs in one block.

In order to test the relative performance of incremental and non-incremental analysis in the context of addition, we timed the analysis of the same benchmarks but adding the clauses one by one. I.e., the analysis was first run for the first clause only. Then the next clause was added and the resulting program re-analyzed. This process was repeated until the last clause of the program. The total time involved in this process is given in Table 2.2 by **I_cl**, for the case of incremental analysis, and by **NI_cl** for the case of restarting the analysis from scratch every time a clause is added (as would be necessary with the original PLAI system). In the latter case, the same incremental implementation was actually used (but erasing the tables between analyses) in order to factor out any differences in fixpoint algorithms. **I SD** and **NI SD** represent, respectively, the *slowdown* due to clause by clause addition with respect to analyzing the whole program at once (i.e., with respect to **Incr**). **I SU** (NI_cl / I_cl) shows the speedup obtained by incremental analysis. The results are quite encouraging: in the worst case studied of compiling clause by clause the slowdown with respect to analyzing the entire file in one block is on the average of only a factor of 5.44. Doing the same thing with the non-incremental analysis implies an average slowdown of 33.53 (i.e., over six times worse than the incremental analysis).

In order to test the relative performance of incremental and non-incremental analysis in the context of deletion, we timed the analysis of the same benchmarks but deleting the clauses one by one. Starting from an already analyzed file, the last clause was deleted and the resulting program (re-)analyzed. This process was repeated until the file was empty. The total time involved in this process is given in Table 2.3 by **NI**, for the case of restarting the analysis from scratch every time a clause is deleted (this is equivalent to **NI_cl - Incr** in Table 2.2), by **I_td** for the case of incremental analysis using the "top-down" algorithm, and by **I_bu** for the "bottom-up" algorithm. **I_td SU** and **I_bu SU** represent the speedups obtained by the top-down and bottom up algorithms, respectively, with

Bench.	NI	I_td	I_bu	I_td SU	I_bu SU	td/bu
aiakl	2294	1676	1016	1.37	2.26	1.65
ann	635863	37590	14230	16.92	44.68	2.64
bid	15975	4463	1726	3.58	9.26	2.59
boyer	268235	54656	19903	4.91	13.48	2.75
browse	9044	1693	720	5.34	12.56	2.35
deriv	1244	1136	1033	1.10	1.20	1.10
fib	37	23	23	1.61	1.61	1.00
grammar	1018	399	253	2.55	4.02	1.58
hanoiapp	790	746	723	1.06	1.09	1.03
mmatrix	674	626	180	1.08	3.74	3.48
occur	188	63	80	2.98	2.35	0.79
peephole	560967	125843	37472	4.46	14.97	3.36
progeom	810	110	86	7.36	9.42	1.28
qplan	152464	51079	3086	2.98	49.41	16.55
qsortapp	797	497	316	1.60	2.52	1.57
query	4441	2179	700	2.04	6.34	3.11
rdtok	36400	26242	8146	1.39	4.47	3.22
read	996213	716599	254420	1.39	3.92	2.82
serialize	2823	553	449	5.10	6.29	1.23
tak	43	46	40	0.93	1.07	1.15
warplan	118596	63690	5426	1.86	21.86	11.74
witt	62583	3413	2466	18.34	25.38	1.38
zebra	41194	13289	2110	3.10	19.52	6.30
			Average	2.63	8.21	3.12

Table 2.3: Incremental vs. Non-incremental Deletion

respect to non-incremental analysis (**NI**). The results are also very encouraging: in the worst case studied of compiling clause by clause the speedup with respect to the non-incremental algorithm is on the average a factor of 2.63 for top-down and 8.21 for bottom-up. The results seem to favour the bottom-up algorithm, as shown by **td/bu** in Table 2.3, giving an average speedup of 3.12 for bottom-up with respect to top-down.

Bench.	Cl	1^{st}	Inc	Scr	Scr\Inc	Scr$\backslash(1^{st} + Inc)$
aiakl	1	3859	642	4046	6.30	0.90
ann	11	7746	3526	10256	2.91	0.91
bid	6	962	533	1395	2.62	0.93
boyer	2	2808	909	4725	5.20	1.27
browse	4	516	545	1043	1.91	0.98
deriv	4	492	1018	1239	1.22	0.82
hanoiapp	1	629	319	778	2.44	0.82
mmatrix	2	329	416	669	1.61	0.90
occur	2	335	473	752	1.59	0.93
peephole	2	6605	1082	7546	6.97	0.98
progeom	1	256	52	276	5.31	0.90
qplan	2	2036	273	2312	8.47	1.00
query	2	185	109	242	2.22	0.82
read	1	47899	52	48618	934.96	1.01
serialize	1	733	58	803	13.84	1.02
warplan	8	3966	4086	22589	5.53	2.81
zebra	1	4146	3019	4729	1.57	0.66
				Average	6.55	1.11

Table 2.4: Local Change

Although we have implemented it, we do not report on the performance of arbitrary change because of the difficulty in modeling in a systematic way the types of changes that are likely to occur in the circumstances in which this type of change occurs (as, for example, during an interactive program development session). We have studied however the case of local reanalysis in a realistic environment: within the &-Prolog compiler, in which, after a first pass of analysis, new, specialized clauses are generated containing run-time tests, and a reanalysis is performed in order to propagate the more precise information which can be obtained in the program beyond the points where the new tests have been introduced. This more precise information is then used for multiple specialization (see Chapter 5). The results are shown in Table 2.4 in which benchmarks that do not generate run-time tests have been left out, since no specialization is performed for them and no reanalysis is needed in that case. Only program entry points are given to the analysis, i.e., no input patterns are specified for such entry points. This represents the likely situation where the user provides little information to the analyzer and also produces more run-time tests and thus more specializations and reanalysis, which allows us to study more benchmarks (note that if very precise information is given by the user then many benchmarks are parallelized without any run-time tests and then no specialization—and thus no reanalysis—occurs). **Cl** is the number of clauses that have changed. $\mathbf{1}^{st}$ is the time for analysis of the program in the first pass. **Inc** is the time for additional analysis after annotation (using the incremental algorithm). **Scr** is the time for additional analysis after annotation but restarting the analysis from scratch, i.e., no incrementality. **Scr****Inc** is the speedup in the reanalysis part due to incrementality. **Scr**\\(1^{st}+**Inc)** is a measure of the incrementality (close to one or over one is desirable). The results of incremental analysis of local change are even more encouraging than the previous ones. The speedups are quite impressive and the incrementality level is high or very high in all cases. In fact, in `boyer`, and, specially, in `warplan` incrementality is indeed very high. This is related to the fact that there is a high degree of specialization in these programs and the first analysis is run over many less clauses than the second pass, which penalizes reanalyzing from scratch.

2.8 Chapter Conclusions and Future Work

We have presented a generic fixpoint algorithm which generalizes the analysis algorithms used in current analysis engines. Such generic analysis engine is used as a basis for describing the proposed algorithms for incremental analysis. We have classified the possible changes to a program into addition, deletion, local change, and arbitrary change, and proposed, for each one of these, algorithms for identifying the parts of the analysis that must be recomputed and for performing the actual recomputation. We have also discussed the potential benefits and drawbacks of these algorithms. Finally, we have presented some experimental results obtained with an implementation of the algorithms in the PLAI generic abstract interpretation framework. The results show significant benefits when using the proposed incremental analysis algorithms.

We argue that our work contributes to open the door to practical, every day use of global analysis in the compilation of logic programs, even in the interactive environment which is often preferred by the users of such systems. We also argue that our results may shed new light into new possibilities for modular analysis. Furthermore, while current analyzers can deal correctly with dynamic program modification primitives, this implies having to give up on many optimizations not only for the dynamic predicates themselves but also for any code called from such predicates. The ability to update global information incrementally (and thus with reduced overhead) down to the level of single clause additions and deletions makes it possible to deal with these primitives in a much more accurate way.

Finally, it should be pointed out that in order to have a fully incremental compiler not only the analysis phase, but also the optimization phases of the compiler which use the information obtained from global analysis must be made incremental. We have reported on our results for a certain kind of such incremental optimization in the context of local change due to specialization in the &-Prolog parallelizer. Given the good results obtained in our experiments we plan on making the complete optimization process of both the &-Prolog and CLP($\mathcal{R}$) compilers fully incremental.

Chapter 3

Optimized Algorithms for Incremental Analysis

Global analysis of logic programs can be performed effectively by the use of one of several existing efficient algorithms. However, the traditional global analysis scheme in which all the program code is known in advance and no previous analysis information is available is unsatisfactory in many situations. As seen in Chapter 2, incremental analysis of logic programs is feasible and much more efficient in certain contexts than traditional (non-incremental) global analysis. However, incremental analysis poses additional requirements on the fixpoint algorithm used. In this chapter we identify these requirements, identify an important class of strategies meeting the requirements, present sufficient a priori conditions for such strategies, and propose, implement, and evaluate experimentally a novel algorithm for incremental analysis based on these ideas. The experimental results show that the proposed algorithm performs very efficiently in the incremental case while being comparable to (and, in some cases, considerably better than) other state-of-the-art analysis algorithms even for the non-incremental case. We argue that our discussions, results, and experiments also shed light on some of the many tradeoffs involved in the design of algorithms for logic program analysis.

3.1 Introduction

As seen in Chapterchap:Incremental-Analysis, incremental analysis of logic programs has been shown to be feasible and much more efficient in certain contexts than traditional (non-incremental) global analysis (see [HPMS95, KB95] for a different approach). In particular, in Chapter 2 we have discussed the different types of changes that have to be dealt with in an incremental setting, provided overall solutions for dealing with such changes (in terms of which parts of the analysis graph need to be updated and recomputed), and proposed a basic set of techniques that showed the feasibility of the approach. It was also observed that incremental analysis poses additional requirements on the fixpoint algorithm used since some assumptions on the program that traditional algorithms make are no longer valid. In this chapter, we directly address this issue by identifying more concretely such requirements and proposing optimizations in order to improve the performance of the fixpoint algorithm while meeting the requirements. We also aim to define, implement, and evaluate experimentally a novel algorithm for incremental analysis and compare it to some previously proposed algorithms for incremental as well as non-incremental analysis.

To the best of our knowledge, this is the first work dealing with the design and experimentation of fixpoint algorithms specially tailored for incremental analysis of logic programs. Additionally, our results imply performance improvements even in a non-incremental setting. Thus, we believe our discussions, results, and experiments may also clarify some of the many tradeoffs involved in the design of algorithms for logic program analysis in general.

3.2 Incremental Analysis Requirements

As mentioned before, the aim of the kind of (goal oriented) program analysis performed by the analysis engines mentioned in the previous section is, for a particular description domain, to take a program and a set of initial *calling patterns* (descriptions of the possible calling modes into the program) 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 calling patterns.

The aim of *incremental* global analysis is, given a program, its least analysis graph, and a series of changes to the program, to obtain the new least analysis graph as efficiently as possible. A simple but inefficient way of computing the new least analysis graph is to simply discard the previous analysis graph and start analysis from scratch on the new program. However, much of the information in the previous analysis graph may still be valid, and incremental analysis should be able to reuse such information, instead of recomputing it from scratch.

Unfortunately, traditional fixpoint algorithms for abstract interpretation of logic programs cannot be used directly (at least in general) in the context of incremental analysis for reasons of accuracy, efficiency, and even correctness. This is because such algorithms assume that once a local fixpoint has been reached for a calling pattern, i.e., an *answer pattern* for this calling pattern has been computed, this information will not change and can be used safely thereafter. This assumption is no longer valid in the incremental case, since an answer pattern may become inaccurate if some clauses are eliminated from the program (*incremental deletion*) or even incorrect if more clauses are added to the program (*incremental addition*). When performing *arbitrary change* on the program (i.e., when both additions and deletions are performed), the old answer pattern can be incorrect, inaccurate, or both.

We now discuss two requirements that incremental analysis poses on the fixpoint algorithm.

Detailed Dependency Information: Most practical fixpoint algorithms try to make iterations as local as possible by using some kind of dependency information. Thanks to this information it is possible to revisit only a reduced set of nodes of the graph when an answer pattern changes during analysis. Additionally, dependency information can also be used to detect earlier that a fixpoint has been reached. The more accurate such dependency information is, the more localized (and, thus, less costly) the fixpoint iterations can be.

In the context of incremental analysis, in addition to localizing the fixpoint and detecting termination earlier, dependencies are useful for a third reason: they help locate the parts of the analysis graph that may be affected by program changes and which thus need to be recomputed as required by such changes. Obviously, if more detailed dependency information is kept track of, a smaller part of the analysis graph will have to be recomputed after modifying the program.

Propagation of Incrementally Updated Answer Patterns: Incremental deletion, local change, and arbitrary change (Chapter 2) do not pose extra requirements on the analysis algorithm, provided that detailed dependency information is available, since such changes only require the analysis algorithms to deal with new calling patterns. However, in incremental addition, i.e., when new clauses are added to a program already analyzed, the new clauses may also generate unexpected changes to previously computed answer patterns, i.e., they may update any answer pattern in the analysis graph. Once a global fixpoint has been reached, there is usually no way to propagate this updated information to the places in the analysis graph that may be affected using traditional analysis algorithms. If an algorithm is to deal efficiently with incremental addition it needs to be able to deal incrementally with the update (due to the additional clauses) of any answer pattern.

3.3 Optimizing the Generic Algorithm

The generic algorithm presented in Section 2.2 is parametric with respect to the event handling strategy in the priority queue, in order to capture the behaviour of several possible algorithms. As seen in Chapter 2, correctness of the analysis does not depend on the order in which events are processed. However, efficiency does.

The cost of analysis can be split into two components. The cost of computing the arc events, which for a given program P and a queuing strategy q will be denoted $\mathcal{C}_a(P,q)$, and the cost associated with dealing with the event queue which will be denoted $\mathcal{C}_q(P,q)$. There is clearly a trade-off between $\mathcal{C}_a(P,q)$ and $\mathcal{C}_q(P,q)$ in that a more sophisticated event handling strategy may result in a lower number of arcs traversed but at a higher event handling cost. We now discuss some possible optimizations to the generic fixpoint algorithm.

3.3.1 General Simplifications

Dealing Only with Arc Events in the Priority Queue: The generic fixpoint algorithm in Section 2.2 has to deal with three different kinds of events, namely *updated*, *arc* and *newcall*. This can make the priority mechanism for the queue rather complicated. Looking at the actions performed for each one of these events and the optimizations presented below, it can be seen that the effect of both updated and newcall can be reduced to that of the arc events. Additionally, newcall performs an initial guess of the answer pattern. However, we will always use $\perp$ as the trivial initial guess for newcall events. Therefore, the event queue only needs to deal with arc events. Whenever the generic analysis algorithm would add to the queue an updated or newcall event, the optimized algorithm will directly add to the queue the required arc events.

In what follows, the current event queue will be denoted as Q and will be a set of triples $\langle arc, q(arc), type\rangle$, where *type* can be either newcall or updated and indicates whether such arc was introduced due to a newcall or an updated answer pattern, and q will be a function called *queuing strategy* that will assign a priority (a natural number) to each arc event. $\top(Q)$ is a function that returns (and deletes) from a non-empty queue Q an element with highest priority.

Only One Priority per Rule and Calling Pattern: The generic algorithm makes intensive use of the event queue. Without loss of generality, we will assign priorities to arcs at a somewhat coarser level. Instead of assigning a (possibly) different priority to each arc event, we will always assign the same priority to all the arcs for the same rule and calling pattern.

Never Switching from an Arc to Another with the Same Priority: Once computation for an arc has finished (no other arc with a higher priority can be in the queue), the generic algorithm would add the rest of the arc (if any) to the queue and retrieve one of the arcs with highest priority. Instead, as there cannot be any other arc with higher priority, it is always safe to continue with the arc just added to the queue. Rather than adding the rest of the arc and retrieving it immediately, it is more efficient to process it directly. This optimization allows using the queue only once for each rule and calling pattern.

Indexing the Dependency Arc Table: Whenever a pattern is updated, all the arcs that used the old (incorrect) pattern must be found in order to generate arc events for them. This is done in procedure `add_dependent_rules` by checking all the entries in the dependency arc table against the pattern that has been updated. This process has linear complexity in the size of the analysis graph. The proposed optimization implies keeping a table that for each calling pattern contains the set of arcs that have used this information. In such a way, the set of arcs that depend on a given calling pattern can be found in constant time.

3.3.2 Restricting the Set of Queuing Strategies

Definition 3.3.1 [Dynamic Call Graph] The *dynamic call graph* of a program P, denoted as $D(P)$, is the graph obtained from the answer table and the dependency arc table generated for P by the generic analysis algorithm as follows: for each entry $A : CP \mapsto AP_0$ in the answer table create the node $A : CP$ and for each entry $H : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$ in the dependency arc table create an arc from node $H : CP_0$ to node $B : CP_B$, where $B : CP_B$ is the unique calling pattern for which there exists a renaming σ s.t. $B : CP_B = (B_{k,i} : CP_2)\sigma$.

Definition 3.3.2 [Reduced Call Graph] The *reduced call graph* of a program P, represented as $D_R(P)$ is the directed acyclic graph obtained by replacing each SCC in $D(P)$ by a single node in $D_R(P)$ labeled with the set of nodes in the SCC, and eliminating all arcs which are internal to the SCC.

Definition 3.3.3 [SCC-preserving] A queuing strategy q is *SCC-preserving* if $\forall$ program P $\forall \langle A_1, q(A_1), type1\rangle$, $\langle A_2, q(A_2), type2\rangle \in Q$, where $A_1 = arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2)$ and $A_2 = arc(H_{k'} : CP_{0'} \Rightarrow [CP_{1'}]\ B_{k',i'} : CP_2')$: if there is a path in $D_R(P)$ from $H_k : CP_0$ to $H_{k'} : CP_{0'}$ then $q(A_1) < q(A_2)$.

Theorem 3.3.4 $\forall$ queuing strategy q $\exists q'$ s.t. q' is SCC-preserving and $\forall$ program P $\mathcal{C}_a(P, q') \leq \mathcal{C}_a(P, q)$.

This theorem implies that if $\mathcal{C}_q(P, q')$ is low enough, the set of queuing strategies considered can be restricted to those which are SCC-preserving. Definition 3.3.3 (SCC-preserving) is not operational because $D_R(P)$ cannot be computed until analysis has finished. It is thus an “a posteriori” condition. Next, we give sufficient "a priori" conditions that ensure that a queuing strategy is SCC-preserving.

Definition 3.3.5 [Newcall Selecting] Let $Q \neq \emptyset$ be a queue with $\langle A, q(A), type\rangle = \top(Q)$ where $A = arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2)$. Let $A_1, \ldots, A_n$ be the set of arc events which the event $newcall(B_{k,i} : CP_2)$ will insert in the queue. A queuing strategy q is *newcall selecting* iff $\forall \langle A', q(A'), type'\rangle \in Q\ \forall i = 1\ldots, n : q(A_i) > q(A')$.

The intuition behind a newcall selecting strategy is that analysis processes calling patterns in a depth-first fashion. Note also that if no recursive predicate appears in the program, the least fixpoint would be obtained in one iteration. If the queuing strategy is not newcall selecting, several iterations may be needed even for non-recursive programs.

Definition 3.3.6 [Update Selecting] Let $Q \neq \emptyset$ be a queue with $\langle A, q(A), type\rangle = \top(Q)$. Suppose that after processing the last literal in A, an $updated(H : CP)$ event is generated. Let $A_1, \ldots, A_n$ be the set of arc events which the event $updated(H : CP)$ will insert in the queue and let $\langle A_k, q(A_k), newcall\rangle \in Q$ be such that $\forall \langle A', q(A'), newcall\rangle \in Q : q(A_k) \geq q(A')$. A queuing strategy q is *update selecting* iff $\forall \langle A', q(A'), type'\rangle \in Q\ \forall i = 1\ldots, n : (q(A_k) \geq q(A')) \rightarrow (q(A_i) > q(A'))$. I.e., the arc events generated by an updated event must have higher priority than any existing arc in the queue except for the arcs of updated type that were introduced after the last newcall.

When update selecting strategies are used together with delayed dependencies introduced below, the analysis algorithm locally iterates whenever an answer pattern may not be final rather than using this possibly incorrect information in parts of the analysis graph outside the SCC the answer pattern belongs to.

Delaying Entries in the Dependency Arc Table: This modification to the generic algorithm consists in executing "$AP_0 :=$ get_answer$(B_{k,i} : CP_2)$" before the conditional "**if** ($B_{k,i}$ is not a constraint) add (...) to dependency_arc_table" in the procedure `process_arc` in the generic algorithm. The aim is not no introduce any dependency until an answer pattern is actually used.

Notice than in this case we are not restricting the set of considered queuing strategies but rather we are modifying the generic algorithm itself.

Theorem 3.3.7 [Delaying Dependencies] If the queuing strategy is newcall and update selecting then the algorithm obtained from the generic one by delaying dependencies produces the same analysis results as the generic algorithm.

Theorem 3.3.8 If dependencies are delayed and the queuing strategy is newcall and update selecting then all the arc events generated by an $updated(A : CP)$ event belong to the same SCC as $A : CP$.

Suppose that when processing the event $arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2)$, the answer pattern for $B_{k,i} : CP_2$ is updated m times. In the worst case, the continuation of $arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2)$ would be computed m times. Additionally, this computation may generate an $updated(H_k : CP_0)$ event which may in turn generate update events for any calling pattern in the analysis graph. This theorem ensures that unless $B_{k,i} : CP_2$ and $H_k : CP_0$ are in the same SCC, the continuation of the arc will only be computed once due to updated values of $B_{k,i} : CP_2$, independently of the number of times the answer pattern for $B_{k,i} : CP_2$ is updated and the number of iterations needed to compute it.

Theorem 3.3.9 [SCC-preserving] If dependencies are delayed then if a queuing strategy q is newcall selecting and update selecting then q is SCC-preserving.

This is a sufficient "a priori" condition to obtain SCC-preserving strategies.

3.3.3 Parametric Strategies

Ordering Arcs from Newcall Events – the Newcall Strategy: Although SCC-preserving strategies are efficient in general, for any given program P different SCC-preserving strategies may have different values for $\mathcal{C}_a(P, q)$ and $\mathcal{C}_q(P, q)$. There are still several degrees of freedom associated with the event handling strategy. The first one, which we will call the *newcall strategy* refers to the priorities among the different arcs generated by a single newcall (there will be one arc event per clause defining the called predicate). We know that all of them should have a higher priority than the existing arcs, but nothing has been said up to now about their relative priorities.

```
analyze(S)
  foreach A : CP ∈ S
    new_calling_pattern(A : CP)

process_update(Updates)
  if Updates = A1 :: As
    UAs := process_arc(A1)
    NAs :=
    global_updating_strategy(As, UAs)
    process_update(NAs)

insert_answer_info(H : CP ↦ AP)
  AP0 := lookup_answer(H : CP)
  AP1 := Alub(AP, AP0)
  A := {}
  if (AP0 <> AP1)
    add (H : CP ↦ AP1) to answer_table
    foreach arc of the form
      Hk : CP0 ⇒ [CP1] Bk,i : CP2
      in dependency_arc_table
      where there exists renaming σ
      s.t. H : CP = (Bk,i : CP2)σ
      A := A ⋃
        {Hk : CP0 ⇒ [CP1] Bk,i : CP2}
  return:=local_updating_strategy(A)

lookup_answer(A : CP)
  if (there exists a renaming σ s.t.
    σ(A : CP) ↦ AP in answer_table)
    return σ^-1(AP)
  else
    return σ^-1(
      new_calling_pattern(σ(A : CP)))
    where σ is a renaming s.t.
    σ(A) is in base form

new_calling_pattern(A : CP)
  add A : CP ↦ ⊥ to answer_table
  A0 := {}
  foreach rule Ak ← Bk,1, ..., Bk,nk
    CP0 := Aextend(CP, vars(Bk,1, ..., Bk,nk))
    CP1 := Aproject(CP, Bk,1)
    A0 := A0 ⋃
    {Ak : CP ⇒ [CP] Bk,1 : CP1}
  Arcs := newcall_strategy(A0)
  process_newcall(Arcs)
  Let σ be a renaming s.t.
    σ(A : CP) ↦ AP in answer_table
    return σ^-1(AP)

process_newcall(NewCalls)
  if NewCalls = A1 :: As
    UArcs := process_arc(A1)
    process_update(UArcs)
    process_newcall(As)

process_arc(Hk : CP0 ⇒ [CP1] Bk,i : CP2)
  if (Bk,i is not a constraint)
    AP0 := lookup_answer(Bk,i : CP2)
    add Hk : CP0 ⇒ [CP1] Bk,i : CP2
    to dependency_arc_table
  else
    AP0 := Aadd(Bk,i, CP2)
  CP3 := Acombine(CP1, AP0)
  if (CP3 <> ⊥ and i <> nk)
    CP4 := Aproject(CP3, Bk,i+1)
    U := process_arc(
      Hk : CP0 ⇒ [CP3] Bk,i+1 : CP4)
  elseif (CP3 <> ⊥)
    AP1 := Aproject(CP3, Hk)
    U := insert_answer_info(
        H : CP0 ↦ AP1)
  return U
```

Figure 3.1: Optimized SCC-preserving analysis algorithm

Ordering Arcs from Updated Events – the Updating Strategy: The newcall selecting condition is in a sense stronger than the updating strategy condition. The newcall selecting condition requires the new arcs to be assigned priorities which are higher than any other existing one. Therefore, there is even more freedom to assign priorities to arcs generated by updated events. The approach taken will be to split the updating strategy into two components. One is the relative order of the arc events introduced by a single updated event (*local updating strategy*). The other one is the order of these new arc events with respect to the already existing updated type arc events in the queue that were introduced in the queue later than any newcall type arc event (*global updating strategy*).

3.4 An Optimized Analysis Algorithm

Figure 3.1 presents an optimized analysis algorithm in which dependencies are delayed. It also ensures that the newcall selecting and updating selecting conditions will hold, thus always providing SCC-preserving strategies (Theorem 3.3.9). It is parametric with respect to the newcall strategy and local and global updating strategies introduced above. Different choices of these strategies will provide different SCC-preserving instances of the algorithm with possibly different efficiency.

The two different types of arc events are treated separately by procedures `process_newcall` and `process_update`. Also, rather than having an external data structure for the queue, we will use explicit parameters to store the arc events that have to be processed. The run-time stack of procedure and function calls will isolate and store the arcs. Assuming that the pseudo-code used to describe the algorithm is sequential, the newcall selecting condition is satisfied because if no entry is stored for a calling pattern in the answer table, the procedure `look_up_answer` will have to wait for `new_calling_pattern` to finish before returning control to the calling `process_arc` procedure. The update selecting condition is also satisfied because in the procedure `process_newcall`, `process_update` is called after processing each arc and before executing the recursive call to `process_newcall` for the remaining arcs from the same newcall.

```
incremental_addition(R)
    A_0 = {}
    foreach rule A_k ← B_{k,1}, ..., B_{k,n_k} ∈ R
        foreach entry A : CP ↦ AP
            in the answer_table
            CP_1 := Aproject(CP, B_{k,1})
            A_0 = A_0 ⋃
                {A_k : CP ⇒ [CP] B_{k,1} : CP_1}
    A := inc_updating_strategy(A_0)
    process_inc_update(A)

process_inc_update(Updates)
    if Updates = A_1 :: As
        U := process_arc(A_1)
        NAs :=
            inc_updating_strategy(As, U)
        process_inc_update(NAs)
```

Figure 3.2: Optimized Incremental Addition Algorithm

3.4.1 Augmenting the Algorithm for Incremental Addition

In order to cope with incremental addition, i.e., a set of rules R is added to a program, analysis should process each rule in R with all the existing calling patterns in the answer table for the predicate the rule belongs to. This is done by procedure `incremental_addition` in Figure 3.2. Note that a specialized version of procedure `process_update` which is called `process_inc_update` is used to start incremental analysis of the new arcs. However, in this case delaying dependencies is not possible because before incrementally analyzing the new clauses, a fixpoint will have been reached and all dependencies will have been introduced. Therefore, for any node $A : CP$ which existed in the analysis graph before incremental analysis started the arc events generated by an event *updated*$(A : CP)$ will not necessarily belong to the same SCC as $A : CP$ and analysis may no longer be SCC-preserving. Thus, it makes sense to use a more involved updating strategy for this case than for the non-incremental one in order to avoid unneeded recomputations. This strategy will be called the *inc_updating_strategy*. Incremental addition will be SCC-preserving or not depending on this strategy. However, for any new calling pattern in the analysis graph it is possible to delay dependencies and thus the algorithm in Figure 3.1 will be SCC-preserving for them. Thus, for such calling patterns it is profitable to use `process_update` whenever possible rather than `process_inc_update`. This is automatically achieved as the call to `process_arc` in procedure `process_inc_update` will always call `process_update` for any new calling pattern.

3.5 Experimental Results

A series of experiments has been performed for both the incremental and non-incremental case. The fixpoint algorithms we experiment with have been implemented as extensions to the PLAI generic abstract interpretation system. We argue that this makes comparisons between the new fixpoint algorithms and that of PLAI meaningful, since on the one hand PLAI is an efficient, highly optimized, state-of-the-art analysis system, and on the other hand the algorithms have been implemented using the same technology, with many data structures in common. They also share the domain dependent functions which, as in Chapter 2, are those of the *sharing+freeness* domain [MH91] in all the experiments.

Three analysis algorithms, as well as PLAI[1] have been considered. **DD** is the algorithm for incremental analysis used in Chapter 2 (**Incr** or **I** in the experimental results). Both **DI** and $\mathbf{DI}_S$ are instances of the algorithm presented in Figure 3.1, with the extensions for incremental addition presented in Figure 3.2. The difference between **DI** and $\mathbf{DI}_S$ is the newcall strategy used. $\mathbf{DI}_S$ uses the more elaborated strategy of computing the SCC of the static graph in order to give higher priority to non-recursive clauses. **DI** simply uses the lexical order of clauses to assign them different priorities. Both use the same updating strategy: the local strategy is to process arcs in the order they were introduced in the dependency arc table, and the global strategy is to use a LIFO stack and eliminate subsumed arcs, i.e., other arcs in the queue exist which ensure that their computation is redundant. The incremental updating strategy is to use a FIFO queue and eliminate subsumed arcs. **DD** uses *depth-dependent* and both **DI** and $\mathbf{DI}_S$ *depth-independent* propagations.

3.5.1 Analysis Times for the Non-Incremental Case

Table 3.1 shows the analysis times for a series of benchmark programs using the algorithms mentioned above. Times are in milliseconds on a Sparc 10 (SICStus 2.1, fastcode). The same set of benchmark programs as in Chapter 2, where they are briefly described. They can be obtained from `http://www.clip.dia.fi.upm.es`.

[1] The algorithm used for PLAI is the one in the standard distribution which has been augmented to keep track of detailed dependencies that are later used in multiple specialization (see Chapter 5). This introduces a small overhead over the original algorithm.

Bench.	**Cl**	**PLAI**	**DD**	**DI$_S$**	**DI**	**DD_SU**	**DI$_S$_SU**	**DI_SU**
aiakl	12	3526	3532	2563	2483	1.00	1.38	1.42
ann	170	6572	6593	6615	6906	1.00	0.99	0.95
bid	50	783	779	769	789	1.01	1.02	0.99
boyer	133	2352	2346	2339	2475	1.00	1.01	0.95
browse	29	329	339	343	393	0.97	0.96	0.84
deriv	10	420	436	421	406	0.96	1.00	1.03
fib	3	29	36	29	33	0.81	1.00	0.88
grammar	15	132	128	129	119	1.03	1.02	1.11
hanoiapp	4	579	565	619	539	1.02	0.94	1.07
mmatrix	6	309	306	312	326	1.01	0.99	0.95
occur	8	296	299	316	273	0.99	0.94	1.08
peephole	134	5855	5919	4870	5090	0.99	1.20	1.15
progeom	18	199	199	199	219	1.00	1.00	0.91
qplan	148	1513	1499	1422	1383	1.01	1.06	1.09
qsortapp	7	346	332	323	402	1.04	1.07	0.86
query	52	108	116	109	89	0.93	0.99	1.21
rdtok	54	2528	2509	1316	1209	1.01	1.92	2.09
read	88	44362	44259	14123	11765	1.00	3.14	3.77
serialize	12	629	629	663	616	1.00	0.95	1.02
tak	2	98	99	102	103	0.99	0.96	0.95
warplan	101	3439	3352	2789	2803	1.03	1.23	1.23
witt	160	1902	1902	1762	1738	1.00	1.08	1.09
zebra	18	3376	3356	3362	3259	1.01	1.00	1.04
					Overall	1.00	1.75	1.84
						(1.00)	(1.13)	(1.12)

Table 3.1: Analysis Times for the Non-Incremental Case

However, the number of clauses is included in the table (column **Cl**) for reference. **DD_SU**, **DI$_S$_SU** and **DI_SU** are the speed-ups obtained in analysis time by each fixpoint algorithm with respect to PLAI. As already observed in Chapter 2 the performance of **DD** is almost identical to that of PLAI (it introduces no relevant overhead) but has the advantage of being able to deal with incremental addition. On the other hand, both **DI** and **DI$_S$** show significant advantage with respect to **DD** (and PLAI). **DI** is the most efficient of the three, but the margin

Bench.	**DD**	**DI**$_S$	**DI**	**SU**$_{DD}$	**SU**$_{DI_S}$	**SU**$_{DI}$	**SD**$_{DD}$	**SD**$_{DI_S}$	**SD**$_{DI}$
aiakl	3860	3527	3237	1.52	1.38	1.29	1.09	1.38	1.30
ann	41680	25686	8120	12.66	22.82	72.83	6.32	3.88	1.18
bid	4220	2240	1433	3.82	6.54	9.80	5.42	2.91	1.82
boyer	20029	9039	3870	13.00	29.21	69.35	8.54	3.86	1.56
browse	1110	652	556	5.61	3.91	4.82	3.27	1.90	1.41
deriv	3083	1570	1126	0.54	1.63	2.07	7.07	3.73	2.77
fib	57	49	49	1.68	1.96	1.84	1.58	1.69	1.48
grammar	510	300	209	2.41	4.17	5.34	3.98	2.33	1.76
hanoiapp	990	779	816	1.37	1.86	1.46	1.75	1.26	1.51
mmatrix	709	360	343	1.37	2.67	3.12	2.32	1.15	1.05
occur	456	396	322	1.32	3.73	3.97	1.53	1.25	1.18
peephole	59899	15333	8533	8.66	28.19	52.05	10.12	3.15	1.68
progeom	389	360	283	2.63	2.87	3.44	1.95	1.81	1.29
qplan	39890	11303	2342	3.69	12.42	56.94	26.61	7.95	1.69
qsortapp	623	506	466	1.81	2.17	2.73	1.88	1.57	1.16
query	2296	919	277	2.23	7.14	20.32	19.79	8.43	3.11
rdtok	24176	3822	2363	1.66	6.96	10.06	9.64	2.90	1.95
read	176779	35760	22160	5.57	8.16	11.28	3.99	2.53	1.88
serialize	1496	1290	973	2.23	2.63	3.25	2.38	1.95	1.58
tak	139	120	113	1.31	1.75	1.77	1.40	1.18	1.10
warplan	41999	9436	5479	2.69	10.71	17.32	12.53	3.38	1.95
witt	19336	18606	2523	3.08	3.37	17.57	10.17	10.56	1.45
zebra	8580	2716	2480	4.87	15.32	16.44	2.56	0.81	0.76
Overall	6.64	2.13	1	6.15	13.74	28.36	5.69	3.18	1.57

Table 3.2: Incremental Addition Times

over **DI**$_S$ is small. Two overall speed-ups appear in the table for each algorithm. The one in brackets represents the overall speed-up after eliminating the `read` benchmark, because of the atypical results. The relative advantage of **DI** and **DI**$_S$ is inverted in this case. The peculiarity in `read` stems from the fact that the dynamic call graph has many cycles with lengths that are as high as 13. However, even when taking `read` out **DI** and **DI**$_S$ are both still somewhat better that **DD** and PLAI.

3.5.2 Analysis Times for the Incremental Case

Among the different types of incremental change identified in Chapter 2 the one which is really relevant for experimentation is incremental addition. The performance of the fixpoint algorithms in the other types of changes will be directly related to the efficiency of the algorithms in the non-incremental case, as no incremental update propagation is needed. Table 3.2 shows the analysis times for the same benchmarks but, as in the experiment in Chapter 2, adding the clauses one by one. I.e., the analysis was first run for the first clause only. Then the next clause was added and the resulting program (re-)analyzed. This process was repeated until the last clause of the program. The total time involved in this process is given by **DD**, $\mathbf{DI}_S$, and **DI**. Columns $\mathbf{SU}_{DD}$, $\mathbf{SU}_{DI_S}$, and $\mathbf{SU}_{DI}$ contain the speed-up obtained with respect to analyzing with the same algorithm the program clause by clause but erasing the analysis graph between analyses. Thus, it is a measure of the incrementality of each algorithm. An important speed-up is observed in $\mathbf{SU}_{DD}$ (as already noted in Chapter 2), but the incrementality of $\mathbf{DI}_S$ is twice as high, and that for **DI** in turn twice as high as that of $\mathbf{DI}_S$.

The last three columns in the table contain the slow-downs for clause by clause incremental analysis with respect to the time taken by the same algorithm when analyzing the file all at once. If we use the **DD** algorithm in an incremental way, the overhead resulting from analyzing clause by clause is greatly reduced with respect to the non-incremental case. However, the time required if we use **DI** incrementally is only about 3/2 of the time required to analyze the program all at once. There is even one case (the `zebra` benchmark) in which using the **DI** algorithm clause by clause is somewhat faster than analyzing the program all at once. However, we believe this is related to working set size and cache memory effects, as the number of arc events processed in both cases (presented in Table 3.3) is almost the same. In the **Overall** row we give the average analysis times for each algorithm, taking as unit the time for analysis clause by clause using the **DI** algorithm. At least for the benchmark programs **DI** is more than twice as fast as $\mathbf{DI}_S$ and more than 6 times faster than **DD**

Bench.	**N**	**U**	**T**	**U/T**	$\mathbf{N}_I$	$\mathbf{U}_I$	$\mathbf{UI}_I$	$\mathbf{T}_I$	$\mathbf{UI}_I/\mathbf{T}_I$
aiakl	50	19	69	0.28	52	8	76	136	0.56
ann	570	179	749	0.24	496	203	101	800	0.13
bid	191	14	205	0.07	144	10	165	319	0.52
boyer	248	70	318	0.22	82	34	330	446	0.74
browse	41	19	60	0.32	21	3	78	102	0.76
deriv	24	1	25	0.04	0	0	52	52	1.00
fib	14	3	17	0.18	6	3	8	17	0.47
grammar	24	0	24	0	2	0	28	30	0.93
hanoiapp	21	15	36	0.42	18	11	26	55	0.47
mmatrix	10	9	19	0.47	2	3	14	19	0.74
occur	15	14	29	0.48	12	12	4	28	0.14
peephole	255	170	425	0.40	180	23	440	643	0.68
progeom	41	9	50	0.18	38	9	3	50	0.06
qplan	384	41	425	0.10	205	31	235	471	0.50
qsortapp	44	15	59	0.25	23	4	41	68	0.60
query	59	0	59	0	0	0	62	62	1.00
rdtok	332	33	365	0.09	145	24	328	497	0.66
read	840	155	995	0.16	720	22	1398	2140	0.65
serialize	43	15	58	0.26	16	1	102	119	0.86
tak	27	5	32	0.16	17	5	10	32	0.31
warplan	330	38	368	0.10	169	13	362	544	0.67
witt	389	39	428	0.09	352	36	44	432	0.10
zebra	51	2	53	0.04	28	2	24	54	0.44
Overall	4003	865	4868	0.18	2728	457	3931	7116	0.45

Table 3.3: Number of *arc* Events Processed

3.5.3 Measuring $\mathcal{C}_a(P, q)$: Number of Arc Events

Table 3.3 shows the number of arc events needed to analyze each benchmark program in both the non-incremental and incremental case using the **DI** algorithm. This is equivalent to counting the number of times the function `process_arc` in the algorithm in Figure 3.1 is called (including any recursive calls) from (**N**) `process_newcall`, (**U**) `process_update`, and (**UI**) `process_inc_update`. **T** is the total number of arc events processed. $_I$ is used for the incremental case. The last row in the table shows the number of arc events of each type needed to analyze all the benchmarks. The remaining two columns (**U/T** and $\mathbf{UI}_I/\mathbf{T}_I$) give respectively the ratio of the total arc events that were due to update events in the non-incremental case and those due to the newly introduced clauses in the incremental case. **U/T** gives an idea of how much analysis effort is due to fixpoint computation for recursive calls. These figures show that using a good analysis algorithm, less than 20% of the effort is due to iterations. $\mathbf{UI}_I/\mathbf{T}_I$ gives the ratio of the computation performed by `process_inc_update` (which may use a more complex updating strategy). The ratio between the total number of arcs computed in the incremental and non-incremental case explains the slow-down associated to the analysis clause by clause. It is $7116 \div 4868 = 1.46$ in number of arc events processed and 1.57 in analysis times for the **DI** algorithm. The table also seems to imply that, for the strategies used, counting arc events is a good (and architecture independent) indicator of analysis time.

3.6 Chapter Conclusions

We have identified certain requirements that incremental analysis poses on the fixpoint algorithm used in global analysis of logic programs. We have proposed the class of SCC-preserving strategies, and shown that they meet incremental analysis requirements and at the same time offer low event handling queue cost and a small number of arc handling events. We have also presented sufficient a priori conditions for characterizing such SCC-preserving strategies. We have proposed a novel analysis algorithm embodying SCC-preserving strategies, and implemented and evaluated experimentally two instantiations of this algorithm incorporating newcall strategies of different complexity. The experimental results show that our proposed algorithm improves significantly on previously proposed algorithms for incremental analysis both in overall time and in incrementality, the instance using the simplest newcall strategy showing the most advantage. Full incremental addition, i.e., analyzing and reanalyzing while adding a file clause by clause, is less than than 60% slower than analyzing the program as a whole. In non-incremental analysis the differences between the two newcall strategies appear minimal, while the simple newcall strategy is much more profitable for the incremental case. In addition, even in the non-incremental case our incremental algorithm also improves significantly over previously proposed incremental algorithms and even also over the standard algorithms used in the PLAI system, which we believe is representative of current state of the art analyzers for logic programs.

Chapter 4

Analysis of Full Languages

Abstract interpretation-based data-flow analysis of logic programs is, at this point, relatively well understood from the point of view of general frameworks and abstract domains. On the other hand, comparatively little attention has been given to the problems which arise when analysis of a full, practical dialect of the Prolog language is attempted, and only few solutions to these problems have been proposed to date. Existing proposals generally restrict in one way or another the classes of programs which can be analyzed. This work attempts to fill this gap by considering a full dialect of Prolog, essentially the recent ISO standard, pointing out the problems that may arise in the analysis of such a dialect, and proposing a combination of known and novel solutions that together allow the correct analysis of arbitrary programs which use the full power of the language.

4.1 Introduction

As mentioned before, global program analysis, generally based on abstract interpretation [CC77], is becoming a practical tool in logic program compilation. However, most proposals to date have concentrated on general frameworks and suitable abstract domains. On the other hand, comparatively little attention has been given to the problems which arise when analysis of a full, practical language is attempted. Such problems relate to dealing correctly with all builtins, including meta-logical, extra-logical, and dynamic predicates (where the program is modified during execution). Often, problems also arise because not all the program code is accessible to the analysis, as is the case for some builtins (meta-calls), some predicates (multifile and/or dynamic), and some programs (multifile or modular).

Implementors of the analyses obviously have to somehow deal with such problems, and some of the implemented analyses provide solutions for some problems. However, the few solutions which have been published to date [VD92, Deb89b, HWD92, MH92, CRV94] generally restrict the use of builtin predicates in one way or another (and thus the class of programs which can be analyzed).

The work presented in this chapter attempts to fill this gap. We consider the correct analysis of a *full* dialect of Prolog. For concreteness, we essentially follow the recently accepted ISO standard [PRO94, DEDC96]. Our purpose is to review the features of the language which pose problems to global analysis and propose alternative solutions for dealing with these features. The most important objective is obviously to achieve correctness, but also as much accuracy as possible. Since arguably the main problem in static analysis is having dynamic code, which is not available at compile-time, we first propose a general solution for solving the problems associated with features such as dynamic predicates and meta-predicates, and consider other alternative solutions. The proposed alternatives are a combination of known solutions when they are useful, and novel solutions when the known ones are found lacking. The former are identified by giving references.

One of the motivations of our approach is that we would like to accommodate at the same time two types of users. First, the naive user, which would like analysis to be as transparent as possible. Second, we would also like to cater for the advanced user, which may like to guide the analysis in difficult places in order to obtain better optimizations. Thus, for each feature, we will propose solutions that require no user input, but we will also propose solutions that allow the user to provide input to the analysis process. This requires a clear interface to the analyzer at the program text level. Clearly, this need also arises when expressing the information gathered by the different analyses supported. We solve this by proposing an interface, in the form of *assertions*, which is useful not only for two-way communication between the user and the compiler, but also for the cooperation among different analysis tools and for connecting analyses with other modules of the compiler. Assertions are syntactic constructions which allow expressing properties of programs. The assertions proposed in this chapter are essentially a subset of the assertion language presented in Chapter 8. Thus, the assertions in this chapter are expressed in such language.[1]

After necessary preliminaries in Section 4.2, we propose several novel general solutions to deal with the analysis of dynamic programs in Section 4.3. A set of program assertions which can help in this task is then proposed in Section 4.4. We then revise our and previous solutions to deal with each of the language features in Section 4.5, except for modules and multifile programs, which are discussed in Section 4.6. There we propose a solution based on incremental analysis, and another one based on our program assertions. We conclude with Section 4.7.

We argue that the proposed set of solutions is the first one to allow the correct analysis of arbitrary programs which use the full power of the language without input from the user (while at the same time allowing such input if so desired).

4.2 Preliminaries and Notation

For simplicity we will assume that the abstract interpretation based analysis is constructed using the "Galois insertion" approach [CC77], in which an abstract domain is used which has a lattice structure, with a partial order denoted by $\sqsubseteq$, and whose top value we will refer to by $\top$, and its bottom value by $\perp$. We will refer to the least upper bound (lub) and greatest lower bound (glb) operators in the lattice by $\sqcup$ and $\sqcap$, respectively. The abstract computation proceeds using abstract counterparts of the concrete operations, the most relevant ones being unification (mgu^{α}) and composition ($\circ^{\alpha}$), which operate over abstract substitutions (α). Abstract unification is however often also expressed as a function $unify^{\alpha}$ which computes the abstract mgu of two concrete terms in the presence of a given abstract substitution.

Usually, a *collecting* semantics is used which attaches one or more (abstract) substitutions to program points (such as, for example, the point just before or just after the call of a given literal — the call and success substitutions for that literal). A goal dependent analysis associates abstract success substitutions to specific goals, in particular to call patterns, i.e. pairs of a goal and an abstract call substitution which expresses how the goal is called. Depending on the granularity

[1]It only slightly differs from the original notation in [BCHP96].

of the analysis, one or more success substitutions can be computed for different call patterns at the same program point. Goal independent analyses compute abstract success substitutions for generic goals, regardless of the call substitution.

In general we will concentrate on top-down analyses, since they are at present the ones most frequently used in optimizing compilers. However, we believe the techniques proposed are equally applicable to bottom-up analyses. In the text, we consider in general goal dependent analyses, but point out solutions for goal independent analyses where appropriate (see, e.g., [GDL92, GGL94, CGBH94]).

The pairs of call and success patterns computed by the analysis, be it top-down or bottom-up, goal dependent or independent, will be denoted by $AOT^{\alpha}(P)$ for a given program P. A *most general goal pattern* (or simply "goal pattern," hereafter) of a predicate is a *normalized* goal for that predicate, i.e. a goal whose predicate symbol and arity are those of the predicate and where all arguments are distinct variables. In goal dependent analyses, for every call pattern of the form $(goal\_pattern, call\_substitution)$ of a program P there are one or more associated success substitutions which will be denoted hereafter by $AOT^{\alpha}(P, call\_pattern)$. The same holds for goal independent analysis, where the call pattern is simply reduced to the goal pattern. By *program* we refer to the entire program text that the compiler has access to, including any directives and assertions.

4.3 Static Analysis of Dynamic Program Text

A main problem in statically analyzing logic programs is that not all of the code that would actually be run is statically accessible to the analysis. This can occur either because the particular calls occuring at some places are dynamically constructed, or because the code defining some predicates is dynamically modified. The following problems appear:

1. How to compute success substitutions for the calls which are not known; we call this the *success substitution problem*, and

2. How to determine calls and call substitutions which may appear from the code which is not known; we call this the *extra call pattern problem.*

Consider the following program, to be analyzed with entry point `goal`. The predicate `p/2` is known to be dynamic, and may thus be modified at run-time.

```
goal:- ..., X=a, ..., p(X,Y), ...

:- dynamic p/2.
p(X,Y):- q(X,Y).

q(X,Y).
l(a,b).
```

Assume that the call pattern of the goal `p(X,Y)` in the analysis indicates that `X` is ground and `Y` free. If we do not consider the possibility of run-time modifications of the code, the success pattern for `p(X,Y)` is the same as the call pattern. Also, since no calls exist to `l/2`, its definition is dead code. Assume now that a clause "`p(X,Y):- l(X,Y).`" is asserted at run-time. The previous analysis information is not correct for two reasons. First, the success pattern of `p(X,Y)` should now indicate that `Y` may be ground (success substitution problem). Second, a call for `l/2` now occurs which has not been considered in the previous analysis (extra call pattern problem).

The first problem is easier to solve: using appropriate topmost substitutions. We call an abstract substitution α *topmost* w.r.t. a tuple (set) of variables $\vec{x}$ iff $vars(\alpha) = \vec{x}$ and for all other substitution α' such that $vars(\alpha') = \vec{x}$, $\alpha' \sqsubseteq \alpha$. An abstract substitution α referring to variables $\vec{x}$ is said to be *topmost of* another substitution α', referring to the same variables, iff $\alpha \equiv \alpha' \circ^{\alpha} \alpha''$, where α'' is the topmost substitution w.r.t. $\vec{x}$. Therefore, for a given call substitution, the topmost abstract substitution w.r.t. it is the most accurate approximation which solves the success substitution problem. This is in contrast to roughly considering $\top$ or just giving up in the analysis. Topmost substitutions are preferred, since they are usually more accurate for some domains. For example, if a variable is known to be ground in the call substitution, it will continue being ground in the success substitution.

Note that this is in fact enough for goal independent analyses, for which the second problem does not apply. However, for goal dependent analyses the second problem needs to be solved in some way. This problem is caused by the impossibility of statically computing the subtree underlying a given call, either because this call is not known (it is statically undetermined), or because not all of the code defining the predicate for that call is available. Therefore, since from these subtrees new calls (and new call patterns) can appear, which affect other parts of the program, the whole analysis may not be correct.

There is a first straightforward solution to the extra call pattern problem. It can be tackled by simply assuming that there are unknown call patterns, and thus any of the predicates in the program may be called (either from the undetermined call or from within its subtree). This means that analysis may still proceed but topmost call patterns must be assumed for all predicates. This is similar to performing a goal independent analysis and it may allow some optimizations, but it will probably preclude others. However, if program multiple specialization (see Chapter 5) is done, a non-optimized version of the program should exist (since all the predicates in the program must be prepared to receive any input value), but other optimized versions could be inferred.

Consider the previous example. To solve the success substitution problem we can

(a) assume a topmost substitution w.r.t. `X` and `Y`, which will indicate that nothing is known of these two variables; or

(b) assume the topmost substitution w.r.t. the call substitution, which will indicate that nothing is known of `Y`, but still `X` is known to be ground.

To solve the extra call pattern problem we can

(a) assume new call patterns with topmost substitutions for all predicates in the program, since the asserted clause is not known during analysis; or

(b) perform the transformation proposed below, which will isolate the problem to predicate `l/2`, which is the only one affected.

We propose a second complete solution which is general enough and very elegant, with the only penalty of some cost in code size. The key idea is to compile essentially two versions of the program — one that is a straightforward compilation of the original program, and another that is analyzed assuming that the only possible calls to each predicate are those that appear explicitly in the program. This version will contain all the optimizations, which will be performed ignoring the effect of the undetermined calls. Still, in the other version, any optimizations possible with a goal independent analysis, or a topmost call pattern goal dependent analysis, may be introduced. Calling from undetermined calls into the more optimized version of the program (which will possibly be unprepared for the call patterns created by such calls) is avoided by making such calls call the less optimized version of the program. This will take place automatically because the terms that will be built at run-time will use the names of the original predicates. When a predicate in the original program is called, it will also call predicates in the original program. Therefore, the original predicate names are used for the less optimized version, and predicates in the more optimized version are renamed in an appropriate way (we will assume for simplicity that it is by using the prefix "`opt_`"). Thus, correctness of a transformation such as the following is guaranteed. Assume that `call(X)` is an undetermined call. If a clause such as the first one appears in the program, the second one is added:

```
p(...) :- q(...), call(X), r(...).
opt_p(...) :- opt_q(...), call(X), opt_r(...).
```

The top-level rewrites calls which have been declared as entry points to the program so that the optimized version is accessed. Note that this also solves (if needed) the general problem of answering queries that have not been declared as entry points: they simply access the less optimized version of the program. If the top-level does also check the call patterns, then it guarantees that only the entry patterns used in the analysis will be executed. For the declared entry patterns, execution will start in the optimized program and will move to the original program to compute a resolution subtree each time an undetermined call is executed. Upon return from the undetermined call, execution will go back to the optimized program.

We shall see how this solution can be applied both to the case of meta-predicates and to that of dynamic predicates, allowing full optimizations to be performed in general to "dynamic" programs. The impact of the optimizations performed in the renamed copy of the program will depend on the time that execution stays in each of the versions. Therefore, the relative computational load of undetermined calls w.r.t. the whole program will condition the benefits of the optimizations achieved. The only drawback with this solution is that it implies keeping two full copies of the program, although only in case there are undetermined calls. In cases where code space is a pressing issue, the user should be given the choice of turning this copying on and off.

4.4 Program Assertions

Assertions are statements regarding a program that are introduced as part of its code. Assertions refer to a given program point. We consider two general classes of program points: points inside a clause (such as, for example, before or after the execution of a given goal — the "goal level") and points that refer to a whole predicate (such as, for example, before entering or after exiting a predicate — the "predicate level"). At all levels assertions describe properties of the variables that appear in the program. We will call the descriptions of such properties *declarations*. There are at least two ways of representing declarations which we will call "property oriented" and "abstract domain oriented". In a property oriented assertion framework, there are declarations for each property a given variable or set of variables may have. Examples of such declarations are:

`mode(X,+)`	`X` is bound to a non-variable term
`term(X,r(Y))`	`X` is bound to term `r(Y)`
`depth(X,r/1)`	`X` is bound to a term `r(_)`

The property oriented approach presents two advantages. On one hand, it is easily extensible, provided one defines the semantics for the new properties one wants to add. On the other hand, it is also independent from any abstract domain for analysis. One only needs to define the semantics of each declaration, and, for each abstract domain, a translation into the corresponding abstract substitutions. For concreteness, and in order to avoid referring to any abstract domain in particular, we propose to use such a framework.

An alternative solution is to define declarations in an abstract domain oriented way. For example, for the sharing domain [JL88]:

`sharing([[X],[Y,Z]])` shows the sharing pattern among variables X, Y, Z

This is a simple enough solution but has the disadvantage that the meaning of such domains is often difficult for users to understand. Also, the interface is bound to change any time the domain changes. It has two other disadvantages. The semantics and the translation functions mentioned above have to be defined pairwise, i.e. one for each two different domains to be communicated. And, secondly, there can exist several (possibly overlapping) properties declared, one for each different domain. In the property oriented approach, additional properties that several domains might take advantage of are declared only once. In any case, both approaches are compatible via the *syntactic* scheme we propose.

4.4.1 Predicate Level: Entry Assertions

One class of predicate level assertions are `entry` assertions. They are specified using a directive style syntax, as follows:

`:- entry` *goal_pattern* `:` *declaration*`.`

These assertions state that calls to that predicate with the given abstract call substitution may exist at execution time. For example, the following assertion states that there can be a call to predicate `p/2` in which its two arguments are ground:

```
:- entry p(X,Y) : (ground(X),ground(Y)).
```

Entry assertions and goal dependent analysis. A crucial property of `entry` assertions, which makes them useful in goal dependent analyses, is that they must be *closed with respect to outside calls.* No call patterns other than those specified by the assertions in the program may occur from outside the program text. I.e., the list of `entry` assertions includes all calls that may occur to a program, apart from those which arise from the literals explicitly present in the program text. Obviously this is not an issue in goal independent analyses.

Entry assertions and multiple program specialization. If analysis is multivariant it is often convenient to create different versions of a predicate, i.e., to perform multiple specialization (see Chapter 5). This allows implementing different optimizations in each version. Each one of these versions generally receives an automatically generated unique name in the multiply specialized program. However, in order to keep the multiple specialization process transparent to the user, whenever more than one version is generated for a predicate which is a declared entry point of the program (and, thus, appears in an `entry` directive), the original name of the predicate is reserved for the version that will be called upon program query. If more than one `entry` assertion appears for a predicate and different versions are used for different assertions, it is obviously not possible to assign to all of them the original name of the predicate. There are two solutions to this. The first one is to add a front end with the exported name and run-time tests to determine the version to use. However, this implies run-time overhead. As an alternative we allow the `entry` directive to have one more argument, separated by ";" which indicates the name to be used for the version corresponding to this entry point. For example, given:

```
:- entry mmultiply(A,B,C) : ground([A,B]) ; mmultiply_ground.
:- entry mmultiply(A,B,C) : true ; mmultiply_any.
```

if these two entries originate different versions, they would be given different names. If two or more versions such as those above are collapsed into one, this one will get the name of any of the entry points and, in order to allow calls to all the names given in the assertions, binary clauses will be added to provide the other entry points to that same version.

4.4.2 Predicate Level: Trust Assertions

In addition to the more standard `entry` assertions we propose a different kind of assertions at the predicate level, which take the following form:

`:- trust pred` *goal_pattern* `:` *call_decl* `=>` *success_declaration*.

Declarations in `trust` assertions put in relation the call and the success patterns of calls to the given predicate. These assertions can be read as follows: if a literal that corresponds to *goal_pattern* is executed and *call_decl* holds for the associated call substitution, then *success_declaration* holds for the associated success substitution. Thus, these assertions relate abstract call and success substitutions. Note that *call_decl* can be empty (i.e., `true`). In this way, properties can be stated that must always hold for the success substitution, no matter what the call substitution is. This is useful also in goal independent analyses (and in this case it is equivalent to the "omode" declaration of [HWD92]).

Let $(p(\vec{x}), \alpha)$ denote the call pattern and α' the success substitution of a given `trust` assertion of a program P. The semantics of `trust` implies that $\forall \alpha_c\ (\alpha_c \sqsubseteq \alpha \Rightarrow AOT^{\alpha}(P, (p(\vec{x}), \alpha_c)) \sqsubseteq \alpha')$. I.e., for all call substitutions approximated by that of the given call pattern, their success substitutions are approximated by that of the assertion. For this reason, the compiler will "trust" them. This justifies their consideration of "extra" information, and thus and in contrast to `entry` assertions, the list of `trust` assertions of a program does *not* have to be closed w.r.t. all possible call patterns occurring in the program.

One of the main uses of `trust` assertions is in describing predicates that are not present in the program text. For example, the following assertions describe the behaviour of the predicate `p/2` for two possible call patterns:

```
:- trust pred p(X,Y) : (ground(X),free(Y)) => (ground(X),ground(Y)).
:- trust pred p(X,Y) : (free(X),ground(Y)) => (free(X),ground(Y)).
```

This would allow performing the analysis even if the code for `p/2` is not present. In that case the corresponding success information in the assertion can be used ("trusted") as success substitution.

In addition, `trust` assertions can be used to improve the analysis when the results of the analysis are imprecise. However, note that this does not save analyzing the predicate for the corresponding call pattern, since the abstract underlying subtree may contain call patterns that do not occur elsewhere in the program.

If we analyze a call pattern for which a `trust` assertion exists, two abstract success patterns will be available for it: that computed by the analysis (say α_s) and that given by the `trust` assertion (say α', for a call substitution α). As both must be correct, the intersection of them (which may be more accurate than any of them) must also be correct. The intersection among abstract substitutions (whose domain we have assumed has a lattice structure) is computed with the glb operator, $\sqcap$. Therefore, $AOT^{\alpha}(P, (p(\vec{x}), \alpha_c)) = \alpha_s \sqcap \alpha'$, provided that $\alpha_c \sqsubseteq \alpha$. Since $\forall \alpha_s \forall \alpha' \ (\alpha_s \sqcap \alpha' \sqsubseteq \alpha_s \wedge \alpha_s \sqcap \alpha' \sqsubseteq \alpha')$ correctness of the analysis within the `trust` semantics is guaranteed, i.e. $AOT^{\alpha}(P, (p(\vec{x}), \alpha_c)) \sqsubseteq \alpha'$ and $AOT^{\alpha}(P, (p(\vec{x}), \alpha_c)) \sqsubseteq \alpha_s$. However, if their informations are incompatible, their intersection is empty, and $\alpha_s \sqcap \alpha' = \bot$. This is an error (if $\alpha_s \neq \bot$ and also $\alpha' \neq \bot$), because the analysis information must be correct, and the same thing is assumed for the `trust` information. The analysis should give up and warn the user.

A similar scheme can be used to check the mutual consistency of assertions provided by the user. The result of the glb operation between inconsistent assertions will be $\bot$. Also, note that, in addition to improving the substitution at the given point, the trusted information can be used to improve previous patterns computed in the analysis. This might be done by "propagating" the information backwards in the analysis process.

4.4.3 Goal Level Assertions

Assertions at the goal level refer to the state of the variables of the clause just at the point where the assertion appears: between two literals, after the head of a clause or after the last literal of a clause.[2] We propose adding extra-literals which enclose all necessary information referring to a given program point in a clause. It takes the form:

`...,` $goal_1$`, trust(`*declaration*`),` $goal_2$`, ...`

where the information in the `trust` literal should be valid before calling $goal_2$ and also after calling $goal_1$, that is, at the success point for $goal_1$ and at the call point of $goal_2$. The information given by `trust` literals can refer to any of the variables in the clause. The information is expressed using the same kind of declarations as in the predicate level assertions. This allows a uniform format for the declarations of properties in assertions at both the predicate and the goal level. These assertions are related to predicate level `trust` assertions in the sense that they give information that should be trusted by the compiler. Therefore, they have similar uses and a similar treatment that them.

4.5 Dealing with Standard Prolog

In this section we discuss different solutions for analyzing the full standard Prolog language. In order to do so we have divided the complete set of builtins offered by the language in several classes.

4.5.1 Builtins as Abstract Functions

Many Prolog builtins can be dealt with efficiently and accurately during analysis by means of functions which capture their semantics. Such functions provide an (as accurate as possible) abstraction of every success substitution for any call to the corresponding builtin. This applies also to goal independent analyses, with minor modifications. It is interesting to note that the functions that describe

[2]Similar assertions can be used at other levels of granularity, from between head unifications to even between low level instructions, but we will limit the discussion for concreteness to goal-level program points.

builtin predicates are very similar in spirit to `trust` assertions. This is not surprising, if builtins are seen as Prolog predicates for which the code is not available. Since most of the treatment of builtins is rather straightforward, the presentation is very brief, concentrating on the more interesting cases of meta-predicates and dynamic predicates. In order to avoid reference to any particular abstract domain any functions described will be given in terms of simple minded `trust` assertions. For the reader interested in the details, the source code for the PLAI analyzer (available by ftp from `clip.dia.fi.upm.es`) contains detailed functions for all Prolog builtins and for a large collection of well known abstract domains. For a description of such functions for some builtins in a different domain see e.g. [CF92].

Control flow predicates include `true` and `repeat`, which have a simple treatment: identity can be used (i.e., they can be simply ignored). The abstraction of `fail` and `halt` is $\perp$. For cut (`!`) it is also possible to use the identity function (i.e., ignore it). This is certainly correct in that it only implies that more cases than necessary will be computed in the analysis upon predicate exit, but may result in some cases (specially if *red cuts* –those which modify the meaning of a program– are used) in a certain loss of accuracy. This can be addressed by using a semantics which keeps track of sequences, rather than sets, of substitutions, as shown in [CRV94]. Finally, exception handling can also be included in this class. The methods used by the different Prolog dialects for this purpose have been unified in the Prolog standard into two builtins: `catch` and `throw`. We propose a method for dealing with this new mechanism: note that, since analysis in general assumes that execution can fail at any point, literals of the form `catch(Goal,Catcher,Recovery)` (where execution starts in `Goal` and backtracks to `Recovery` if the exception described by `Catcher` occurs) can be safely approximated by the disjunction `(Goal;Recovery)`, and simply analyzed as a meta-call. The correctness of this transformation is based on the fact that no new control paths can appear due to an exception, since those paths are a subset of the ones considered by the analysis when it assumes that any goal may fail. The builtin `throw`, which explicitly raises an exception, can then be approximated by directly mapping it to failure, i.e. $\perp$.

The function corresponding to `=` is simply abstract unification. Specialized versions of the full abstract unification function can be used for other builtins such as \=, `functor`, `arg`, `univ` (`=..`), and `copy_term`. Other term and string manipulation builtins are relatively straightforward to implement. Arithmetic builtins and base type tests such as `is`, `>`, `@>`, `integer`, `var`, `number`, etc., usually also have a natural mapping in the abstract domain considered. In fact, their incomplete implementation in Prolog is an invaluable source of information for the analyzer upon their exit (which assumes that the predicate did not fail — failure is of course always considered as an alternative). For example, their mappings will include relations such as "`:- trust pred is(X,Y) : true => (ground(X),ground(Y)).`" or "`:- trust pred var(X) : true => free(X).`" On the contrary, `==`, `\==`, and their arithmetic counterparts, are somewhat more involved, and are implemented (in the same way as with the term manipulation builtins above) by using specialized versions of the abstract unification function.

Output from the program does not directly pose any problem since the related predicates do not instantiate any variables or produce any other side effects beyond modifying external streams, whose effect can only be seen during input to the program. Thus, identity can again be used in this case. On the other hand, the external input cannot be determined beforehand. The main problem happens to be the success substitution problem. In the general case, analysis can always proceed by simply assuming topmost success substitutions in the domain.

The treatment of *directives* is somewhat peculiar. The directive `dynamic` is used to declare predicates which can be modified at run-time. Dynamic predicates will be considered in detail below. The directive `multifile` specifies that the definition of a predicate is not complete in the program. Multifile predicates can therefore be treated as either dynamic or imported predicates — see Section 4.6. The directives `include` and `ensure_loaded` must specify an accessible file, which can be read in and analyzed together with the current program. The directive `initialization` specifies new (concrete) entry points to the program.

4.5.2 Meta-Predicates

Meta-predicates are predicates which use other predicates as arguments. All user defined meta-predicates are in this class but their treatment can be reduced to the treatment of the meta-call builtins they use. Such meta-calls are literals which call one of their arguments at run-time, converting at the time of the call a term into a goal. Builtins in this class are not only `call`, but also `bagof`, `findall`, `setof`, negation by failure, and `once` (single solution). Calls to the solution gathering builtins can be treated as normal (meta-)calls since most analyzers are "collecting" in the sense that they always consider all solutions to predicates. Negation by failure (`\+`) can be defined in terms of `call` and cut, and can be dealt with by combining the treatment of cut with the treatment of meta-calls. Single solution (`once`) can be dealt with in a similar way since it is equivalent to "`once(X) :- call(X), !.`".

Since meta-call builtins convert a term into a goal, they can be difficult to deal with if it is not possible to know at compile-time the exact nature of those terms [Deb89b, HWD92]. In particular, the success substitution problem for the meta-call appears, as well as the extra call pattern problem (within the code defining the corresponding predicate, and for the possible calls which can occur from such code). Both problems can be dealt with using the techniques in Section 4.3. First, topmost call patterns can be used for all predicates in the program, second, and alternatively, the renaming transformation can also be applied. In this case meta-calls that are fully determined either by declaration or as a result of analysis, and incorporated into the program text will call the more optimized version. Analysis will have taken into account the call patterns produced by such calls since they they would have been entered and analyzed as normal calls. I.e., the following transformation will take place:

```
..., trust(term(X,p(Y))), call(X),...   ⟹   ..., opt_p(Y),...
```

Meta-calls that are partially determined, such as, for example,

```
..., trust(depth(X,p/1)), call(X), ...
```

are a special case. One solution is not to rename them. In that case they will be treated as undetermined meta-calls. Alternatively, the solution in the second item above can be used. It is necessary in this case to ensure that the optimized program will be entered upon reaching a partially determined meta-call. This can be done dynamically, using a special version of `call/1` or by providing binary predicates which transform the calls into new predicates which perform a mapping of the original terms (known from the analysis) into the renamed ones. Using this idea the example above may be transformed into a new literal and a new clause, as follows:

```
..., opt_call(X),...              opt_call(p(X)) :- opt_p(X).
```

Undetermined meta-calls will not be renamed, and thus will call the original (less optimized) code. This fulfills the correctness requirement, since these calls would not have been analyzed, and therefore can not be allowed to call the optimized code.

More precise solutions to both problems are possible if knowledge regarding the terms to be converted is available at compile-time. Thus, following [Deb89b], we can distinguish between:

- *Completely determined* meta-calls. These are calls in which the term (functor and arguments) is given in the program text (this is often the case for example in many uses of `bagof`, `findall`, `setof`, `\+`, and `once`), or can be inferred via some kind of analysis, as proposed in [Deb89b]. In the latter case they can even be incoporated into the program text before analysis. These calls can be analyzed in a straightforward way.

- *Partially determined* meta-calls. The exact term cannot be statically found, but at least its main functor can be determined by program analysis. Then, since the predicate that will be called at run-time is known, it is sufficient for analysis to enter only this predicate using the appropriate projection of the current abstract call substitution on the variables involved in the call.

- *Undetermined* meta-calls.

The first two classes distinguish subclasses of the *fully determined* predicates of [Deb89b], where certain interesting types of programs are characterized which allow the static determination of this generally undecidable property. Relying exclusively on program analysis, as in [Deb89b], has the disadvantage that it restricts the class of programs which can be optimized to those which are fully determined. Our previous solution solves the general case.

There are other possible solutions to the general case. The first and simplest one is to issue a warning if an undetermined meta-call is found and ask the user to provide information regarding the meta-terms. This can be easily done via program-point `trust` assertions. For example, the following assertion:

```
..., trust(( term(X,p(Y)) ; term(X,q(Z)) )), call(X), ...
```

states that the term called in the meta-call is either `p(Y)` or `q(Z)`. Note also that this is in some way similar to giving entry mode information for the `p/1` and `q/1` predicates. This suggests another solution to the problem, which has been used before in Aquarius [VD92], in MA3 [WHD88], and in previous versions of the PLAI analyzer [BGCH93]. The idea (cast in the terms of our discussion) is to take the position that meta-calls are *external calls*. Then, since `entry` assertions have to be closed with respect to external calls it is the user's responsibility to declare any entry points and patterns to predicates which can be "meta-called" via `entry` assertions. Accuracy of the analysis will depend on that of the information supplied by the user. These solutions have the disadvantage of putting the burden on the user — something that we would like to avoid at least for naive users. Our alternative solutions are completely transparent to the user.

4.5.3 Database Manipulation and Dynamic Predicates

Database manipulation builtins include `assert`, `retract`, `abolish`, and `clause`. These builtins (with the exception of `clause`) affect the program itself by adding to or removing clauses from it. Predicates that can be affected by such builtins are called dynamic predicates and must usually be declared as such in modern Prolog implementations (and this is also the case in the ISO standard).

The potential problems created by the use of the database manipulation builtins are threefold:

- The extra call pattern problem appears again since the literals in the body of the new clauses that are added dynamically can produce new and different call patterns not considered during analysis.
- The success substitution problem also appears for literals which call dynamic predicates ("dynamic literals"). Even if abstract success substitutions can be computed from any static definition of the predicate which may be available at compile-time, it may change during program execution.
- There exists the additional problem of computing success substitutions for the calls to the database manipulation builtins themselves. We call this the *database builtin success substitution problem.*

Next we propose solutions to the three problems mentioned above. Note that the builtin `clause` —which can be viewed as a special case of retract— does not modify the database and thus clearly only has the third problem.

Solving the extra call pattern problem. From the correctness point of view, the extra call pattern problem only arises from the use of `assert`, but not from the use of `abolish` or `retract`. These predicates do not introduce new clauses in the program, and thus they do not introduce any new call patterns. This is true even for "intelligent" analyses which can infer definite success or failure of some goals, because these analyses must take `retract` into account to do so, or otherwise would themselves not be correct in general. Therefore, retraction is not a problem in our case. On the other hand, it is conceivable that more accuracy could be obtained if these predicates were analyzed more precisely since removing clauses may remove call patterns which in turn could make the analysis more precise. We discuss this in the context of incremental analysis at the end of the section. The discussion is general enough to subsume the above mentioned intelligent analyses.

The `assert` predicate is much more problematic, since it can introduce new clauses and through them new call patterns. The problem is compounded by the fact that asserted clauses can call predicates which are not declared as dynamic, and thus the effect is not confined to dynamic predicates. In any case, and as pointed out in [Deb89b], not all uses of `assert` are equally damaging. To distinguish these uses, we propose to divide dynamic predicates into the following types:

- `memo` only facts which are logical consequences of the program itself are asserted;
- `data` only facts are asserted, or, if clauses are asserted, they are never called (i.e., only read with `clause` or `retract`);
- `local_call` the dynamic predicate only calls other dynamic predicates;
- `global_call`.

The first two classes correspond to the *unit-assertive* and *green-assertive* predicates of [Deb89b], except that we have slightly extended the unit-assertive type by also considering in this type arbitrary predicates which are asserted/retracted but never called. Clauses used in this way can be seen as just recorded terms: simply a set of facts for the predicate symbol `:-/2`.

`data` predicates are guaranteed to produce no new call patterns and therefore they are safe with respect to the extra call pattern problem.[3] This is also the case for `memo` predicates since they only assert facts.[4] If all dynamic predicates are of the `local_call` type, then the analysis of the static program is correct except for the clauses defining the dynamic predicates themselves. Analysis can even ignore the clauses defining such predicates. Optimizations can then be performed over the program text except for those clauses, which in any case may not be such a big loss since in some systems such clauses are not compiled, but rather interpreted.

While the classification mentioned above is useful, two problems remain. The first one is how to detect that dynamic procedures are in the classes that are easy to analyze (dynamic predicates in principle need to be assumed in the `global_call` class). This can be done through analysis for certain programs, as shown in [Deb89b], but, as in the case of meta-calls, this does not offer a solution in all cases.

The general case in which `global_call` dynamic predicates appear in the program is similar to that which appeared with undetermined meta-calls. In fact, the calls that appear in the bodies of asserted clauses can be seen as undetermined meta-calls, and similar solutions apply. Additionally, the static clauses of the dynamic predicates themselves are subject to the same treatment as the rest of the program, and therefore subject to full optimization. Clearly, this solution can be combined with the previous ones when particular cases can be identified.

Solving the dynamic literal success substitution problem. If only `abolish` and `retract` are used in the program, the abstract success substitutions of the static clauses of the dynamic predicates are a safe approximation of the run-time success substitutions. However, a loss of accuracy can occur, as the abstract success substitution for the remaining clauses (if any) may be more particular. In the presence of `assert`, a correct (but possibly inaccurate) analysis is obtained by using appropriate topmost abstract substitutions. Finally, note that in the case of `memo` predicates (and for certain properties) this problem is avoided

[3] In fact, the builtins `record` and `recorded` provide the functionality of `data` predicates but without the need for dynamic declarations and without affecting global analysis. However, those builtins are now absent from the Prolog standard.

[4] Note however that certain analyses, and especially cost analyses which are affected by program execution time, need to treat these predicates specially.

since the success substitutions computed from the static program are correct.

Solving the database builtin success substitution problem. This problem does not affect `assert` and `abolish` since the success substitution for calls to these builtins is the same as the call substitution. On the other hand, success substitutions for `retract` (and `clause`) are more difficult to obtain. However, appropriate topmost substitutions can always be safely used. In the special case of dynamic predicates of the `memo` class, and if the term used as argument in the call to `retract` or `clause` is at least partially determined, abstract counterparts of the *static* clauses of the program can be used as approximations in order to compute a more precise success substitution.

Dynamic analysis and optimization. There is still another, quite different and interesting solution to the problem of dynamic predicates, which is based on incremental global analysis as presented in Chapter 2. Note that in order to implement `assert` some systems include a copy of the full compiler at runtime. The idea would be to also include the (incremental) global analyzer and the analysis information for the program, computed for the static part of the program. The program is in principle optimized using this information but the optimizer is also assumed to be incremental. After each non-trivial assertion or retraction (some cases may be treated specially) the incremental global analysis and optimizer are rerun and any affected parts of the program reanalyzed (and reoptimized). This has the advantage of having fully optimized code at all times, at the cost of increasing the cost of calls to database manipulation predicates and of executable size. A system along these lines has been built by us for a parallelizing compiler. The results presented in Chapter 2 show that such a reanalysis can be made in a very small fraction of the normal compilation time.

4.6 Program Modules

The main problem with studying the impact of modularity in analysis (and the reason we have left the issue until this section) is the lack of even a de-facto standard. There have been many proposals for module systems in logic programming languages (see [BLM94]). For concreteness, we will focus on that proposed in the new draft ISO standard [PRO95]. In this standard, the module interface is *static*, i.e. each module in the program must declare the procedures it exports,[5] and imports. The `module` directive is used for this.

As already pointed out in [HWD92] `module` directives provide the entry points for the analysis of a module for free. Thus, as far as entry points are concerned, only exported predicates need be considered. They can be analyzed using the substitutions declared in the `entry` assertions if available, and topmost otherwise. The analysis of literals which call imported predicates requires new approaches, some of which are discussed in the following paragraphs. One advantage of modules is that they help encapsulate the propagation of complex situations such as with `global_call` dynamic predicates.

Compositional Analysis. Modular analyses based on compositional semantics (such as, for example, that of [CDG93]) can be used to analyze programs split in modules. Such analyses leave the abstract substitutions for the predicates whose definitions are not available *open*, in the sense that some representation of the literals and their interaction with the abstract substitution is incorporated as a handle into the substitutions themselves. Once the corresponding module is analyzed and the (abstract) semantics of such open predicates known, substitutions can be composed via these handles. The main drawback of this interesting approach is that the result of the analysis is not definite if there are open predicates. In principle, this would force some optimizations to be delayed until the final composed semantics is known, which in general can only be done when the code for all modules is available. Therefore, although analysis can be performed for each module separately, optimizations (and thus, compilation) cannot in principle use the global information.

Incremental Analysis. When analyzing a module, each call to a predicate not declared in it is mapped to $\perp$. Each time analysis information is updated, it is applied directly to the parts of the analysis where this information may be

[5]This is in contrast with other module systems used in some Prolog implementations that allow entering the code in modules at arbitrary points other than those declared as exported. This defeats the purpose of modules. We will not discuss such module systems since the corresponding programs in general need to be treated as non modular programs from the point of view of analysis.

relevant. Incremental analysis as presented in Chapter 2 is conservative: it is correct and optimal. By optimal we mean that if we put together in a single module the code for all modules (with the necessary renaming to avoid name clashes) and analyze it in the traditional way, we will obtain the same information. However, incremental analysis, in a very similar way to the previous solution, is only useful for optimization if the code for all modules is available, since the information obtained for one isolated module is only partial. On the other hand, if optimization is also made incremental, then this does present a solution to the general problem: modules are optimized as much as possible assuming no knowledge of the other modules. Optimizations will be correct with respect to the partial information available at that time. Upon module composition incremental reanalysis and reoptimization will make the composed optimized program always correct.

Note that Prolog compilers are incremental in the sense that at any point in time new clauses can be compiled into the program. Incremental analysis (aided by incremental optimization) allows the combination of full interactive program development with full global analysis based optimization.

Trust-Enhanced Module Interface. In [PRO95] imported predicates have to be declared in the module importing them and such a module can only be compiled if all the module interfaces for the predicates it imports are defined, even if the actual code is not yet available. Note that the same happens for most languages with modules (e.g., Modula). When such languages have some kind of global analysis (e.g., type checking) the module interface also includes suitable declarations. We propose to augment the module interface definition so that it may include `trust` assertions for the exported predicates. Each call to a predicate not defined in the module being analyzed but exported by some module interface is in principle mapped to appropriate topmost substitutions. But if in the module interface there are one or more `trust` assertions applicable to the call pattern, such assertions will be used instead. Any call to a predicate not defined in that module and not present in any of the module interfaces can be safely mapped to $\perp$ during analysis (this corresponds to mapping program errors to failure – note that error can also be treated alternatively as a first class element in the analysis). The advantages are that we do not need the code for other modules and also that we can perform optimizations using the (inaccurate) analysis information obtained in this way.

Analysis using the trust-enhanced interface is correct, but it may be suboptimal. This can only be avoided if the programmer provides the most accurate `trust` assertions. The disadvantage of this method is that it requires the trust-enhanced interface for each module. However, the process of generating these `trust` assertions can be automated. Whenever the module is analyzed, the call/success patterns for each exported predicate in the module which are obtained by the analysis are written out in the module interface as `trust` assertions. From there, they will be seen by other modules during their analysis and will improve their exported information. A global fixpoint can be reached in a distributed way even if different modules are being developed by different programmers at different times and running the analysis only locally, provided that, as required by the module system, the module interfaces (but not necessarily the code) are always made visible to other modules.

Summary. In practice it may be useful to use a combination of incremental analysis and the trust-enhanced module interface The trust-enhanced interface can be used during the development phase to compile modules independently. Once the actual code for all modules is present incremental analysis can be used to analyze modules loading them one after the other. In this way we obtain the best accuracy.

Multifile predicates (those defined over more than one file or module) also need to be treated in a special way. They can be easily identified due to the `multifile` declaration. They are similar to `dynamic` predicates (and also imported predicates) in that if we analyze a file independently of others, some of the code of a predicate is missing. We can treat such predicates as dynamic predicates and assume topmost substitutions as their abstract success substitutions unless there is a `trust` assertion for them. When the whole program composed of several files is compiled, we can again use incremental analysis. At that point, clauses for predicates are added to the analysis using *incremental addition*, presented in Section 2.3 (regardless of whether these clauses belong to different files and/or modules).

A case also worth discussing is that of libraries. Usually utility libraries provide predicates with an intended use. The automatic generation of `trust` assertions after analysis can be used for each library to provide information regarding the exported predicates. This is done for different uses and the generated assertions are stored in the library interface. With this scheme it is not necessary to analyze a library predicate when it is used in different programs. Instead, it is only analyzed once, and the information stored in the `trust` assertion is used from then on. If new uses of the library predicates arise for a given program, the library code can be reanalyzed and recompiled, keeping track of this new use for future compilations. An alternative approach is to perform a goal independent analysis of the library, coupled with a goal dependent analysis for the particular call patterns used thereafter [CGBH94].

4.7 Chapter Conclusions

We have studied several ways in which optimizations based on static analysis can be guaranteed correct for programs which use the full power of Prolog, including modules. We have also introduced several types of program assertions that can be used to both increase the accuracy and efficiency of the analysis and to express its results. The proposed techniques offer different trade-offs between accuracy, analysis cost, and user involvement. We argue that the presented combination of known and novel techniques offers a comprehensive solution for the correct analysis of arbitrary programs using the full power of the language.

Part II

Program Specialization based on Abstract Interpretation

Chapter 5

Abstract Multiple Specialization

Program specialization optimizes programs for known values of the input. It is often the case that the set of possible input values is unknown, or this set is infinite. However, a form of specialization can still be performed in such cases by means of abstract interpretation, specialization then being with respect to abstract values (substitutions), rather than concrete ones. We study the multiple specialization of logic programs based on abstract interpretation. This involves in principle, and based on information from global analysis, generating several versions of a program predicate for different uses of such predicate, optimizing these versions, and, finally, producing a new, "multiply specialized" program. While multiple specialization has received theoretical attention, it has not previously been incorporated in a compiler and its effects quantified. In this chapter such a study is performed in the context of a parallelizing compiler. Abstract executability, the main concept underlying the application of abstract specialization, is formalized, and a novel approach to the design and implementation of the specialization system is proposed. The resulting implementation techniques result in identical specializations to those of the best previously proposed techniques but require little or no modification of some existing abstract interpreters. Our results show that, using the proposed techniques, the resulting "abstract multiple specialization" is indeed a relevant technique in practice. In particular, in the parallelizing compiler application, a good number of run-time tests are eliminated and invariants extracted automatically from loops, resulting generally in lower overheads and in several cases in increased speedups.

5.1 Introduction

Compilers often use static knowledge regarding invariants in the execution state of the program in order to optimize the program for such particular cases [AU77]. Standard optimizations of this kind include dead-code elimination, constant propagation, conditional reduction, code hoisting, etc. A good number of optimizations can be seen as special cases of partial evaluation [CD93, JGS93, DGT96]. The main objective of program specialization is to automatically overcome losses in performance which are due to general purpose algorithms by specializing the program for known values of the inputs. In the case of logic programs partial evaluation takes the form of partial deduction [LS91, Kom92], which is closely related to other techniques used in functional languages such as "driving" [Gr94]. Much work has been done in logic program partial deduction and specialization of logic programs (see e.g. [GB90, GCS88, JLW90]).

It is often the case that the set of possible input values is unknown, or this set is infinite. However, a form of specialization can still be performed in such cases by means of abstract interpretation [CC77]. Specialization can then be performed with respect to abstract values, rather than concrete ones. Such abstract values are safe approximations in a "representation domain" of a set of concrete values. Standard safety results imply that the set of concrete values represented by an abstract value is a superset (or a subset, depending on the property being abstracted and the optimizations to be performed) of the concrete values that may appear at a certain program point in all possible program executions. Thus, any optimization allowed in the superset (respectively, subset) will also be correct for all the run-time values. The possible optimizations include again dead-code elimination, (abstract) constant propagation, conditional reduction, code hoisting, etc., which can again be viewed as a special case of a form of "abstract partial evaluation." Consider, for example, the following general purpose addition predicate which can be used when at least any two of its arguments are integers:

```
plus(X,Y,Z):-
     int(X),int(Y),!,Z is X + Y.
plus(X,Y,Z):-
     int(Y),int(Z),!,X is Z - Y.
```

```
plus(X,Y,Z):-
     int(X),int(Z),!,Y is Z - X.
```

If, for example, for all calls to this predicate in the program it is known from global analysis that the first and second arguments are always integers, then the program can be specialized as follows

```
plus(X,Y,Z):-
     Z is X + Y.
```

which would clearly be more efficient because no tests are executed. The optimization above is based on "abstractly executing" the tests, i.e. reducing predicate calls to `true`, `fail`, or a set of primitives (typically, unifications) based on the information available from abstract interpretation. The first contribution of this chapter is to formalize the concept of *abstract executability*, first introduced informally in [GH91], which is instrumental in the optimization process.

It is also often the case that a procedure has different uses within a program, i.e. it is called from different places in the program with different (abstract) input values. In principle, optimizations are then allowable only if the optimization is applicable to all uses of the predicate. However, it is possible that in several different uses the input values allow different and incompatible optimizations and then none of them can take place. This can be overcome by means of "multiple program specialization" [JLW90, GH91, Bru91, Win92] (the counterpart of polyvariant specialization [Bul84]), where different versions of the predicate are generated for each use, so that each one of them is optimized for the particular subset of input values with which each version is to be used. For example, in order to allow maximal optimization, different versions of the `plus/3` predicate should be generated for the following calls:

```
..., plus(X1,Y1,Z1), plus(X2,Y2,Z2), ...
```

if, for example, `X1`, and `Y1` are known to be bound to integers, but no information is available on `X2`, `Y2`, and `Z2`.

While the technique outlined above is very interesting in principle, many practical issues arise, some of which have been addressed in different ways in previous work [JLW90, GH91, Bru91, Win92]. One is the method used for selection of the appropriate version for each call at run-time. This can be done quite simply by renaming calls and predicates. For example, for the situation in the example above this would result in the following calls and additional version `plus1/3` of the `plus/3` predicate:

```
..., plus1(X1,Y1,Z1), plus(X2,Y2,Z2), ...

plus1(X,Y,Z):-
     Z is X + Y.
```

This approach has the potential problem that, in order to create a "path" from the call to the specialized predicate, some intermediate predicates may have to also be specialized even if no optimization is performed for them, with a resulting additional increase in code size. Jacobs et al. [JLW90] propose instead the use of simple run-time tests to discern the different possible call modes and determine the appropriate version dynamically. This is attractive in that it avoids the "spurious" specializations of the previous solutions (and thus reduces code size), but is also dangerous as such run-time tests themselves imply a cost which may be in unfavorable cases higher than the gains obtained due to multiple specialization.

Another problem, which will be discussed in more depth later, is that it is not straightforward to decide the optimum number of versions for each predicate. In general, the more versions generated, the more optimizations possible, but this can lead to an unnecessarily large increase in program size.

Winsborough [Win92] presents an algorithm, based on the notion of minimal function graphs [JM86], that solves the two problems outlined above. A new abstract interpretation framework is introduced which is tightly coupled with the specialization algorithm. The combination is proved to produce a program with multiple versions of predicates that allow the maximum optimizations possible while having the minimal number of versions for each predicate.

The body of work in the area and Winsborough's fundamental results, both briefly summarized above, and the fact that abstract interpretation is becoming a practical tool in logic program compilation [HWD92, VD92, MH92, SCWY91, BGH94b], suggests that it may be worthwhile to study whether multiple specialization could be useful in practice. However, little or no evidence on the practicality of abstract interpretation driven multiple specialization in logic programs has been provided previous to our work [PH95, PH97b]. The second contribution of this work is to fill this gap. Improvements for a few small, hand-coded examples were reported in [MJMB89, VD92]. More recently, an implementation of multiple specialization has also been reported in [KMM$^+$95, KMM$^+$96], applied to CLP($\mathcal{R}$). Given that the specialization algorithm used in that work is relatively naïve, the results are interesting in that they provide experimental evidence on the relevance of multiple specialization even using a naïve strategy. We report on the implementation of multiple specialization in a parallelizing compiler for Prolog which incorporates an abstract interpretation-based global analyzer. We present a performance analysis of multiple specialization in this system, in which a minimization of the number of versions is performed. We argue that our results show that multiple specialization is indeed practical and useful in the application, and also that such results shed some light on its possible practicality in other applications.

Finally, we also propose a novel technique for the practical implementation of multiple specialization. While the analysis framework used by Winsborough is interesting in itself, several generic analysis engines, such as PLAI [MH92, MH90a] and GAIA [CV94], which greatly facilitate construction of abstract interpretation analyzers, are available, well understood, and in comparatively wide use. We believe that it is of practical interest to specify a method for multiple specialization which can be incorporated in a compiler using a minimally modified existing generic analyzer. This was previously attempted in [GH91], where a simple program transformation technique which has no direct communication with the abstract interpreter is proposed, as well as a simple mechanism for detecting cases in which multiple specialization is profitable. However, this technique is not capable of detecting all the possibilities for specialization or producing a minimally specialized program. It also requires running the interpreter several times after specialization, repeating the analysis–program transformation cycle until a fixpoint is reached. The third contribution of the work presented in this chapter is to propose an algorithm which achieves the same results as those of Winsborough's but with only a slight modification of a standard abstract interpreter and by assuming minimal communication with such interpreter (namely, access to the memoization tables). Our algorithm can be seen as an implementation technique for Winsborough's method in the context of standard analyzers. Regarding the problem of version selection, our implementation uses predicate renaming to create paths from calls to specialized predicates. However, we argue that our technique is equally valid in the context of run-time test based clause selection.

The structure of this chapter is as follows. The notion of abstract executability is formalized and discussed in Section 5.2. In Section 5.3 we propose a naive implementation method for multiple specialization based on abstract interpretation. In Section 5.4 we then present an algorithm for minimizing the number of versions and show that it terminates and is indeed minimal by reasoning over the lattice of transformed programs. Then Section 5.5 presents the application where multiple specialization will be applied: automatic parallelization. Section 5.6 shows the design of the abstract specializer and an example of a specialized program. Section 5.7 presents the experimental results, which are then discussed in Section 5.8. Finally, Section 5.9 concludes.

5.2 Abstract Execution

The concept of *abstract executability* was, to our knowledge, first introduced informally in [GH91]. It allows reducing at compile-time certain literals in a program to the value *true* or *false* using information obtained with abstract interpretation. That work also introduced some simple semantics-preserving program transformations and showed the potential of the technique, including elimination of invariants in loops. We introduce in the following an improved formalization of abstract executability.

We start by introducing (recalling) some notation. A *program* is a sequence of *clauses*. Clauses are of the form h :- b, where h is an atom and b is a possibly empty conjunction of literals. As in previous chapters, clauses in the program are written with a unique subscript attached to the head atom (the clause number), and dual subscript (clause number, body position) attached to each literal in the body atom e.g. H_k :- $B_{k,1}, \ldots, B_{k,n_k}$ where $B_{k,i}$ is a subscripted literal. The clause may also be referred to as clause k, the subscript of the head atom, and each literal in the program is uniquely identified by its subscript k, i.[1] We will

[1]Our implementation supports essentially all the built-ins of ISO-Prolog, as presented in Chapter 4. However, for simplicity we avoid their discussion except in cases where it may be specially relevant. This includes for example programs which have if-then-else's in the body of

denote by $\lambda_{k,i}$ the *abstract call substitution* for the literal $L_{k,i}$ which is the abstract substitution just before calling the literal $L_{k,i}$.The set of variables in a literal L is represented as $var(L)$. The restriction of the substitution θ to the variables in L is denoted $\theta|_L$.

Operationally, each literal L in a program P can be viewed as a procedure call. Each run-time invocation of the procedure call L will have a local *environment* e, which stores the particular values of each variable in $var(L)$ for that invocation. We will write $\theta \in e(L)$ if θ is a substitution such that the value of each variable in $var(L)$ is the same in the environment e and the substitution θ.

Definition 5.2.1 [Run-time Substitution Set] Given a literal L from a program P we define the *run-time substitution set* of L in P as

$$RT(L, P) = \{\theta|_L : e \text{ is a run-time environment for } L \text{ and } \theta \in e(L)\}$$

$RT(L, P)$ is not computable in general. The set of run-time environments for a literal is not known at compile-time. However, it is sometimes possible to find a set of bindings which will appear in *any* environment for L. These "invariants" can be synthesized in a substitution θ_s such that $\forall \theta \in RT(L, P)\ \exists \theta_d : L\theta = L\theta_s\theta_d$. Note that it is always possible to find a trivial $\theta_s = \epsilon$, the empty substitution, which corresponds to having no static knowledge of the run-time environment. In this case, we can simply take $\theta_d = \theta$ for any θ.

The substitutions θ_s and θ_d correspond to the so-called *static* and *dynamic* values respectively in partial evaluation [JGS93]. As a result, we can specialize L for the statically known data θ_s. Specialization is then usually performed by *unfolding* $L\theta_s$. If all the leaves in the SLD tree for $L\theta_s$ are failing nodes and $L\theta_s$ is pure (i.e., its execution does not produce side-effects), then the literal L can be replaced by $false$. If all the leaves are failing nodes except for one which is a success node and $L\theta_s$ is pure then L can be replaced by a set of unifications on $var(L\theta_s)$ which have the same effect as actually executing $L\theta_s\theta_d$ in P. If such set of unifications is empty, L can be replaced by *true*.

clauses, such as those generated by automatic parallelization, as will be seen in Section 5.5.2. This construct poses no additional theoretical difficulties: the same effect (modulo perhaps some run-time overhead) can be achieved using conjunctions of literals and the cut.

The goal of abstract specialization is also to replace a literal by $false$, $true$ or a set of unifications, but rather than starting from $RT(L,P)$ it will use information on $RT(L,P)$ provided by abstract interpretation, i.e., the abstract call substitution for L. For simplicity, we will restrict our discussion to replacing L with $false$ or $true$.

Definition 5.2.2 [Trivial Success Set] Given a literal L from a program P we define the *trivial success set* of L in P as

$$TS(L,P) = \begin{cases} \left\{ \begin{array}{l} \theta|_L : L\theta \text{ succeeds exactly once in } P \\ \text{with empty answer substitution } (\epsilon) \end{array} \right\} & \text{if L is pure} \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 5.2.3 [Finite Failure Set] Given a literal L from a program P we define the *finite failure set* of L in P as

$$FF(L,P) = \begin{cases} \{\theta|_L : L\theta \text{ fails finitely in } P\} & \text{if L is pure} \\ \emptyset & \text{otherwise} \end{cases}$$

Note that if two distinct literals $L_{k,i}$ and $L_{l,j}$ are equal up to renaming then the sets $TS(L_{k,i},P)$ (resp. $FF(L_{k,i},P)$) and $TS(L_{l,j},P)$ (resp. $FF(L_{l,j},P)$) will also be equal up to renaming. However, there is no a priori relation between $RT(L_{k,i},P)$ and $RT(L_{l,j},P)$.

Definition 5.2.4 [Elementary Literal Replacement] *Elementary Literal Replacement* (ER) of a literal L in a program P is defined as:

$$ER(L,P) = \begin{cases} true & \text{if } RT(L,P) \subseteq TS(L,P) \\ false & \text{if } RT(L,P) \subseteq FF(L,P) \\ L & \text{otherwise} \end{cases}$$

Note that given the definitions of $TS(L,P)$ and $FF(L,P)$, any literal L which is not dead code and produces some side-effect (i.e., L is not pure) will not be affected by elementary literal replacement, i.e., $ER(L,P) = L$.

Theorem 5.2.5 [Elementary Replacement] Let P_{ER} be the program obtained by replacing each literal $L_{k,i}$ in P by $ER(L_{k,i}, P)$. P and P_{ER} produce the same computed answers and side-effects.

The idea is to optimize a program by replacing the execution of $L\theta$ with the execution of either the builtin predicate *true* or $fail$, which can be executed in zero or constant time. Even though the above optimization may seem not very widely applicable, for many builtin predicates such as those that check basic types or meta-logical predicates that inspect the instantiation state of terms and as we will see in Section 5.7, this optimization is indeed very relevant. However, elementary replacement is not directly applicable because $RT(L, P)$, $TS(L, P)$, and $FF(L, P)$ are generally not known at specialization time.

Definition 5.2.6 [Abstract Trivial Success Set] Given an abstract domain D_α we define the *abstract trivial success set* of L in P as

$$TS_\alpha(L, P, D_\alpha) = \{\lambda \in D_\alpha : \gamma(\lambda) \subseteq TS(L, P)\}$$

Definition 5.2.7 [Abstract Finite Failure Set] Given an abstract domain D_α we define the *abstract finite failure set* of L in P as

$$FF_\alpha(L, P, D_\alpha) = \{\lambda \in D_\alpha : \gamma(\lambda) \subseteq FF(L, P)\}$$

Note that by using the least upper bound operator ($\sqcup$) of the abstract domain D_α, $TS_\alpha(L, P, D_\alpha)$ and $FF_\alpha(L, P, D_\alpha)$ could be represented by a single abstract substitution (rather than a set of them), say $\lambda_{TS_\alpha(L,P,D_\alpha)} = \sqcup_{TS_\alpha(L,P,D_\alpha)}\lambda$ and $\lambda_{FF_\alpha(L,P,D_\alpha)} = \sqcup_{FF_\alpha(L,P,D_\alpha)}\lambda$. However, this alternative approximation of the actual sets $TS(L, P)$ and $FF(L, P)$ can introduce an important loss of accuracy for some abstract domains because $\gamma(\lambda_{TS_\alpha(L,P,D_\alpha)}) \supseteq \bigcup_{TS_\alpha(L,P,D_\alpha)} \gamma(\lambda)$, thus reducing the optimizations achievable by abstract executability .

Definition 5.2.8 [Abstract Execution] *Abstract Execution* (AE) of L in P with abstract call substitution $\lambda \in D_\alpha$ is defined as:

$$AE(L, P, D_\alpha, \lambda) = \begin{cases} true & \text{if } \lambda \in TS_\alpha(L, P, D_\alpha) \\ false & \text{if } \lambda \in FF_\alpha(L, P, D_\alpha) \\ L & \text{otherwise} \end{cases}$$

If $AE(L, P, D_\alpha, \lambda) = true$ (resp. $false$) we will say that L is abstractly executable to $true$ (resp. $false$). If $AE(L, P, D_\alpha, \lambda) = L$ then L is not abstractly executable.

Theorem 5.2.9 [Abstract Executability] Let let P_{AE} be the program obtained by replacing each literal $L_{k,i}$ in P by $AE(L_{k,i}, P, D_\alpha, \lambda_{k,i})$. P and P_{AE} produce the same computed answers and side-effects.

The advantage of abstract executability as given in Definition 5.2.8 over elementary replacement is that instead of using $RT(L, P)$ which is not computable in general, such sets are approximated by abstract substitutions which for appropriate abstract domains (and widening mechanisms) will be computable in finite time.

Definition 5.2.10 [Optimal TS_α] An abstract trivial success set $TS_\alpha(L, P, D_\alpha)$ is *optimal* iff

$$(\bigcup_{\lambda \in TS_\alpha(L,P,D_\alpha)} \gamma(\lambda)) = TS(L, P)$$

Optimal abstract finite failure sets are defined similarly. One first possible disadvantage of abstract execution with respect to elementary replacement is due to the loss of information associated to using an abstract domain instead of the concrete domain. This is related to the expressive power of the abstract domain, i.e. what kind of information it provides. If $TS_\alpha(L, P, D_\alpha)$ and/or $FF_\alpha(L, P, D_\alpha)$ are not optimal then there may exist literals in the program such that $RT(L, P) \subseteq TS(L, P)$ or $RT(L, P) \subseteq FF(L, P)$ and thus elementary replacement could in principle be applied but abstract execution cannot. In general, domains will be optimal for *some* predicates but not all.

Another possible disadvantage is that even if the abstract domain is expressive enough and both $TS_\alpha(L, P, D_\alpha)$ and $FF_\alpha(L, P, D_\alpha)$ are optimal, the computed abstract substitutions may not be accurate enough to allow abstract execution. Therefore, the choice of the domain should be first guided by the predicates whose optimization is of interest so that $TS_\alpha(L, P, D_\alpha)$ and $FF_\alpha(L, P, D_\alpha)$ are as adequate as possible for them, and second by the accuracy of the abstract substitutions it provides and its computational cost.

Definition 5.2.11 [Maximal Subset] Let S be a set and let $\sqsubseteq$ be a partial order over the elements of S. We define the *maximal subset* of S with respect to $\sqsubseteq$ as

$$M_{\sqsubseteq}(S) = \{s \in S : \nexists s' \in S \ (s \neq s' \wedge s \sqsubseteq s')\}$$

Abstract execution as given in Definition 5.2.8 is not applicable in general because even though each $\lambda_{k,i}$ is computable by means of abstract interpretation, TS_α and FF_α are not computable in general. Additionally, if D_α is infinite, TS_α and FF_α may also be infinite. However, based on the observation that if $\lambda \in TS_\alpha$ then $\forall \lambda' \sqsubseteq \lambda \quad \lambda' \in TS_\alpha$, the conditions $\lambda \in TS_\alpha(L, P, D_\alpha)$ and $\lambda \in FF_\alpha(L, P, D_\alpha)$ are equivalent to $\exists \lambda' \in M_{\sqsubseteq}(TS_\alpha(L, P, D_\alpha)) : \lambda \sqsubseteq \lambda'$ and $\exists \lambda' \in M_{\sqsubseteq}(FF_\alpha(L, P, D_\alpha)) : \lambda \sqsubseteq \lambda'$ respectively and thus can be replaced in Definition 5.2.8. Unlike TS_α and FF_α, $M_{\sqsubseteq}(TS_\alpha(L, P, D_\alpha))$ and $M_{\sqsubseteq}(FF_\alpha(L, P, D_\alpha))$ are finite for any D_α with finite width. Additionally, they usually have one or just a few elements for most practical domains.

Definition 5.2.12 [Base Form] The *Base Form* of a literal L which calls predicate $Pred$ of arity n (represented as $\overline{L}$) is the literal $Pred(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ are distinct free variables.

As the number of literals in a program that call a given predicate is not bounded and in order to reduce the number of TS_α and FF_α sets that need to be computed to optimize a program, in what follows we will only consider one TS_α and FF_α per predicate which refers to its base form.

The function named *call_to_entry*, which is normally defined for each domain in most abstract interpretation frameworks, will be used to relate an abstract substitution over the variables of an arbitrary literal with the base form of the literal. The format of this function is $call\_to\_entry(L1, L2, D_\alpha, \lambda)$. Given a literal $L1$ and an abstract substitution $\lambda \in D_\alpha$ over the variables in $L1$, this function computes an abstract substitution over the variables in $L2$ which is the result of unifying $L1$ and $L2$ both with respect to concrete and abstract substitutions.

Using the base form and *call_to_entry* the conditions $\lambda \in TS_\alpha(L, P, D_\alpha)$ and $\lambda \in FF_\alpha(L, P, D_\alpha)$ in Definition 5.2.8 can be replaced by $call\_to\_entry(L, \overline{L}, D_\alpha, \lambda) \in TS_\alpha(\overline{L}, P, D_\alpha)$ and $call\_to\_entry(L, \overline{L}, D_\alpha, \lambda) \in FF_\alpha(\overline{L}, P, D_\alpha)$ respectively. The transformed conditions are not equivalent, but are sufficient. This means that correctness is guaranteed, but possibly some optimizations will be lost.

5.2.1 Optimization of Calls to Builtin Predicates.

Even though abstract executability is applicable to any predicate, in what follows we will concentrate on builtin predicates. This is because the semantics of builtin predicates does not depend on the particular program in which they appear, i.e., $\forall P, P'\ TS_\alpha(\overline{B}, P, D_\alpha) = TS_\alpha(\overline{B}, P', D_\alpha) = TS_\alpha(\overline{B}, D_\alpha)$. As a result, we can compute $TS_\alpha(\overline{B}, D_\alpha)$ and $FF_\alpha(\overline{B}, D_\alpha)$ once and for all for each builtin predicate B and they will be applicable to all literals that call the builtin predicate in any program.

Definition 5.2.13 [Operational Abstract Execution of Builtins] *Operational abstract execution* (OAEB) of a literal L with abstract call substitution λ that calls a builtin predicate B is defined as:

$$OAEB(L, D_\alpha, \lambda) = \begin{cases} true & \text{if } \exists \lambda' \in A_{TS}(\overline{B}, D_\alpha) : \\ & \quad call\_to\_entry(L, \overline{B}, D_\alpha, \lambda) \sqcup \lambda' = \lambda' \\ false & \text{if } \exists \lambda' \in A_{FF}(\overline{B}, D_\alpha) : \\ & \quad call\_to\_entry(L, \overline{B}, D_\alpha, \lambda) \sqcup \lambda' = \lambda' \\ L & \text{otherwise} \end{cases}$$

$A_{TS}(\overline{B}, D_\alpha)$ and $A_{FF}(\overline{B}, D_\alpha)$ are approximations of $M_{\sqsubseteq}(TS_\alpha(\overline{B}, D_\alpha))$ and $M_{\sqsubseteq}(FF_\alpha(\overline{B}, D_\alpha))$ respectively. This is because there is no automated method that we are aware of to compute $M_{\sqsubseteq}(TS_\alpha(\overline{B}, D_\alpha))$ and $M_{\sqsubseteq}(FF_\alpha(\overline{B}, D_\alpha))$ for each builtin predicate $\overline{B}$. For soundness it is required that both $A_{TS}(\overline{B}, D_\alpha) \subseteq TS_\alpha(\overline{B}, D_\alpha)$ and $A_{FF}(\overline{B}, D_\alpha) \subseteq FF_\alpha(\overline{B}, D_\alpha)$. We believe that a good knowledge of D_α allows finding safe approximations, and that in many cases it is easy to find the best possible approximations $A_{TS}(\overline{B}, D_\alpha) = M_{\sqsubseteq}(TS_\alpha(\overline{B}, D_\alpha))$ and $A_{FF}(\overline{B}, D_\alpha) = M_{\sqsubseteq}(FF_\alpha(\overline{B}, D_\alpha))$.

Additionally, the condition $call\_to\_entry(L, \overline{B}, D_\alpha, \lambda) \sqsubseteq \lambda'$ has been replaced by the equivalent one $call\_to\_entry(L, \overline{B}, D_\alpha, \lambda) \sqcup \lambda' = \lambda'$, where $\sqcup$ stands as before for the least upper bound, which can generally be computed effectively.

Theorem 5.2.14 [Operational Abstract Executability of Builtins] Let P be a program and let P_{OAEB} be the program obtained by replacing each literal $L_{k,i}$ in P by $OAEB(L_{k,i}, \lambda_{k,i})$ where $\lambda_{k,i}$ is the abstract call substitution for $L_{k,i}$.
If $\forall B\ A_{TS}(\overline{B}, D_\alpha) \subseteq TS_\alpha(\overline{B}, D_\alpha) \wedge A_{FF}(\overline{B}, D_\alpha) \subseteq FF_\alpha(\overline{B}, D_\alpha)$ then P_{OAEB} is

computable in finite time, and both P and P_{OAEB} produce the same computed answers and side-effects.

Example 5.2.15 Suppose we are interested in optimizing calls to the builtin predicate *ground/1* by reducing them to the value true. Then, $TS(ground(X_1)) = \{\{X_1/g\}$ where g is any term without variables $\}$. Suppose also that we use the abstract domain D_α consisting of the five elements $\{$*bottom*, *int*, $float$, $free$, *top*$\}$. These elements respectively correspond to the empty set of terms, the set of all integers, the set of floating point numbers, the set of all unbound variables, and the set of all terms. Then, the abstract version of $TS(ground(X_1))$, i.e., $TS_\alpha(ground(X_1), D_\alpha) = \{int, float, bottom\}$ is clearly not optimal (there are many ground terms which are neither integers nor floating numbers). We can take $A_{TS}(ground(X_1), D_\alpha) = \{int, float\} = M_{\sqsubseteq}(\{int, float, bottom\})$. Consider the following clause containing the literal $ground(X)$:

$$p(X, Y) :\!-\ q(Y), ground(X), r(X, Y).$$

Assume now that analysis has inferred the abstract substitution just before the literal $ground(X)$ to be $\{Y/free, X/int\}$. Then $OAEB(ground(X), D_\alpha, X/int)$ $= true$ (the literal can be replaced by $true$) because $call\_to\_entry(ground(X), ground(X_1), D_\alpha, \{X/int\}) = \{X_1/int\}$, and $X_1/int \sqcup X_1/int = X_1/int$.

If we were also interested in reducing literals that call *ground/1* to false, the most accurate $A_{FF}(ground(X_1), D_\alpha) = \{free\} = M_{\sqsubseteq}(FF_\alpha(ground(X_1), D_\alpha))$ which again is not optimal.

5.3 Multiple Specialization using Abstract Interpretation

As mentioned before, traditional, goal-driven abstract interpreters for logic programs produce as a result a program analysis graph which can be viewed as a finite representation of the (possibly infinite) set of (possibly infinite) and–or trees explored by the concrete execution [Bru91]. Execution and–or trees which are infinite can be represented finitely through a "widening" [CC92] into a rational tree. Also, the use of abstract values instead of concrete ones allows representing infinitely many concrete execution trees with a single abstract analysis graph. The graph has two sorts of nodes: those belonging to rules (also called "and–nodes") and those belonging to atoms (also called "or–nodes").

In order to increase accuracy, analyzers usually perform a form of multiple program specialization during the analysis. As a result, several nodes in the and–or graph can correspond to a single program point (in the original, non-specialized version of the program). Actual analyzers differ in the degree of specialization supported and in the way such specialization is represented, but, in general, most analyzers generate all possible versions since this allows the most accurate analysis [Bru91, MH92, MH90a, CV94]. Normally, the results of the analysis are simply "folded back" into the program: information from nodes which correspond to the same points in the original program is combined using the least upper bound operator. The main idea that we will exploit is to instead use the implicit multiply specialized program explored by the analyzer not only to improve analysis information, but also to generate a multiply specialized program in which we have more accurate information for each version and specialize each version accordingly.

Example 5.3.1 Consider the following example program, where the predicate *plus/3* is defined as before and *go/2* is known to be always called with both arguments bound to integers

```
go(A,B):-
     p(A,B,_), p(A,_,B).
p(X,Y,Z):-
     plus(X,Y,Z),
     write(Z), write(' is '),
     write(X), write(' + '), write(Y), nl.
```

Consider also the abstract domain D_α of Example 5.2.15 consisting of the five elements $\{bottom, int, float, free, top\}$. Figure 5.1 shows the analysis graph for the example program. Clearly, as there are infinitely many integer values, such graph represents an infinite number of concrete graphs. Circles are used to represent calls to builtin predicates.

In order to perform multiple specialization given the information available at the end of the analysis two problems remain: the first one is devising a method for actually materializing the versions generated taking such information as a starting

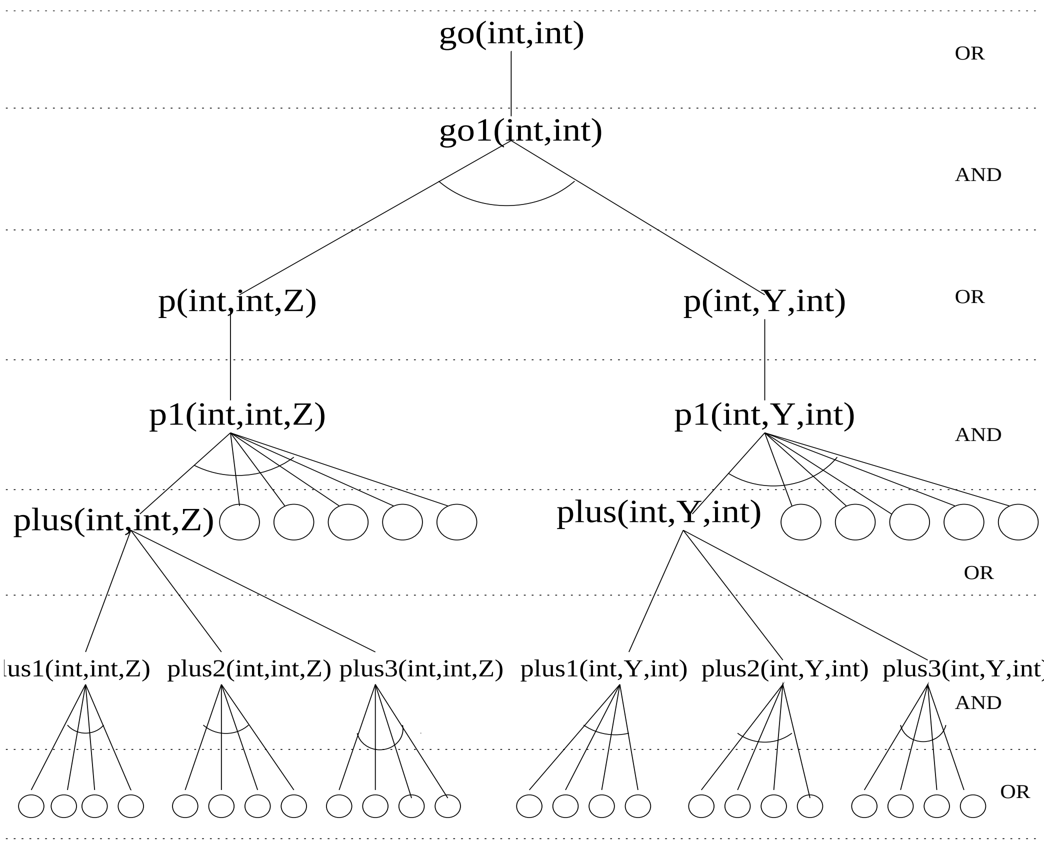

Figure 5.1: Example Analysis Graph

point and creating the paths connecting each version with its corresponding call point(s). The second one is ensuring that not all possible versions are materialized in the specialized program, but rather only the minimal number necessary to perform all the optimizations which are possible. The first problem is addressed in the remainder of this section and the second in Section 5.4.

5.3.1 Analyses with Explicit Construction of the And-Or Graph

As mentioned before, some formulations of top-down abstract interpretation for logic program, such as the original one in Bruynooghe's seminal work [Bru91], are based on explicitly building an abstract version of the resolution tree which contains all possible specialized versions [MWB90, JB92]. This has the advan-

predicate	id	call	success	ancestors
go/2	1	$go(int, int)$	$go(int, int)$	{(query,1)}
p/3	2	$p(int, int, free)$	$p(int, int, int)$	{(go/2/1/1,1)}
p/3	4	$p(int, free, int)$	$p(int, int, int)$	{(go/2/1/2,1)}
plus/3	3	$plus(int, int, free)$	$plus(int, int, int)$	{(p/3/1/1,2)}
plus/3	5	$plus(int, free, int)$	$plus(int, int, int)$	{(p/3/1/1,4)}

Table 5.1: Analysis Memo Table for Example Program

tage that, while not directly represented in the abstract and–or graph, it is quite straightforward to derive a fully specialized program (i.e. with all possible versions) from such graph and the original program. Essentially, a new version is generated for a predicate for each or–node present for that predicate in the abstract graph. Thus, the fully specialized program includes a different, uniquely named version of a predicate for each or–node corresponding to this predicate. Different descendent and–nodes represent different calls in the bodies of the clauses of the specialized predicates. Each call in each clause body in the specialized program is replaced with a call to the unique predicate name corresponding to the successor or–node in the graph for each predicate. We will refer to the program constructed as explained above as the *extended* program. Note that if analysis terminates the number of or-nodes in the graph for each predicate must be finite, and thus the extended program will be finite.

The correctness of this multiply specialized program is given by the correctness of the abstract interpretation procedure, as specialization is simply materializing the (implicit) specialized program from which the analysis has obtained its information.

5.3.2 Tabulation-based Analyses

For efficiency reasons, most practical analyzers [Deb89a, HWD92, MH92, CV94, MS89] do not explicitly build the analysis graph. Instead, a representation of the information corresponding to each program point is kept in a "memo table." Entries in such memo tables typically contain matched pairs of call and success patterns. In most systems some of the graph structure is lost and the data available after analysis (essentially, the memo table) is not quite sufficient for connecting each version with its call point(s). For concreteness, we consider here the case of PLAI [MH92, MH90a]. In the standard implementation of this analyzer, the memo table essentially contains only entries which correspond to or–nodes in the table. And–nodes are also computed and used, but they are not stored. In the following we will refer to the table entries which correspond to or–nodes as *or–records*. Each or–record contains the following information: predicate to which the or–record belongs, call–pattern, abstract success substitution, and a number identifying the or–record itself. This information needs to be augmented with one more field which contains the *ancestor* information, i.e., the point(s) in the multiply specialized program where this or–record (version) is used. By a point we mean a literal within an or–record. It should be noted that the traditional fixpoint algorithm in PLAI had to be modified slightly so that this information is correctly stored, but this modification is straightforward. The version to use in each call can then be determined by the ancestor information and no run-time tests are needed to choose among versions. In the new fixpoint algorithm based on the ideas presented in Chapter 3 recently integrated into PLAI, such ancestor information is already stored in the or–records as it is needed for guiding iterations and thus, in that case, no modification of the fixpoint algorithm is needed at all.

Example 5.3.2 After analysis of the program in Example 5.3.1 the or–records in the memo table for user-defined predicates are shown in Table 5.1.

The first field is the predicate to which the or–record belongs, the second is the number that identifies the or–record, the third and fourth give the call–success pair, and the fifth is the ancestor information. This is a list of pairs (literal, or–record). A literal is identified using the following format: *Predicate/Arity/Clause/Literal.* For example, *go/2/1/2* stands for the second literal in the first clause of predicate *go/2*. If programs are as defined in Section 5.2, this format allows uniquely identifying a literal in a program. [2]

Figure 5.2 represents the ancestor information for each or–record graphically. For clarity, each or–record is represented by its identifier. It is clear that the ancestor information can be interpreted as backward pointers in the analysis

[2]The number of clause $_K$ used in Section 5.2 is here replaced by *Predicate/Arity/Clause.*

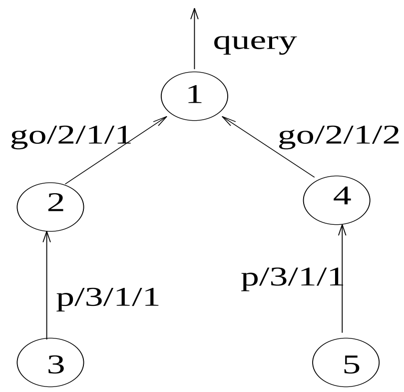

Figure 5.2: Ancestor Information for the Example

graph. These pointers can be followed to determine the version to use in the multiply specialized program. The special literal `query` indicates the starting point of the top-down analysis. PLAI admits any number of starting (entry) points. They are identified by the second number of the pair (query,N).

Finally, the program in which every or–record is implemented in a different version is given in Figure 5.3.

5.4 Minimizing the Number of Versions

The number of versions in the multiply specialized program introduced in Section 5.3 does not depend on the possible optimizations but rather on the number of versions generated during analysis. Even if no benefit is obtained, the program may have more than one version of each predicate. In this section we address the issue of finding the minimal program that allows the same set of optimizations as the fully specialized one. In order to do that we collapse into the same version those or–records that are equivalent.[3] In this way, the set of or–records for each predicate is partitioned into equivalence classes. We now provide an informal description of an algorithm for finding such a program, followed by a more formal description and an algebraic interpretation of the algorithm. The section ends with an example illustrating the execution of the algorithm.

[3]The equivalence relation will be presented more formally in Section 5.4.2.

```
go(A,B) :-
        p1(A,B,_), p2(A,_,B).

p1(X,Y,Z) :-
        plus1(X,Y,Z),
        write(Z), write(' is '),
        write(X), write(' + '), write(Y), nl.
p2(X,Y,Z) :-
        plus2(X,Y,Z),
        write(Z), write(' is '),
        write(X), write(' + '), write(Y), nl.

plus1(X,Y,Z) :-
        Z is X+Y.
plus2(X,Y,Z) :-
        Y is Z-X.
```

Figure 5.3: Extended Program for Example 5.3.1

5.4.1 Informal Description of the algorithm

As mentioned before, the purpose of this algorithm is to minimize the number of versions needed of each predicate while maintaining all the possible optimizations. After analysis and prior to the execution of the minimizing algorithm, we compute the optimizations that would be allowed in each version of the extended program (i.e., the one which generates one version of a predicate for each or–record). A simple but very inefficient way of doing this would be to implement the extended program and let the optimizer run on this program and collect the optimizations performed in each version. These optimizations are represented in the algorithm as finite sets which are associated with the corresponding or–record. The algorithm receives as input the set of table entries (or–records) computed during analysis, augmented with the set of optimizations allowed in each or–record. The output of the algorithm is a partition of the or–records for each predicate into equivalence classes. This information together with the original program is enough to build the final program. For each predicate in the original program as many copies are generated as equivalence classes exist for it. Each of these copies (implementations) receives a unique name. Then, the predicate symbols of the calls to predicates with multiple versions are replaced with the predicate symbols of the corresponding version. This is done using the ancestor information. At the same time, some optimizations can take place in each specialized version.

Not all the information in the or–records is necessary for this algorithm. It is sufficient to use the identifier of the or–record, the ancestor information, and the set of optimizations.

It is important to note that two or–records that allow the same set of optimizations cannot be blindly collapsed since they may use for the same literal (program point) versions of predicates with different optimizations, and thus these two or–records must be kept separate if all possible optimizations are to be maintained. This is why the algorithm consists of two phases. In the first one all the or–records for the same predicate that allow the same set of optimizations are joined in a single version. In the second phase those that use different versions of a predicate for the same literal are split into different versions. Note that each time versions are split it is possible that other versions may also need to be split. This process goes on until no more splitting is needed (a fixpoint is reached). The process always terminates as the number of versions for a predicate is bounded by the number of times the predicate has been analyzed with a different call pattern. Thus, in the worst case we will have as many versions for a predicate as or–records have been generated for it by the analysis, i.e., we will not be able to perform any minimization and we would get back the extended program.

5.4.2 Formalization of the algorithm

In this section some notation is first introduced and then the algorithm and the operations involved are formalized based on such definitions. Also, some of the definitions presented help in understanding the desirable properties multiply specialized programs should have, such as being minimal, of maximal optimization, feasible, etc. At this point, we will not be concerned with termination, which will be discussed in Section 5.4.3. In the following definitions a program is a set of predicates, a predicate a set of versions, and a version a set of or–records. The algorithm is independent of the kind of optimizations being performed. Thus, no definition of optimization is presented. Instead, it is left open for each particular implementation. However, this algorithm requires sets of optimizations for different or–records to be comparable for equality. As an example, in our implementation an optimization is a pair $(literal, value)$, where value is `true` or `fail` (or a list of unifications), generated via abstract executability. The optimization will be materialized in the final program using source to source transformations.

Definition 5.4.1 [Or–record] An *or–record* is a triple $o = \langle N, P, S\rangle$ where N is a natural number that identifies the or–record, P is a set of pairs (literal, number of or–record) and S a set of optimizations.

Definition 5.4.2 [Set of Or–records of a Predicate] The *set of or–records* of a predicate *Pred*, denoted by $\mathcal{O}_{Pred}$, is the set of all the or–records the analyzer has generated for $Pred$.

Definition 5.4.3 [Version of a Predicate] Given a predicate *Pred* v is a version of *Pred* if $v \subseteq \mathcal{O}_{Pred}$.

Definition 5.4.4 [Well Defined Set of Versions] Let $\mathcal{O}_{Pred}$ be the set of or–records of *Pred* and let $\mathcal{V}_{Pred} = \{v_i, i = 1, \ldots, n\}$ be the set of versions for *Pred*. $\mathcal{V}_{Pred}$ is a *well defined set of versions* if

$$\bigcup_{i=1}^{n} v_i = \mathcal{O}_{Pred} \text{ and } v_i \bigcap v_j = \emptyset \;\; i \neq j$$

i.e. $\mathcal{V}_{Pred}$ is a partition of $\mathcal{O}_{Pred}$.

Definition 5.4.5 [Feasible Version] A version $v \in \mathcal{V}_{Pred}$ is *feasible* if it does not use two different versions for the same literal, i.e. if $\forall o_i, o_j \in v \; \forall literal \in Pred:$

$$\left.\begin{array}{r} \exists \mathcal{V}_{Pred'}((\exists v_k \in \mathcal{V}_{Pred'} \exists o_l = \langle N_l, P_l, S_l\rangle \in v_k | (literal, N_i) \in P_l) \wedge \\ (\exists v_m \in \mathcal{V}_{Pred'} \exists o_n = \langle N_n, P_n, S_n\rangle \in v_m | (literal, N_j) \in P_n)) \end{array}\right\} \Longrightarrow k = m$$

Programs with versions that are not feasible cannot be implemented without run-time tests to decide the version to use. Infeasible programs use for the same literal *sometimes* a version and *sometimes* another. This *sometimes* must be determined at run-time. A set of versions is feasible if all the versions in it are feasible.

Definition 5.4.6 [Equivalent Or–records] Two or–records $o_i = \langle N_i, P_i, S_i \rangle, o_j = \langle N_j, P_j, S_j \rangle \in \mathcal{O}_{Pred}$, are *equivalent*, denote by $o_i \equiv_v o_j$, if

$$S_i = S_j \text{ and } \{o_i, o_j\} \text{ is a feasible version.}$$

Definition 5.4.7 [Minimal set of Versions] A set of versions $\mathcal{V}_{Pred}$ is minimal if $\forall o_i, o_j \in \mathcal{O}_{Pred}$

$$o_i \equiv_v o_j \Longrightarrow \exists v_k \in \mathcal{V}_{Pred} \text{ such that } o_i, o_j \in v_k$$

Definition 5.4.8 [Version of Maximal Optimization] A version v is of *maximal optimization* if

$$\forall o_i = \langle N_i, P_i, S_i \rangle, o_j = \langle N_j, P_j, S_j \rangle \in v \;\; S_i = S_j$$

(all the or–records in the version allow the same optimizations). A set of versions is of *maximal optimization* if all the versions in it are of maximal optimization.

Definition 5.4.9 [Optimal Set of versions] A set of versions $\mathcal{V}_{Pred}$ for a predicate *Pred* is *optimal* if it is minimal, of maximal optimization, and feasible, i.e. if

$$\forall o_i, o_j \in \mathcal{O}_{Pred} : \exists v_k \in \mathcal{V}_{Pred}(o_i, o_j \in v_k) \Leftrightarrow o_i \equiv_v o_j$$

We extend these definitions in the obvious way. For example, we say that a program is minimal if the sets of versions for all the predicates in the program are minimal.

According to these definitions, the program before multiple specialization is well defined, feasible, and minimal, but not of maximal optimization in general.

Definition 5.4.10 [$Program_0$] For each predicate *Pred* let $\mathcal{V}_{Pred} = \{v_i | v_i = \{o_i\}\}$. We call this program $Program_0$

The extended program of Section 5.3 corresponds to $Program_0$.

Theorem 5.4.11 $Program_0$ is feasible, of maximal optimization, and well defined.

Definition 5.4.12 [Reunion of Versions] Given two versions $v_i, v_j \in \mathcal{V}_{Pred}$ the *reunion* of v_i and v_j, denoted by $v_i \oplus v_j$, is

$$v_i \oplus v_j = \begin{cases} \{v_i \bigcup v_j\} & \text{if } \forall o_k, o_l \in (v_i \bigcup v_j)\ S_k = S_l \\ \{v_i, v_j\} & \text{otherwise} \end{cases}$$

The new set of versions $\mathcal{V}_{Pred'}$ is $\mathcal{V}_{Pred} - \{v_i, v_j\} \bigcup (v_i \oplus v_j)$.

Theorem 5.4.13 Let P' by a program obtained from P by applying *reunion of versions*. If P is well defined and of maximal optimization then P' is also well defined and of maximal optimization.

Definition 5.4.14 [$Program_i$] $Program_i$ is the program obtained from $Program_0$ by reunion of versions when no more reunions are possible (a fixpoint is reached). $Program_i$ corresponds to the program in which the set of or–records for each predicate is partitioned into equivalence classes using the equality of sets of optimizations as equivalence relation.

Theorem 5.4.15 $Program_i$ is well defined, of maximal optimization, and minimal.

Note that from $Program_0$ to $Program_i$ we have gained minimality, but unfortunately, $Program_i$ will not be feasible in general. This is the purpose of phase 2 of the algorithm.

We now introduce the concept of restriction. It will be used during phase 2 to split versions that are not feasible. It allows expressing in a compact way the fact that several or–records for the same predicate must be in different versions. For example $\{\{1\}, \{2, 3\}, \{4\}\}$ can be interpreted as: or–record 1 must be in a different version than 2, 3, and 4. Also or–records 2 and 3 cannot be in the same version as 4 (2 and 3 can, however, be in the same version).

Definition 5.4.16 [Restriction] Given a set of or–records $\mathcal{O}_{Pred}$, $\mathcal{R}$ is a restriction over $\mathcal{O}_{Pred}$ if $\mathcal{R}$ is a partition of $\mathcal{N}$ and $\mathcal{N} \subseteq \mathcal{O}_{Pred}$.

Definition 5.4.17 [Restriction from a Predicate to a Goal] Let $\mathcal{V}_{Pred} = \{v_1, v_2, \cdots, v_i, \cdots, v_n\}$ be a set versions of the predicate *Pred*, and let *lit* be a literal of the program. The restriction from *Pred* to *lit* is

$$\mathcal{R}_{lit,Pred} = \{r_1, r_2, \cdots, r_i, \cdots, r_n\}$$

where r_i is $\{N|\ \exists o = \langle N, P, S\rangle \in v_i$ such that $(lit, N) \in P\}$[4]

Definition 5.4.18 [A Restriction Holds] A restriction $\mathcal{R}$ holds in a version v if

$$\forall o_i, o_j \in v\, \forall r_k, r_l \in \mathcal{R} : N_i \in r_k \wedge N_j \in r_l \Longrightarrow k = l$$

Definition 5.4.19 [Splitting of Versions by Restrictions] Given a version v and a restriction $\mathcal{R}$, the result of splitting v with respect to $\mathcal{R}$ is written $v \otimes \mathcal{R}$ and is

$$v \otimes \mathcal{R} = \begin{cases} \{v\} & \text{if } v \text{ holds the restriction } \mathcal{R} \\ \{v_1, v_2\} & \text{otherwise} \end{cases}$$

where $v_1 = \{o = \langle N, P, S\rangle|\ o \in v \wedge N \in r_k\}$ and $v_2 = v - v_1$. The new set of versions is $\mathcal{V}_{Pred'} = \mathcal{V}_{Pred} - \{v\} \bigcup (v \otimes \mathcal{R})$.

Example 5.4.20 Consider the splitting of version $\{1, 2, 3, 5\}$ by restriction $\{\{1\}, \{2, 3, 4\}, \{5\}\}$. $\{1, 2, 3, 5\} \otimes \{\{1\}, \{2, 3, 4\}, \{5\}\} = \{\{1\}, \{2, 3, 5\}\}$, but in $\{2, 3, 5\}$ the restriction does not hold yet. $\{2, 3, 5\} \otimes \{\{1\}, \{2, 3, 4\}, \{5\}\} = \{\{2, 3\}, \{5\}\}$. Now the restriction holds. Thus, the initial version is split into 3 versions: $\{\{1\}, \{2, 3\}, \{5\}\}$.

Theorem 5.4.21 Let P' be a program obtained by applying *splitting of versions* to a program P. If P is well defined, of maximal optimization, and minimal then P' is also well defined, of maximal optimization and minimal.

Definition 5.4.22 [$Program_f$] $Program_f$ is the program obtained when all the restrictions hold and no more splitting is needed, i.e., when a fixpoint is reached.

Theorem 5.4.23 [Multiple Specialization Algorithm] $Program_f$ is optimal and well defined.

By Theorem 5.4.21 $Program_f$ is well defined, of maximal optimization, and minimal. We can see that it is also feasible because otherwise there would be a restriction that would not hold. This is in contradiction with the assumption that phase 2 (splitting) has terminated.

[4]Note that r_i may be $\emptyset$.

5.4.3 Structure of the Set of Programs and Termination

As shown above, given a multiply specialized program generated by the analyzer, several different (but equivalent) programs may be obtained. They may differ in size, optimizations, and even feasibility. In this section we discuss the structure of this set of programs and the relations among its elements.

Since $Program_0$ is well defined, applying Theorems 5.4.13 and 5.4.21 we can conclude that all the intermediate programs and the final program are well defined. This means that ill-defined programs are not of interest to us and the set of well defined programs is closed under reunion and splitting of versions. This set of well defined programs forms a complete lattice under the $\sqsubseteq$ operation defined as follows. $P \sqsubseteq P'$ iff $\forall$ predicate $Pred$ let $\mathcal{V}_{Pred}$ (resp. $\mathcal{V}'_{Pred}$) be the set of versions for $Pred$ in P (resp. P') $\forall v \in \mathcal{V}_{Pred}\ \exists v' \in \mathcal{V}'_{Pred}$ s.t. $v \subseteq v'$, i.e., all the versions in P are equal or more specific than the versions in P. The $\perp$ element of such a lattice will be given by the program with most specific versions. This is the program with the greatest number of versions, i.e., the extended program ($Program_0$). The $\top$ element is the program with most general versions, i.e, the one in which all the or–records that correspond to the same predicate are in the same version. This program is the one with the minimum number of versions and is the one obtained when no multiple specialization is done. The reunion operation transforms a program P into another program P' s.t. $P \sqsubseteq P'$ (higher in the lattice). Splitting transforms a P into P' s.t. $P' \sqsubseteq P$ (lower in the lattice).

Note that not all the programs in the lattice are feasible. As said before, a program is not feasible when two or–records in the same version use at the same program point or–records that are in different versions. This is the reason why phase 2 of the algorithm is required. This phase ends as soon as a program is reached that is feasible.

Although not formally stated, the two different operations used during the multiple specialization algorithm (namely reunion and splitting) are operators defined on this lattice since they receive a program as input and produce another program as output. Phase 1 starts with $\perp$ and repeatedly applies $operator_1$ (reunion) moving up in the lattice until we reach a fixpoint. Since the lattice is finite and $operator_1$ is monotonic the termination of phase 1 is guaranteed.

Phase 2 starts with the program that is a fixpoint of $operator_1$ ($Program_i$) and moves down in the lattice. During phase 2 using $operator_2$ (splitting) we

Pred	id	ancestors	optimizations
go/2	1	{(query,1)}	$\emptyset$
p/3	2	{(go/2/1/1,1)}	$\emptyset$
p/3	4	{(go/2/1/2,1)}	$\emptyset$
plus/3	3	{(p/3/1/1,2)}	{(plus/3/3/2,fail), (plus/3/2/1,true), (plus/3/1/2,true)}
plus/3	5	{(p/3/1/1,4)}	{(plus/3/3/2,true), (plus/3/2/1,fail), (plus/3/1/2,fail)}

Figure 5.4: Analysis Table with Optimizations for Example Program

move from an infeasible program to (a less) infeasible program, until we reach a feasible program (which will be the fixpoint). $operator_2$ is also monotonic and thus phase 2 also terminates.

5.4.4 Example

We now apply the minimizing algorithm to the program in Example 5.3.1. As was mentioned before, the algorithm also needs to know the set of possible optimizations in each or–record. We will add this information to the or–record registers. We do not show the call–success values of or–records as they are not needed anymore. Figure 5.4 shows the relevant part of the analysis memo table for the example program, i.e. the or–records, which is the starting point for the multiple specialization algorithm.

We will not go into the details of the set of optimizations, because, as mentioned before, the multiple specialization technique presented is independent of the type of optimizations performed. In any case, the set of optimizations is empty in the or–record for go/2 and in the two or–records for p/3. It has three elements in the or–records for plus/3 that indicate the value that the test `int` will take in execution and have been obtained by means of abstract executability. The only thing to note for now is that the set of optimizations is different in these two or–records for plus/3.

Phase 1 starts with each or–record in a different version ($Program_0$). We represent each or–record only by its identifier:

$Program_0$:

go/2	p/3	plus/3
{{1}}	{{2},{4}}	{{3},{5}}

The two or–records for p/3 have the same optimizations (none) and can be joined. At the end of phase 1 we are in the following situation:

$Program_i$:

go/2	p/3	plus/3
{{1}}	{{2,4}}	{{3},{5}}

Now we execute phase 2. Only plus/3 can produce restrictions. The other two predicates only have one version. The only restriction will be $\mathcal{R}_{p/3/1/1,plus/3} = \{\{2\},\{4\}\}$. The intuition behind this restriction is that or–record number 2 must be in a different version than or–record number 4. The restriction does not hold and thus $\{2,4\} \otimes \{\{2\},\{4\}\} = \{\{2\},\{4\}\}$. Now we must check if this splitting has introduced new restrictions. No new restriction appears because there is no literal that belongs to the ancestor information of both or–record 2 and or–record 4. Thus, the result of the algorithm will be:

$Program_f$:

go/2	p/3	plus/3
{{1}}	{{2},{4}}	{{3},{5}}

The program that the minimization algorithm indicates that should be built coincides in this case with the extended program, which was already depicted in Figure 5.3.

Figure 5.5 shows the lattice for the example program. The node marked with a cross (B) is infeasible. That is why during phase 2 we move down in the lattice and return to $program_0$.

We can use Figure 5.5 to illustrate the definitions introduced in Section 5.4.2. Nodes B and D are of maximal optimization. A and C are not because or–records with different optimizations (3,5) are in the same version. Nodes A, C, and D are feasible. B is not feasible because for the literal p/3/1/1 it uses both or–record

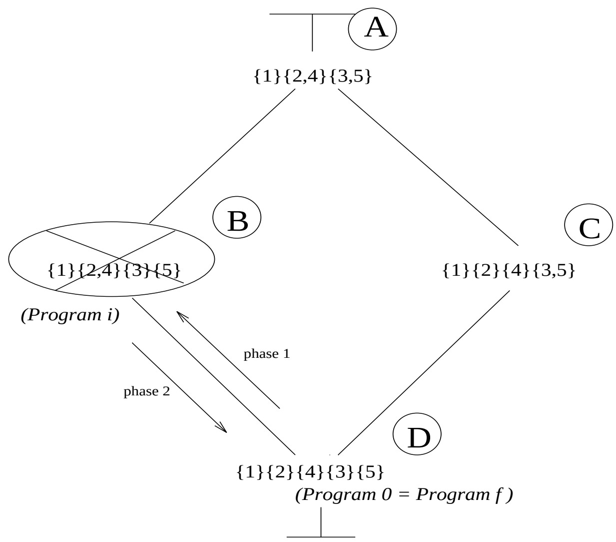

Figure 5.5: Lattice for the Example Program

3 and 5 (we cannot decide at compile-time which one to use). All the nodes in the lattice are minimal. A program is not minimal if two or–records that are equivalent are in different versions. No two or–records are equivalent, thus all the programs in the lattice are minimal.

5.5 The Application: Compile-Time Parallelization

The final aim of parallelism is to achieve the maximum speed (effectiveness) while computing the same solution (correctness) as the sequential execution. The two main types of parallelism which can be exploited in logic programs are well known [Con83, CC94]: or-parallelism and and–parallelism. And-parallelism refers to the parallel execution of the goals in the body of a clause (or, more precisely, of the goals in a resolvent). Several models have been proposed to take advantage of such opportunities (see, for example, [CC94] and it references).

Guaranteeing correctness and efficiency in and–parallelism is complicated by the fact that dependencies may exist among the goals to be executed in parallel, due to the presence of shared variables at run–time. It turns out that when these dependencies are present, arbitrary exploitation of and–parallelism does not guarantee efficiency. Furthermore, if certain impure predicates that are relatively common in Prolog programs are used, even correctness cannot be guaranteed.

However, if only *independent goals* are executed in parallel, both correctness and efficiency can be ensured [Con83, HR95]. Thus, the dependencies among the different goals must be determined, and there is a related parallelization overhead involved. It is vital that such overhead remain reasonable. Herein we follow the approach proposed initially in [WHD88, HWD92] (see their references for alternative approaches) which combines local analysis and run–time checking with a data-flow analysis based on abstract interpretation [CC77]. This combination of techniques has been shown to be quite useful in practice [WHD88, MJMB89, VD90, Tay90, dMSC93].

5.5.1 The Annotation Process and Run-time Tests

In the &-Prolog system, the automatic parallelization process is performed as follows [BGH94a]. Firstly, if required by the user, the Prolog program is analyzed using one or more global analyzers. These analyzers are aimed at inferring useful information for detecting independence. These analyses use the optimized fixpoint algorithm presented in Chapter 3. Secondly, since side–effects cannot be allowed to execute freely in parallel, the original program is analyzed using the global analyzer described in [MH89a] which propagates the side–effect characteristics of builtins determining the scope of side–effects. In the current implementation, side-effecting literals are not parallelized. Finally, the *annotators* perform a source–to–source transformation of the program in which each clause is annotated with parallel expressions and conditions which encode the notion of independence used. In doing this they use the information provided by the global analyzers mentioned before.

The annotation process is divided into three subtasks. The first one is concerned with identifying the dependencies between each two literals in a clause and generating the conditions which ensure their independence. The second task aims at simplifying such conditions by means of the information inferred by the local or global analyzers. In other words, transforming the conditions into the minimum number of tests which, when evaluated at run–time, ensure the independence of the goals involved. Finally, the third task is concerned with the core

```
:-module(mmatrix,[mmultiply/3]).

mmultiply([],_,[]).
mmultiply([V0|Rest], V1, [Result|Others]):-
        multiply(V1,V0,Result), mmultiply(Rest, V1, Others).

multiply([],_,[]).
multiply([V0|Rest], V1, [Result|Others]):-
        vmul(V0,V1,Result), multiply(Rest, V1, Others).

vmul([],[],0).
vmul([H1|T1], [H2|T2], Result):-
        Product is H1*H2, vmul(T1,T2, Newresult),
        Result is Product+Newresult.
```

Figure 5.6: mmatrix.pl

of the annotation process [BGH94a, MH90b], namely the application of a particular strategy to obtain an optimal (under such a strategy) parallel expression among all the possibilities detected in the previous step.

5.5.2 An Example: Matrix Multiplication

We illustrate the process of automatic program parallelization with an example. Figure 5.6 shows the code of a Prolog program for matrix multiplication. The declaration `:-module(mmatrix,[mmultiply/3]).` is used by the (goal-oriented) analyzer to determine that the only predicate which may appear in top-level queries is *mmatrix/3*. No information is given about the arguments in calls to the predicate *mmatrix/3*. This could be done using one or more *entry* declarations (Chapter 4). If for example we want to specialize the program for the case in which the first two arguments of *mmatrix/3* are ground values and we inform the analyzer about this, the program would be parallelized without the need for any run-time tests. However, for the purposes of studying multiple specialization, we will con-

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

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

Figure 5.7: Parallel mmatrix

sider the case in which no information at all is provided by the user regarding calling patterns, beyond the exported predicate information present in the module declaration. In this case the analyzer must in principle assume no knowledge regarding the instantiation state of the arguments at the module entry points.

Figure 5.7 contains the result of automatic parallelization under these assumptions. `if-then-elses` are written `(cond -> then ; else)`, i.e., using standard Prolog syntax. The `&` signs between goals indicate that they can be executed in parallel. The predicate *vmul/3* does not appear in Figure 5.7 because automatic parallelization has not detected any profitable parallelism in it (due to granularity control) and its code remains the same as in the original program.

It is clear from Figure 5.7 that a good number of run-time tests have been introduced in the parallelization process. These tests are necessary to determine independence at run-time, given that nothing is known about the input arguments. If the tests hold the parallel code is executed. Otherwise the original sequential code is executed. As usual, `ground(X)` succeeds if `X` contains no variables. `indep(X,Y)` succeeds if `X` and `Y` have no variables in common. For conciseness and efficiency, a series of tests `indep(X1,X2), ..., indep(Xn-1,Xn)` is written as `indep([[X1,X2], ..., [Xn-1,Xn]])`.

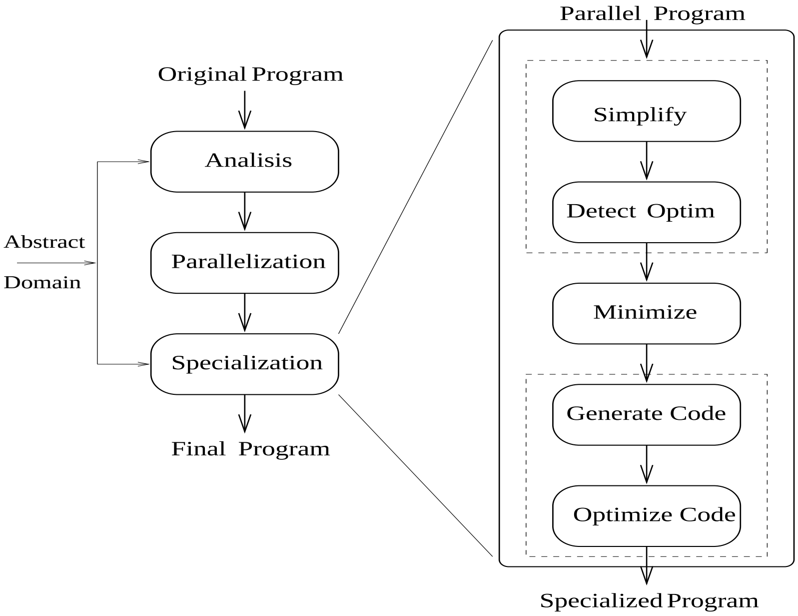

Figure 5.8: Program Annotation and Abstract Multiple Specialization

Even though groundness and independence tests are executed by efficient builtin predicates in the &-Prolog system, these tests may still cause considerable overhead in run-time performance, to the point of not even knowing at first sight if the parallelized program will offer speedup, i.e., if it will run faster than the sequential one. Our purpose is to study whether multiple specialization can be used to reduce the run-time test overhead and to increase speedups.

5.6 Multiple Specialization in the &-Prolog Compiler

Figure 5.8 (picture on the left) presents the role of abstract multiple specialization in the &-Prolog system. As stated in the previous section, automatic parallelization may introduce run-time tests and conditionals if the information available does not allow determining the dependence/independence of literals statically. As mentioned before, it is this checking overhead that the multiple specialization which has been added to the &-Prolog compiler and is the subject of our performance study is aimed at reducing. Note that because of the way the parallelization process is performed, if the same abstract domain is used to provide information to both the parallelization and specialization phases, none of the run-time tests introduced is superfluous and thus none of them can be eliminated by the specializer unless multiple specialization is performed.

Even though not depicted in Figure 5.8, analysis information is not directly available at all program points after automatic parallelization, because the process modifies certain parts of the program originally analyzed. However, the &-Prolog system uses incremental analysis techniques to efficiently obtain updated analysis information from the one generated for the original program (see Chapter 2).

Conceptually, the process of abstract multiple specialization is composed of five steps, which are shown in Figure 5.8 (picture on the right). In the first step (*simplify*) the program optimizations based on abstract execution are performed whenever possible. This saves having to optimize the different versions of a predicate when the optimization is applicable to all versions. Any optimization that is common to all versions of a predicate is performed at this stage. The output is a monovariant abstractly specialized program. This is also the final program if multiple specialization is not performed. The remaining four steps are related to *multiple* specialization.

In the second step (*detect optimizations*) information from the multi-variant abstract interpretation is used to detect (but not to perform) the optimizations allowed in each of the (possibly) multiple versions generated for each predicate during analysis. Note that only one step of analysis is required in our system in order to both compute the set of or–records for each predicate and the optimizations allowed for each one of them. This is only possible if we can identify the abstract substitutions for the different or–records at each program point. In our analyzer this is done by just storing the or–record identifiers along with each substitution generated by polyvariant analysis. Even though the addition of this identifier to abstract substitutions may seem an overhead, they will be used as detailed dependencies while computing the analysis graph. This will allow analysis to be more efficient, as studied in Chapter 3, i.e., to converge faster to a fixpoint, and to be incremental (see Chapter 2).

Note that the source for the multiply specialized program has not been generated yet (this will be done in the fourth step, *generate code*) but rather the code generated in the first step is used, considering several abstract substitutions for each program point instead of their least upper bound, as is done in the first step. The output of this step is the set of literals that become abstractly executable (and their value) in each version of a predicate due to multiple specialization. Note that these literals are not abstractly executable without multiple specialization, otherwise the optimization would have already been performed in the first step.

The third step (*minimize*) is concerned with reducing the size of the multiply specialized program as much as possible, while maintaining all possible optimizations and without the need for introducing run-time tests to select among different versions of a predicate. A detailed presentation of the algorithm used in this step and its evaluation is the subject of [PH95].

In the fourth step (*generate code*) the source code of the minimal multiply specialized program is generated. The result of the minimization algorithm in the previous step indicates the number of implementations needed for each predicate. Each of them receives a unique name. Also, literals must also be renamed appropriately for a predicate with several implementations.

In the fifth step (*optimize code*), the particular optimizations associated with each implementation of a predicate are performed. Other simple program optimizations like eliminating literals in a clause to the right of a literal abstractly executable to false, eliminating a literal which is abstractly executable to true from the clause it belongs instead of introducing the builtin `true/1`, dead code elimination, etc. are also performed in this step.

In the implementation, for the sake of efficiency, the first and second steps, and the fourth and fifth are performed in one pass (this is marked in Figure 5.8 by dashed squares), thus reducing to two the number of passes through the source code. The third step is not performed on source code but rather on a synthetic representation of sets of optimizations and versions. The core of the multiple specialization technique (steps *minimize* and *generate code*) is independent of the actual optimizations being performed.

The abstract specializer is parametric with respect to the abstract domain used. Currently, the specializer can work with all the abstract domains implemented in the analyzer in the &-Prolog system. In order to augment the specializer to use the information provided by a new abstract domain

Domain	$TS_\alpha(gr(X_1))$	$FF_\alpha(gr(X_1))$	$TS_\alpha(ind(X_1))$	$FF_\alpha(ind(X_1))$
sharing	O	N	O	N
sh+fr	O	S	O	S
asub	O	N	O	N

Table 5.2: Optimality of Different Domains

(D_α), correct $A_{TS}(\overline{B}, D_\alpha)$[5] and $A_{FF}(\overline{B}, D_\alpha)$ sets must be provided to the analyzer for each builtin predicate B whose optimization is of interest. Alternatively, and for efficiency issues, the specializer allows replacing the conditions in Definition 5.2. 5.2.13 with specialized ones because in $\exists \lambda' \in A_{TS}(\overline{B}, D_\alpha)$: $call\_to\_entry(L, \overline{B}, D_\alpha, \lambda) \sqcup \lambda' = \lambda'$ all values are known before specialization time except for λ which will be computed by analysis. I.e., conditions can be partially evaluated with respect to D_α, $\overline{B}$ and a set of λ', as they are known in advance.

Table 5.2 shows the accuracy of a number of abstract domains (*sharing* [JL92, MH92], *sharing+freeness* (sh+fr) [MH91], and *asub* [Son86, CDY91]) present in the &-Prolog system with respect to the run-time tests (i.e., `ground/1`, `indep/1`). For the three of them the sets TS_α and FF_α are computable and we can take $A_{TS} = M_{\sqsubseteq}(TS_\alpha(\overline{B}, D_\alpha))$ and $A_{FF} = M_{\sqsubseteq}(FF_\alpha(\overline{B}, D_\alpha))$. O stands for optimal, S stands for approximate, and N stands for none, i.e. $FF_\alpha(\overline{B}, D_\alpha) = \{\perp\}$. The three of them are optimal for abstractly executing both types of tests to true. However, only sharing+freeness (sh+fr) allows abstractly executing these tests to false, even though not in an optimal way.

Example 5.6.1 The resulting program after abstract multiple specialization is performed is shown in Figure 5.9. The program generated in our implementation is equivalent to the one presented except that internal names are used for specialized versions to avoid clashes with other user defined predicates. Two versions have been generated for the predicate *mmultiply/3* and four for the predicate *multiply/3*. They all have unique names, and literals have been renamed appropriately to avoid having to use run-time tests for deciding the right version to use. As in Figure 5.7, the predicate *vmul/3* is not presented in the figure because

[5] See Section 5.2.

```
mmultiply([],_,[]).
mmultiply([V0|Rest],V1,[Result|Others]) :-
        (ground(V1),
         indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             multiply1(V1,V0,Result) & mmultiply1(Rest,V1,Others)
        ;    multiply2(V1,V0,Result), mmultiply(Rest,V1,Others)).
mmultiply1([],_,[]).
mmultiply1([V0|Rest],V1,[Result|Others]) :-
        (indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             multiply1(V1,V0,Result) & mmultiply1(Rest,V1,Others)
        ;    multiply1(V1,V0,Result), mmultiply1(Rest,V1,Others)).

multiply1([],_,[]).
multiply1([V0|Rest],V1,[Result|Others]) :-
        (ground(V1), indep([[Result,Others]]) ->
             vmul(V0,V1,Result) & multiply3(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply1(Rest,V1,Others)).
multiply2([],_,[]).
multiply2([V0|Rest],V1,[Result|Others]) :-
        (ground(V1),
         indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             vmul(V0,V1,Result) & multiply4(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply2(Rest,V1,Others)).
multiply3([],_,[]).
multiply3([V0|Rest],V1,[Result|Others]) :-
        (indep([[Result,Others]]) ->
             vmul(V0,V1,Result) & multiply3(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply3(Rest,V1,Others)).
multiply4([],_,[]).
multiply4([V0|Rest],V1,[Result|Others]) :-
        (indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]]) ->
             vmul(V0,V1,Result) & multiply4(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply4(Rest,V1,Others)).
```

Figure 5.9: Specialized mmatrix

its code is identical to the one in the original program in Figure 5.6 (and the parallelized program). Only one version has been generated for this predicate even

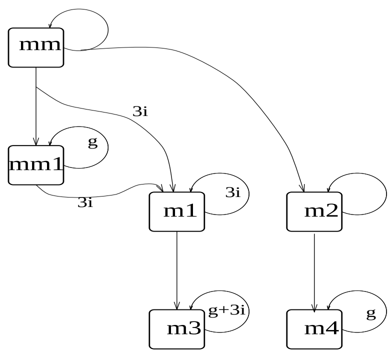

Figure 5.10: Call Graph of the Specialized Program

though multi-variant abstract interpretation generated eight different variants for it. As no further optimization is possible by implementing several versions of *vmul/3*, the minimization algorithm has collapsed all the different versions of this predicate into one.

It is important to mention that abstract multiple specialization is able to automatically detect and extract some invariants in recursive loops: once a certain run-time test has succeeded it does not need to be checked in the following recursive calls [GH91]. Figure 5.10 shows the call graph of the specialized program of Figure 5.9. `mm` stands for mmultiply and `m` for multiply. Edges are labeled with the number of tests which are avoided in each call to the corresponding version with respect to the non specialized program. For example, `g+3i` means that each execution of this specialized version avoids a groundness and three independence tests. It can be seen in the figure that once the groundness test in any of `mm`, `m1`, or `m2` succeeds, it is detected as an invariant, and the more optimized versions `mm1`, `m3`, and `m4` respectively will be used in all remaining iterations.

5.7 Experimental Results

In this section we present a series of experimental results. The primary aim of these experiments is to assess whether performing abstract multiple specialization for improving automatically parallelized programs is profitable or not. This assessment will be realized by studying some of the cost/benefit tradeoffs involved in multiple specialization, in terms of time and space. Even though the results have been obtained in the context of a particular implementation and type of optimizations, we believe that it is possible to derive some conclusions from the results regarding the cost and benefits of multiple specialization in general.

The benchmarks considered have been automatically parallelized using the *mel* [MH90b] heuristic algorithm for the generation of parallel expressions and the $sharing + freeness$ abstract domain [MH91] to introduce as few run-time tests as possible. Such combination of techniques has been experimentally shown [BGH94b] to be capable of effectively parallelizing logic programs with quite reasonable run-time overhead for checking independence, producing useful speedups in parallel execution. However, in those experiments the availability of a reasonable description of the instantiation state of the arguments of exported predicates (i.e., predicates accessible from outside the module being analyzed) was assumed.

In the current set of experiments, as in the experiment for local change in Chapter 2, we study what is a very unfavorable situation for automatic parallelization: the case in which no information is provided to the analyzer regarding the possible input values. This situation is interesting in that it appears when modules written by naïve users are compiled in isolation. In this case the analysis has to be performed with only the entry points to the programs given in the module declarations as input data regarding external call patterns. Since as a result of this the analyzer will sometimes have incomplete information, a large number of run-time tests will in some cases be included in the resulting programs, which are then potential targets for multiple specialization. The relatively wide set of benchmarks considered is the subset of the benchmarks used in Chapters 2 and 3 for (incremental) program analysis[6] which cannot be parallelized without the need of run-time tests in the unfavorable case presented above. The other benchmarks (fib, qsortapp, tak, and witt) are parallelized without run-time tests even in this case and are therefore not studied in this work.

5.7.1 The Cost of Multiple Specialization

In order to assess the cost of specialization in terms of compilation time, Table 5.3 compares the analysis, parallelization, and specialization times for each bench-

[6] And previously in [BGH94b] for automatic parallelization.

Bench	Ana	Par	ReA	Spec	Total	SD
aiakl	1.40	1.59	0.36	0.06	3.41	1.14
ann	3.24	2.11	2.01	0.90	8.26	1.54
bid	0.34	0.20	0.24	0.23	1.00	1.88
boyer	0.97	0.27	0.37	0.32	1.94	1.56
browse	0.17	0.11	0.26	0.25	0.78	2.82
deriv	0.18	0.20	0.42	0.13	0.93	2.46
hanoiapp	0.28	0.22	0.14	0.04	0.69	1.37
mmatrix	0.13	0.08	0.17	0.08	0.45	2.20
occur	0.12	0.06	0.18	0.08	0.43	2.38
progeom	0.09	0.06	0.03	0.05	0.22	1.53
qplan	0.74	1.23	0.15	0.46	2.58	1.31
query	0.04	0.04	0.04	0.06	0.19	2.26
read	7.26	2.39	0.02	0.63	10.31	1.07
serialize	0.26	0.18	0.03	0.06	0.52	1.19
warplan	1.21	0.21	0.11	0.26	1.79	1.26
zebra	2.27	17.25	2.02	0.06	21.59	1.11
					Overall	1.23

Table 5.3: Specialization and Parallelization Times (Using No Call Pattern Info)

mark. We argue that it is reasonable to compare these times as the programs that accomplish those tasks are implemented using the same technology, are integrated in the same system, they share many data structures, and work with the same input program (or slightly modified versions of it). Times are in seconds on a Sparc 1000. **Ana** is the time taken to analyze the original program, **Par** is the parallelization time, **ReA** is the reanalysis time required to update analysis information after parallelization in an incremental way, using the algorithm for local change described in Chapter 2, and **Spec** is the multiple specialization time which includes computing the possible optimizations in each version using the notion of abstract executability, minimizing the number of versions, and materializing the new program in which the new versions are optimized (using source to source transformations). The time required for automatic parallelization is

Bench	**Pr**	**Max**	**min**	**Ind**	**M(%)**	**m(%)**	**I(%)**	**Ratio**
aiakl	9	4	0	0	44	0	0	1.44
ann	77	70	29	16	90	37	21	1.39
bid	22	39	9	4	177	40	18	1.97
boyer	27	57	9	7	211	33	26	2.33
browse	9	19	15	7	211	166	78	1.17
deriv	5	5	5	1	100	100	20	1.00
hanoiapp	3	10	2	1	333	66	33	2.60
mmatrix	3	11	4	0	366	133	0	2.00
occur	5	15	7	3	300	140	60	1.67
progeom	10	5	0	0	50	0	0	1.50
qplan	48	17	6	4	35	12	8	1.20
query	6	1	0	0	16	0	0	1.17
read	25	52	0	0	208	0	0	3.08
serialize	6	3	0	0	50	0	0	1.50
warplan	37	130	42	29	351	113	78	2.11
zebra	7	10	0	0	142	0	0	2.43
Overall					147	43	24	1.73
Relative Overall					208	80	33	1.72

Table 5.4: Number of Versions

the sum of **Ana** and **Par**. The cost of multiple specialization should be viewed as **ReA** plus **Spec** as specialization requires analysis information to be up to date. **Total** gives the total time required for the whole process. The last column, **SD** is the slow-down introduced by multiple specialization in the parallelization process and is computed as **Total**/(**Ana**+**Par**). Finally, **Overall** gives the slow-down obtained by taking for each column the sum of times for all benchmarks. The results can be interpreted as indicating that performing multiple specialization after parallelization slows down the compilation process over all benchmarks approximately by a factor of 1.23.

It appears that the time required for multiple specialization, at least in this application, is reasonable. However, a potentially greater concern in multiple specialization than compilation time can be the increase in program size. Table 5.4

shows a series of measurements relevant to this issue. **Pred** is the number of predicates in the original program. **Max** is the number of additional (versions of) predicates that would be introduced if the minimization algorithm were not applied (when adding it to **Pred** this is also the number of versions that the analyzer implicitly uses internally during analysis). **Min** is the number of additional versions if the minimization algorithm is applied. As mentioned before, sometimes, in order to achieve an optimization some additional versions have to be created just to create a "path" to another specialized version, i.e. to make the program feasible (using the terminology of Section 5.2). The impact of this is measured by **Ind** which represents the number of such "Indirect" versions in the minimized program that have been included during phase 2 of the algorithm. I.e., this is the number of versions which have the same set of optimizations as an already existing version for that predicate.

We observe that for some benchmarks **Min** is 0. This means that multiple specialization has not been able to optimize the benchmark any further. That is, the final program equals the original program. However, note that if we did not minimize the number of versions the program size would be increased even though no additional optimization is achieved. **M(%)** is computed as $\frac{Max}{Preds} \times 100$. **M(%)** and **I(%)** are computed similarly but replacing **Max** by **Min** and **Ind** in the formula respectively. Finally **Ratio** is the relation between the sizes (in number of predicates) of the multiply specialized programs with and without minimization. The last rows of Table 5.4 show two different overall figures. The first is computed considering all the benchmark programs and the second considering only the programs in which the specialization method has obtained some optimization (**Min**> 0).

According to the overall figures, the specialized program has 43% additional versions with respect to the original program. However, this average greatly depends on the number of possible optimization points in the original program (in our case run-time tests) and cannot be taken as a general result. Of much more relevance are the ratios between **M(%)** and **N(%)**, and between **I(%)** and **M(%)**, which are in some ways independent of the number of possible optimizations in the program. This is supported by the relative independence of the ratios from the benchmarks. The first ratio measures the effectiveness of the minimization algorithm. This ratio is 3.41 or 2.6 using global or relative averages

Bench	**Orig**	**Par**	**Spec**	**P/O**	**S/O**	**S/P**
aiakl	3317	4667	4386	1.41	1.32	0.94
ann	43368	55402	66776	1.28	1.54	1.21
bid	10242	14159	17031	1.38	1.66	1.20
boyer	37340	38273	43030	1.02	1.15	1.12
browse	3460	5977	11013	1.73	3.18	1.84
deriv	2747	5957	10299	2.17	3.75	1.73
hanoiapp	1115	2120	3014	1.90	2.70	1.42
mmatrix	1257	3048	5802	2.42	4.62	1.90
occur	2093	3270	6377	1.56	3.05	1.95
progeom	3510	4334	4174	1.23	1.19	0.96
qplan	35155	36679	38501	1.04	1.10	1.05
query	7313	8816	8563	1.21	1.17	0.97
read	23147	23718	23556	1.02	1.02	0.99
serialize	2994	3749	3622	1.25	1.21	0.97
warplan	22788	23047	19922	1.01	0.87	0.86
zebra	3645	4912	4842	1.35	1.33	0.99
			Overall	1.17	1.33	1.14
			Relative Overall	1.18	1.39	1.18

Table 5.5: Size of Programs

respectively. I.e., the minimizing algorithm is able to reduce to a third the number of additional versions needed by multiple specialization. The second ratio represents how many of the additional versions are indirect. It is 56% or 41% (Global or Relative). This means that half of the additional versions are due to indirect optimizations. Another way to look at this result is as meaning that on the average there is one intermediate, indirect predicate between an originating call to an optimized, multiply specialized predicate and the actual predicate. It seems that this can in many cases be an acceptable cost in return for no run-time overhead in version selection.

Another pragmatic and very significant way of comparing the cost in program size incurred by multiple specialization is by comparing the size of the compiled programs (in bytecode quick-load format) before and after multiple specialization. For reference, we also compare to the size of the byte code for the original program. Table 5.5 presents the size in bytes of the original (**Orig**), parallelized (**Par**), and specialized (**Spec**) programs in bytes for &-Prolog . **P/O** gives the increase in size due to parallelization, and **S/O** the increase due to the composition of specialization and parallelization with respect to the original program. **S/P** presents the cost in space incurred by multiple specialization alone. As in Table 5.4, two cases have been considered for computing the overall space cost of multiple specialization: **Overall**, in which all benchmarks are considered, and **Relative Overall** in which only those benchmarks which benefit from multiple specialization are considered. The results can be interpreted as indicating that, in our system, multiple specialization increases program size by a ratio of 1.14 or 1.18 (relative). This increase is very similar to that introduced by parallelization (1.17 – 1.18) in the set of benchmarks considered. Finally, when multiple specialization and parallelization are composed, the overall increase in program size is around 1/3 even in the unfavorable case studied of not giving any information to the analyzer regarding the instantiations of the input arguments of exported predicates.

Note that the cost in program size for multiple specialization presented in Table 5.5 is better than that presented in Table 5.4. There are several reasons for this. First, the specializer performs some degree of dead-code elimination. Second, abstract executability allows in many cases performing source to source transformations which shorten the program, e.g., by simplifying a conditional, eliminating one of the branches in an if–then–else, etc. Third, because the number of additional versions is not necessarily a good estimate of program size as this will greatly depend on the size of the predicates which are being replicated.

5.7.2 Benefits of Multiple Specialization

Having discussed the cost of multiple specialization in automatic parallelization both in terms of time and space, we now measure experimentally the benefits introduced by multiple specialization.

The addition of run-time tests and conditionals in parallelized programs will introduce some overhead which can be seen as an additional amount of work to be performed at run-time. On the other hand, this additional work will allow

Bench	Orig/Par	Orig/Spec	Par/Spec	Improv(%)
ann	0.68	0.69	1.01	4
bid	0.63	0.71	1.11	28
boyer	0.82	0.85	1.03	18
browse	0.64	53.54	84.03	—
brow_nf	0.86	0.89	1.04	27
deriv	0.19	0.21	1.11	12
hanoiapp	0.60	0.75	1.26	51
mmatrix	0.43	0.86	2.02	88
occur	0.84	1.01	1.21	107
qplan	0.97	0.99	1.02	70
warplan	1.00	1.00	1.00	—

Table 5.6: Sequential Performance

performing different tasks at the same time, i.e., in parallel. If several workers (processors) are available, the overall execution time of the parallelized program will hopefully be less than that of original program. Table 5.6 shows the slow-downs with respect to the original program of the parallel (**Par**) and specialized (**Spec**) programs. The main contribution of multiple specialization in program parallelization will be in reducing the overhead of run-time tests and conditionals further, i.e., by getting a high value for **Orig/Spec**. This value will be 1 when the overhead of run-time tests has been completely eliminated and will not be much higher than 1 if the original program was optimally written, i.e., by an experienced programmer. Note that programs which are parallelized without any run-time tests already have 1 as **Orig/Par**. Thus, they are not considered in Table 5.6 as they contain no test which can be eliminated by multiple specialization.

The **Par/Spec** column provides the sequential speedup achieved due to multiple specialization. It is always greater than 1, i.e., no slow-downs are introduced. Speedups range from a small 1.01 for ann to 2.02 for mmatrix.

In the case of browse, the original benchmark contains the clause:

```
p_match([P|Patterns],D) :-
        (match(D,P), fail; true),
```

```
            p_match(Patterns, D).
```

where match(D,P) produces no side-effects. The specializer transforms this clause into:

```
p_match([P|Patterns],D) :-
            p_match(Patterns, D).
```

and all the work performed in the calls to match/2 is eliminated from the execution. In order to isolate the effects of multiple specialization from these optimizations (which can be performed without generating different versions of the predicate) we have studied instead a modified version of the benchmark, *brow_nf*, which is obtained by removing the call to fail after match(D,P) in the original benchmark and eliminating the clause

```
property([],_X,_Y) :- fail.
```

which is also eliminated automatically by the specializer.

Finally, column **Improv(%)** is computed for each benchmark as $\frac{Spec-Par}{1-Par} \times 100$ and gives an idea of the degree to which multiple specialization in the &-Prolog system has accomplished its primary task, i.e., eliminating the overhead introduced by the run-time tests and conditionals as much as possible. Note that this figure makes no sense for *browse.pl* as the improvement is much beyond the overhead of run-time tests, and is thus not presented. This figure is not given for *warplan* either since the overhead introduced by the run-time tests is insignificant.

Another interesting question is how the improvement in sequential execution time, i.e., the reduction of total work to be performed, affects performance in parallel execution, which is of course the ultimate objective of the parallelizing compiler. With this aim we compare the execution speed of the original program with the parallelized (**P**) and specialized (**S**) programs and show the results in Table 5.7. The improvement in execution speed, **P/S**, is given by column **I**. This is done for three different cases. In the first one five processors are available and dedicated to the execution of the program. In the second, ten processors are used, and in the third an unlimited number of processor can be used, i.e., it gives an estimate of the best possible parallel performance. These three cases are distinguished by the subindex $_5$, $_{10}$ and $_\infty$ respectively. Additionally, in the columns

Bench	$\mathbf{P}_5$	$\mathbf{S}_5$	$\mathbf{I}_5$	$\mathbf{P}_{10}$	$\mathbf{S}_{10}$	$\mathbf{I}_{10}$	$\mathbf{P}_{\#p}$	$\mathbf{S}_{\#p}$	$\mathbf{I}_\infty$
ann	2.40	2.39	1.00	3.34	3.59	1.07	4.30_{53}	4.33_{52}	1.01
bid	1.13	1.27	1.12	1.13	1.27	1.12	1.13_{9}	1.27_{9}	1.13
boyer	0.82	0.85	1.04	0.82	0.85	1.04	0.82_{7}	0.85_{7}	1.03
brow_nf	1.85	1.89	1.02	2.03	2.07	1.02	2.12_{124}	2.17_{130}	1.02
deriv	0.79	0.86	1.09	1.20	1.24	1.03	1.36_{175}	1.38_{166}	1.02
hanoi	0.89	1.18	1.33	0.89	1.18	1.33	0.82_{34}	1.10_{6}	1.33
mmat	1.94	3.94	2.03	3.56	7.33	2.06	5.03_{47}	15.01_{56}	2.98
occur	3.96	4.75	1.20	6.34	8.84	1.39	9.85_{34}	28.29_{108}	2.87
qplan	1.31	1.35	1.03	1.31	1.35	1.03	1.31_{4}	1.35_{3}	1.03
warplan	1.07	1.07	1.00	1.07	1.07	1.00	1.07_{5}	1.07_{5}	1.00

Table 5.7: Parallel Performance

$\mathbf{P}_{\#p}$ and $_{\#p}$, an upper bound on the number of processor required to achieve such optimal speed for each benchmark is given as a subindex. These speedup figures have been obtained with the IDRA simulation tool [FCH92]. This tool allows obtaining speedup results which have been shown to match closely the actual speedups obtained in the &-Prolog system for the number of processors available for comparison. It is also believed that the results obtained are good approximations of the best possible parallel execution for larger numbers of processors [FCH92]. This approach allows concentrating on the available parallelism, without the limitations imposed by a fixed number of physical processors, a particular scheduling, bus bandwidth, etc. IDRA takes as input an execution trace file generated from the execution of a parallelized program on one or more processors and the time taken by the sequential program, and computes the achievable speedup for any number of processors. The trace files list the events occurred during the execution of the parallel program, such as a parallel goal being started or finished, and the times at which the events occurred. Since &-Prolog normally generates all possible parallel tasks in a parallel program, regardless of the number of processors in the system, information is gathered for all possible goals that would be executed in parallel. Using this data, IDRA builds a task dependency graph whose edges are annotated with the exact execution times. The possible actual execution graphs (which could be obtained if more processors were available) are constructed from this data and their total execution times compared to the sequential time, thus making quite accurate estimations of (ideal — in the sense that some low level overheads are not taken into account) speedups.

5.8 Discussion

The experimental results presented in Section 5.7 allow us to conclude that, at least in the application considered, abstract multiple specialization is a useful technique: its costs are reasonable and the benefits of sufficient significance. Summarizing the results in terms of compilation time, the additional time required for specialization is about 1/4 of the parallelization time. Regarding the size of the specialized program, it is about 1/6 larger than the parallelized one and about 1/3 larger than the original one. Regarding the actual benefits of multiple specialization in terms of speedup, it varies greatly from one benchmark to another. Thus, it is not easy to give a factor which summarizes the achievable speedup, but many programs do obtain useful speedups. Note also that if our primary aim when performing multiple specialization is, as in the experiments, to reduce the overhead introduced by independence run-time tests, in the relatively frequent case in which automatic parallelization does not require the introduction of any run-time test, specialization can be easily turned off and only applied to those cases which are problematic to automatic parallelization.

If the particular optimizations being considered are appropriate, multiple specialization always generates programs which are, at least theoretically, more optimized than the original. This is confirmed by column **Par/Spec** of Table 5.6, which for all benchmarks presents values greater than 1. Leaving the atypical case of browse.pl aside, the results show that the sequential improvement is low for some benchmarks (ann, qplan, warplan), significant in others (bid, hanoi, occur), and very important in others (mmatrix). This program (Figure 5.6), is a good candidate for parallelization and its execution time decreases nearly linearly with the number of processors. Note, however, that if the user provides enough information regarding the input, this program would be parallelized in the &-Prolog compiler without any run-time tests. However, if no information is provided by the user (the case studied) many such tests are generated and performance decreases. The reason for obtaining such improved speedups for mmatrix when multiple specialization is used is that it is a recursive program in which specialization automatically detects and extracts an invariant, as explained in Example 5.6.1.

Another important conclusion which the experiments seem to bear is that the speedup achieved by multiple specialization generally increases with the number of processors, thus making multiple specialization quite relevant in the context of a parallelizing compiler. The reason for this is that, in general, specialization reduces the overhead of parallelization but does not deeply transform the structure of tasks to be performed: the length of some tasks will be shortened due to the elimination of run-time tests. This is the case for most benchmarks studied. The main exception is `deriv.pl`, which is a program for symbolic differentiation and also a good candidate for parallelization. However, the improvement obtained with specialization is 1.11 for one processor and it decreases to a low 1.02 with 130 processors. This shows that not all programs with significant parallelism are good candidates for specialization.

Another interesting case is `occur.pl`. It counts the number of occurrences of an element in a list. Improvement in the sequential execution is 1.21. This improvement increases with the number of processors. Additionally, the specialized program keeps on accelerating up to 108 processors while the non specialized does not speed up after 34 processors.

5.9 Chapter Conclusions and Future Work

The topic of multiple specialization of logic programs has received considerable theoretical attention and also many of the existing abstract interpreters implement different degrees of polyvariance for improving the accuracy of the analysis. This is in contrast with the fact that most existing optimization systems which use analysis information are monovariant. We have proposed a simple framework capable of exploiting the multivariance performed during analysis in order to obtain multiple specialization without the need for run-time tests in order to select among different versions of a predicate. This framework is potentially capable of generating an expanded version of the program which contains as many versions of a predicate as calling patterns the analysis has considered for it. However, the program is only expanded if such expansion allows further optimizations, thanks to the use of a minimizing algorithm. As in the case of [Win92], the framework we propose has the two important features of being minimal, i.e., eliminating any of the versions implemented (by collapsing them into other versions) would imply losing some of the optimizations allowed in the expanded program, and of maximal optimization, i.e., no more optimizations are possible by implementing more of the versions generated by analysis. The multiple specialization framework we propose is efficient, as shown by the experimental results, because the core of the process, i.e., the minimization algorithm, does not require the expanded program ($program_0$ in the terminology of Section 5.2) to be materialized. Instead it works with a synthetic representation of the program. To this end, each of the individual analysis versions is annotated with the optimizations which could be performed on it in the expanded program. It is only after minimization that the program is materialized.

Another important feature of the framework we propose is that there is no restriction on the nature of the optimizations considered and the multiple specialization algorithm is independent from it. However, we have also discussed a relevant class of optimizations: those based on abstract executability, a concept which we have formalized in this chapter. We refer to this combination of multiple specialization and abstract executability as *abstract multiple specialization*.

We argue that our experimental results in the context of a parallelizing compiler are encouraging and show that multiple specialization has a reasonable cost both in compilation time and final program size. Also, the results provide some evidence that the resulting programs can show useful speedups in actual execution time and that thus multiple specialization is indeed a relevant technique in practice.

It remains as future work to improve the presented multiple specialization system in several directions. One of them would be the extension of the implementation in order to perform other kinds of optimizations both within program parallelization and beyond this application. Obviously, the specialization system should be augmented in order to be able to detect and materialize the new optimizations. Another direction would be to devise and experiment with different minimization criteria: even though the programs generated by the specializer are minimal to allow *all* possible optimizations, it would sometimes be useful to obtain smaller programs even if some of the optimizations are lost.

Chapter 6

Towards Partial Evaluation based on Generic Abstract Interpretation

Information generated by generic abstract interpreters has long been used to perform program specialization. Additionally, and as we have seen in Chapter 5, if the abstract interpreter generates a multivariant analysis, it is also possible to perform multiple specialization by considering the global and–or tree generated by analysis. Information about values of variables is propagated by simulating program execution and performing fixpoint computations for recursive calls. In contrast, traditional specializers mainly use unfolding for propagating values of variables. However, the program transformations induced by unfolding may lead to important optimizations which are not directly achievable in the existing frameworks for multiple specialization based on abstract interpretation. Our aim in this chapter is to devise a specialization framework which extends that presented in Chapter 5 by integrating the better information propagation of abstract interpretation with the powerful program transformations performed by partial evaluation and which can be implemented with small modifications of existing generic abstract interpreters. With this aim, we will relate the and–or graphs used in abstract interpretation with traditional concepts in partial evaluation and sketch how the sophisticated techniques for controlling partial evaluation which are the result of an important amount of work can be adapted to the proposed specialization framework.

6.1 Introduction

Partial evaluation [JGS93, DGT96] specializes programs for known values of the input. Partial evaluation of logic programs has received considerable attention [Neu90, LS91, Sah93, Gal93, Leu97] and several algorithms parameterized by different control strategies have been proposed which produce useful partial evaluations of programs. Regarding the correctness of such transformations, two conditions, defined on the set of atoms to be partially evaluated, have been identified which ensure correctness of the transformation: "closedness" and "independence" [LS91].

From a practical point of view, effectiveness, that is, finding suitable control strategies which provide an appropriate level of specialization while ensuring termination, is a crucial problem which has also received considerable attention. Much work has been devoted to the study of such control strategies in the context of "on-line" partial evaluation of logic programs [MG95, LD97, LM96]. Usually, control is divided into components: "local control," which controls the unfolding for a given atom, and "global control," which ensures that the set of atoms for which a partial evaluation is to be computed remains finite.

In most of the practical program specialization algorithms, the above mentioned control strategies use, to a greater or lesser degree, information generated by static program analysis. One of the most widely used techniques for static analysis is abstract interpretation [CC77, CC92]. Some of the relations between abstract interpretation and partial evaluation have been identified before [GCS88, GH91, Gal92, CK93, PH95, LS96]. However, the role of analysis is so fundamental that it can be asked whether partial evaluation could be achieved directly by a generic abstract interpretation system such as [Bru91, MH92, CV94]. With this question in mind, we present a method for generating a specialized program directly from the output (an and–or graph) of a generic abstract interpreter, in particular the PLAI system [MH89b, MH90a, MH92]. We then explore two main questions which arise. Firstly, how much specialization can be performed by an abstract interpreter, compared to partial evaluation? Secondly, how do the traditional problems of local and global control appear when placed in the setting of generic abstract interpretation? We conclude that although further study is needed, there seem to be some practical and conceptual advantages in using an abstract interpreter to perform program specialization.

6.2 Goal-Dependent Abstract Interpretation

Goal dependent abstract interpretation takes as input a program P, a predicate symbol[1] p (denoting the exported predicate), and, optionally, a restriction of the run-time bindings of p expressed as an abstract substitution λ. Such an abstract interpretation computes a set of triples $Analysis(P, p, \lambda, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$ such that $\forall i = 1..n\ \forall \theta_c \in \gamma(\lambda_i^c)$ if $p_i\theta_c$ succeeds in P with computed answer θ_s then $\theta_s \in \gamma(\lambda_i^s)$. Additionally, $\forall p_i\theta_i$ that occurs in the concrete computation of $p\theta$ s.t $\theta \in \gamma(\lambda)$ where p is the exported predicate and λ the description of the initial calls of p $\exists\langle p_j, \lambda_j^c, \lambda_j^s\rangle \in Analysis(P, p, \lambda, D_\alpha)$ s.t. $p_i = p_j$ and $\theta \in \gamma(\lambda_j^c)$. This condition is related to the closedness condition [LS91] usually required in partial evaluation. As usual, $\perp$ denotes the abstract substitution such that $\gamma(\perp) = \emptyset$. A tuple $\langle p_j, \lambda_j^c, \perp\rangle$ indicates that all calls to predicate p_j with substitution $\theta \in \gamma(\lambda_j^c)$ either fail or loop, i.e., they do not produce any success substitutions. An analysis is said to be multivariant on calls if more than one triple $\langle p, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p, \lambda_n^c, \lambda_n^s\rangle$ $n \geq 0$ with $\lambda_i^c \neq \lambda_j^c$ for some i, j may be computed for the same predicate.[2] If analysis is multivariant on successes, the triples in $Analysis(P, p, \lambda, D_\alpha)$ will be of the form $\langle p_i, \lambda_i^c, S_i^s\rangle$ where $S_i^s = \{\lambda_{i_1}^s, \ldots, \lambda_{i_j}^s\}$ with $j > 0$. Different analyses may be defined with different levels of multivariance [VDCM93]. However, unless the analysis is multivariant on calls, little specialization may be expected in general. We will limit the discussion to analyses (such as the original analysis algorithm in PLAI and those presented in Chapters 2 and 3) which are multivariant on calls but not on successes, though multivariant successes can also be captured by certain abstract domains, as will be discussed in Section 6.5. In our case, in order to compute $Analysis(P, p, \lambda, D_\alpha)$, an and–or graph is constructed which encodes dependencies among the different triples. Note that when several success substitutions have been computed for the same or–node, the different substitutions have to be summarized in a more general one (possibly losing accuracy) before propagating

[1] Extending the framework to sets of predicate symbols is trivial.

[2] If $n = 0$ then the corresponding predicate is not needed for solving any goal in the considered class (p, λ) and is thus dead-code and may be eliminated.

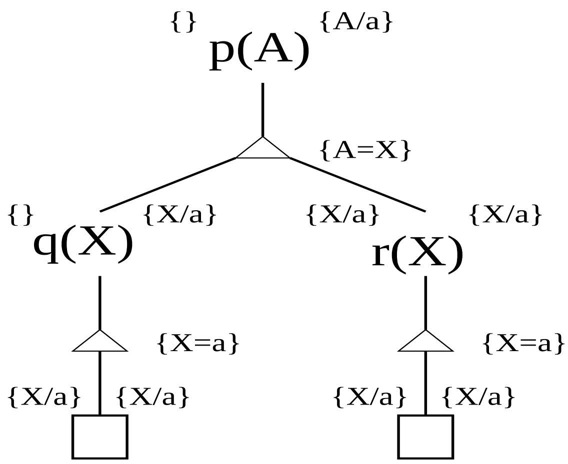

Figure 6.1: And–or analysis graph

this success information. This is done by means of the *least upper bound* (lub) operator. Finiteness of the and–or graph (and thus termination of analysis) is achieved by considering abstract domains with certain characteristics (such as being finite, or of finite height, or without infinite ascending chains) or by the use of a *widening* operator [CC77].

Example 6.2.1 Consider the simple example program below taken from [Leu97]. Figure 6.1 depicts a possible result of analysis for the goal $p(A)$ with A unrestricted using the concrete domain as abstract domain.

```
p(X):- q(X), r(X).
q(a).
r(a).
r(b).
```

We do not describe here how to build analysis and–or graphs. Details can be found in [Bru91, MH90a, MH92] and in Chapter 2. As mentioned in Section 2.2, the graph has two sorts of nodes: those which correspond to atoms (called *or–nodes*) and those which correspond to clauses (called *and–nodes*). Or–nodes are triples $\langle p_i, \lambda_i^c, \lambda_i^s \rangle$. For clarity, in the figures the atom is superscripted with λ^c to the left and λ^s to the right of the atom respectively. For example, the or–node $\langle p(A), \{\}, \{A/a\} \rangle$ is depicted in the figure as ${}^{\{\}}p(A)^{\{A/a\}}$. And–nodes are pairs $\langle Id, Subs\rangle$ where Id is a unique identifier for the node and $Subs$ represents the head unifications of the clause the node refers to. In the figures, they are represented as triangles and the head unifications are depicted to their right. Finally, squares are used to represent the empty (true) atom. Or–nodes have arcs to and–nodes which represent the clauses with which the atom (possibly) unifies. Clearly, if an or–node has no children, the atom will fail. And–nodes have arcs to or–nodes for the corresponding predicate p and call pattern λ^c. If a node $\langle p, \lambda^c, \_\rangle$ is already in the tree it becomes a recursive call and λ^s is obtained by means of a fixpoint computation. $\square$

6.3 Code Generation from an And–Or Graph

The information in $Analysis(P, p, \lambda, D_\alpha)$ has long been used for program optimization. As seen in Chapter 5, multiple specialization allows generating several versions $p_1, \ldots, p_n$ $n \geq 1$ for a predicate p in P. Then, we have to decide which of $p_1, \ldots, p_n$ is appropriate for each call to p. One possibility is to use run-time tests to decide which version to use. If analysis is multivariant on calls but not on successes, another possibility, as done in Chapter 5 and [Win92] is to generate code from $AO(P, p, \lambda, D_\alpha)$ instead of $Analysis(P, p, \lambda, D_\alpha)$. The arcs in $AO(P, p, \lambda, D_\alpha)$ allow determining which p_i to use at each call. Then each version of a predicate receives a unique name and calls are renamed appropriately.

After introducing some notation we present an algorithm which generates a logic program from an analysis and–or graph. This idea was already exploited in [Win92] and has been introduced in detail in Chapter 5. A *program* P is a sequence of clauses of the form H `:-` B where H is an atom and B is a possibly empty conjunction of atoms. The sequence of clauses in a program which define a predicate p is denoted by $def(p)$. We denote by $or(AO)$ the set of or–nodes in an and–or graph AO. Given a node N, $children(N)$ is the sequence of nodes $N_1 :: \ldots :: N_n$ $n \geq 0$ such that there is an arc from N to N' in AO iff $N' = N_i$ for some i and $\forall i, j = 0, \ldots, n$ N_i is to the left of N_j in AO iff $i < j$. Note that $children(N)$ may be applied both to or– and and–nodes. We assume the existence of an injective function pred which given $AO(P, p, \lambda, D_\alpha)$ returns a unique predicate name for each or–node in the graph and $\mathsf{pred}(\langle p(\bar{t}), \lambda, \lambda^s\rangle) = p$ iff $\langle p(\bar{t}), \lambda, \lambda^s\rangle$ is a root node in the graph (to ensure that top-level – exported –

predicate names are maintained).

Definition 6.3.1 [partial concretization] Given an abstract substitution λ, a substitution θ is a *partial concretization* of λ and is denoted $\theta \in part\_conc(\lambda)$ iff $\forall \theta' \in \gamma(\lambda)\ \exists \theta''$ s.t. $\theta' = \theta\theta''$.
$part\_conc(\lambda)$ can be regarded as containing (part of) the definite information about concrete bindings that the abstract substitution λ captures. Note that different partial concretizations of an abstract substitution λ with different accuracy may be considered. For example if the abstract domain is a depth-k abstraction and $\lambda = \{X/f(f(Y)) or X/f(a)\}$, a most accurate $part\_conc(\lambda)$ is $\{X/f(Z)\}$. Note also that $\forall \lambda\ \ \epsilon \in part\_cont(\lambda)$ where ϵ is the empty substitution.

Basically, the algorithm for code generation (Algorithm 6.3.2) given below creates a different version for each different (abstract) call substitution λ^c to the predicate p in the original program. This is easily done by associating a version to each or–node. Note that if we always take the trivial substitution ϵ as $part\_conc(\lambda)$ for any λ (such as in Chapter 5) then such versions are identical except that atoms in clause bodies are renamed to always call the appropriate version.[3] The interest in doing the proposed multiple specialization is that the new program may be subject to further optimizations which were not possible in the original program. Additionally, in Algorithm 6.3.2 predicates whose success substitution is $\perp$ are directly defined as $p(\bar{t}) : -fail$, as it is known that they produce no answers. Even if the success substitution for the predicate (or–node) is not $\perp$, individual clauses for p whose success substitution is $\perp$ (useless clauses) are removed from the final program.

Algorithm 6.3.2 [Code Generation]

Given an analysis and–or graph $AO(P, p, \lambda, D_\alpha)$ generated by analysis for a program P and an atomic goal $\leftarrow p$ with abstract substitution $\lambda \in D_\alpha$ do:

- For each non-empty or–node $N = \langle a(\bar{t}), \lambda^c, \lambda^s \rangle \in or(AO(P, p, \lambda, D_\alpha))$ generate a distinct predicate with name $pred_N = \mathsf{pred}(\langle a(\bar{t}), \lambda^c, \lambda^s \rangle)$.
- Each predicate $pred_N$ is defined by
  - $pred_N(\bar{t})$ `:-` $fail$ if $\lambda^s = \perp$

[3]The program obtained in this way is $program_0$ in the notation of Section 5.4.2.

– $(pred_N(\bar{t}_1)$ `:-` $b'_1)\theta_1 :: \ldots :: (pred_N(\bar{t}_n)$ `:-` $b'_n)\theta_n$ provided that $def(p) = p(\bar{t}_1)$ `:-` $b_1 :: \ldots :: p(\bar{t}_n)$ `:-` b_n otherwise

- Let $children(N) = \langle Id_1, unif_1\rangle :: \ldots :: \langle Id_i, unif_i\rangle :: \ldots :: \langle Id_n, unif_n\rangle$
  Let $children(\langle Id_i, unif_i\rangle) = \langle a_{i1}(\bar{t}_{i1}), \lambda^c_{i1}, \lambda^s_{i1}\rangle :: \ldots :: \langle a_{ik_i}(\bar{t}_{ik_i}), \lambda^c_{ik_i}, \lambda^s_{ik_i}\rangle$.

- Each body b'_i is defined as

  – $b'_i = fail$ if $\lambda^s_{ik_i} = \perp$

  – $b'_i = (pred_{i1}(\bar{t}_{i1}), \ldots, pred_{ik_i}(\bar{t}_{ik_i}))$
  where $pred_{ij} = \mathsf{pred}(\langle a_{ij}(\bar{t}_{ij}), \lambda^c_{ij}, \lambda^c_{ij}\rangle)$ otherwise

- Each substitution θ_i is defined as

  – $\theta_i = \epsilon$ if $b'_i = fail$

  – $\theta_i = \theta_{i1} \ldots \theta_{ik_i}$ provided that
  $\theta_{ij} \in part\_conc(\lambda^s_{ij})$ $j = 1 \ldots k_i$ otherwise

Note that in Algorithm 6.3.2 atoms are specialized w.r.t. answers rather than calls as in traditional partial evaluation. This cannot be done for example if the program contains calls to extra-logical predicates such as `var/1`. Other more conservative algorithms can be used for such cases and for programs with side-effects. Using Algorithm 6.3.2 it is sometimes possible to detect infinite failures of predicates and replace predicate definitions and/or clause bodies by `fail`, which is not possible in partial evaluation. Additionally, as mentioned above, dead-code, i.e., clauses not used to solve the considered goal are removed.

Note that Algorithm 6.3.2 is an improvement over the code-generation phase of Chapter 5 in that it allows applying non-trivial partial concretizations of the abstract (success) substitutions. The program obtained by Algorithm 6.3.2 can then be further optimized by applying the notion of abstract executability as presented in Chapter 5.

Theorem 6.3.3 Let $AO(P, p, \lambda, D_\alpha)$ be an analysis and–or graph for a definite program P and an atomic goal $\leftarrow p$ with the abstract call substitution $\lambda \in D_\alpha$. Let P' be the program obtained from $AO(P, p, \lambda, D_\alpha)$ by Algorithm 6.3.2. Then $\forall \theta_c$ s.t. $\theta_c \in \gamma(\lambda)$

i) $p\theta_c$ succeeds in P' with computed answer θ_s iff $p\theta_c$ succeeds in P with computed answer θ_s.

ii) if $p\theta_c$ finitely fails in P then $p\theta_c$ finitely fails in P'.

Thus, both computed answers and finite failures are preserved. However, the specialized program may fail finitely while the original one loops (see 6.4.2).

6.4 And–Or Graphs Vs. SLD Trees

It is known [LS96] that the propagation of success information during partial evaluation is not optimal compared to that potentially achievable by abstract interpretation.

Example 6.4.1 Consider the program and goal of 6.2.1. The program obtained by applying Algorithm 6.3.2 to the and–or graph in Figure 6.1 is:

```
p(a):- q(a), r(a).
q(a).
r(a).
```

Note that Algorithm 6.3.2 may perform some degree of specialization even if no unfolding is performed. The information in $AO(P, p, \lambda, D_\alpha)$ allows determining that the call to `r(X)` will be performed with `X=a` and thus the second clause for `r` can be eliminated. Such information can only be propagated in partial evaluation by unfolding the atom `q(X)`. □

Example 6.4.2 Consider again the goal and program of 6.2.1 to which a new clause `q(X):- q(X).` is added for predicate `q`. The and–or graph for the new program is depicted in Figure 6.2. The program generated for this graph by Algorithm 6.3.2 is the following:

```
p(a):- q(a), r(a).
q(a).
q(a):- q(a).
r(a).
```

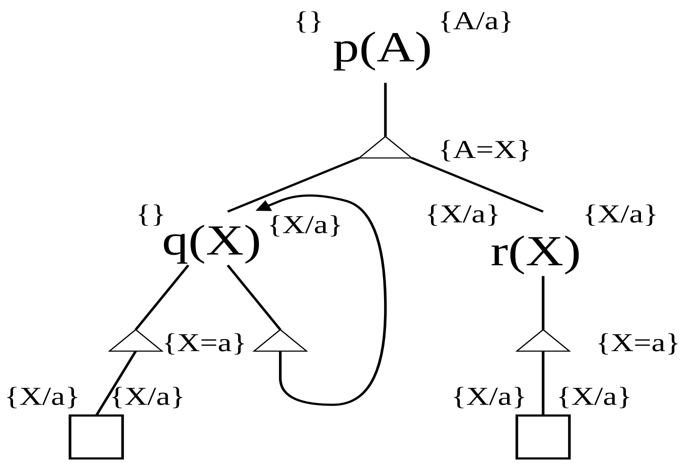

Figure 6.2: Recursive And–or analysis graph

The fact that `r(X)` will only be called with `X=a` cannot be determined by any finite unfolding rule. Note that the original program loops for the goal ← `p(b)` while the specialized one fails finitely. □

The two examples above show that and–or graphs allow a level of success information propagation not possible in traditional partial evaluation, either because the unfolding rule is not aggressive enough (6.4.1) or because the required unfolding would be infinite (6.4.2). This suggests the possible interest of integrating full partial evaluation in an analysis/specialization framework based on abstract interpretation.

In addition, the fact that such a framework can work uniformly with abstract or concrete substitutions makes it more general than partial evaluation and may allow performing optimizations not possible in the traditional approaches to partial evaluation. For example, based on this idea, in Chapter 5, a framework for "abstract specialization" has been presented in which abstract substitutions are used to perform program optimizations. These optimizations use the notion of "abstract executability," which allows reducing calls to some (built-in) predicates at compile-time to the values true or false, or to a set of unifications. Abstract executability had already been introduced informally in [GH91]. An additional pragmatic motivation for this work is the availability of off-the-shelf generic abstract interpretation engines such as PLAI [MH92] or GAIA [CV94] which greatly facilitate the efficient implementation of analyses. The existence of such an abstract interpreter in advanced optimizing compilers is very likely, and using the analyzer itself to perform partial evaluation can result in a great simplification of the architecture of the compiler.

6.5 Partial Evaluation using And–Or Graphs

We have established so far that for any abstract interpretation in the PLAI system (even interpretations over very simple domains such as modes) we can get some corresponding specialized source program with possibly multiple versions by applying Algorithm 6.3.2. Correctness of abstract interpretation ensures that the set of triples computed by analysis must cover all calls performed during execution of any instance of the given initial goal (p, λ). This condition is strongly related to the closedness condition of partial evaluation [LS91]. Furthermore there are well-understood conditions and methods for ensuring termination of an abstract interpretation.

Thus, an important conceptual advantage of formalizing partial evaluation in terms of abstract interpretation is that two of the main concerns of partial evaluation algorithms – namely correctness and termination – are guaranteed by the general principles of abstract interpretation. The other important concern is the degree of specialization that is achieved, which is determined in partial evaluation by the local and global control. We now examine how these control issues appear in the setting of abstract interpretation.

6.5.1 Global Control in Abstract Interpretation

Effectiveness of specialization greatly depends on the set of atoms $\mathbf{A} = \{A_1, \ldots, A_n\}$ for which a specialized version is to be generated. In partial evaluation, this mainly depends on the global control used. If we use the specialization framework based on abstract interpretation, the number of specialized versions depends on the number of or–nodes in the analysis graph. Assuming that the level of multivariance of analysis is fixed (multivariant on calls but not on successes) this is controlled by the choice of abstract domain and widening operators (if any). The finer-grained the abstract domain is, the larger the set $\mathbf{A}$ will be.

In conclusion, the role of so-called global control in partial evaluation is played in abstract interpretation by our particular choice of abstract domain and widening operators (which are strictly required when the abstract domain contains ascending chains which are infinite).

The specialization framework we propose is very general. Depending on the kind of optimizations we are interested in performing, different domains should be used and thus different sets **A** will be obtained. For example, if we are interested in eliminating redundant groundness tests, our abstract domain could in principle collapse the two atoms $p(1)$ and $p(2)$ into one $p(ground)$ as from the point of view of the optimization, whether p is called with the value 1 or 2 is not relevant.

While the main aim of global control is to ensure termination and not to generate too many superfluous versions, it may often be the case that global control (or the domain) does not collapse two versions in the hope that they will lead to different optimizations. If this is not the case, a minimizing step may be performed a posteriori on the and–or graph in order to produce a minimal number of versions while maintaining all optimizations. This was proposed in [Win92], implemented in [PH95] and also discussed in [LM95]. We intend to extend the minimizing algorithm in [PH95] (also presented in Section 5.4) for the case of optimizations based on unfolding.

6.5.2 Local Control in Abstract Interpretation

Local control in partial evaluation determines how each atom in **A** should be unfolded. However, in traditional frameworks for abstract interpretation we usually have a choice for abstract domain and widening operators, but no choice for local control is offered. This is because by default, in abstract interpretation each or-node is related by just one (abstract) unfolding step to its children. This corresponds to a trivial local control (unfolding rule) in partial evaluation.

Several possibilities exist in order to overcome the simplicity of the local control performed by abstract interpretation:

1. According to many authors, [Gal93, LM96] global control is much harder than local control. Thus, subsequent unfolding of the specialized program generated by Algorithm 6.3.2 can be done using traditional unfolding rules to eliminate determinate calls or some non-recursive calls, for example. The

and–or analysis graph may be of much help in order to detect such cases.

2. Use abstract domains which allow propagating enough information about the success of an or–node so as to perform useful specialization on other or-nodes (for example by allowing sets of abstract substitutions). The advantage of this method is that no modification of the abstract interpretation framework is required. Also, as we will see in 6.5.1, it may allow specializations which are not possible by the methods proposed below.[4]

3. Another possibility is a simple modification to the algorithm for abstract interpretation in order to accommodate an unfolding rule. In fact, unfolding can be formalized as a transformation in an and–or graph. In this approach, if the unfolding rule decides that an or-node should not be unfolded, then it is treated as in the usual case. If the rule decides that the atom should be further unfolded, the atom would be analyzed but the corresponding or–node would not be added to the and–or graph. Then, some amount of transformation which is equivalent to the unfolding step should be performed in the analysis graph, and analysis would continue with the usual algorithm. This approach would allow introducing the full power of partial evaluation into our framework by a simple modification of the analysis algorithm. The drawback is the need for the unfolding rule.

4. The last possibility is related to the first alternative in that analysis is performed first with a trivial unfolding rule and once analysis has finished, further unfolding may be performed if desired. However, rather than performing unfolding without modifying the analysis graph as in the first approach, whenever an additional unfolding step is performed, the analysis graph is modified accordingly, using the same graph transformation rules mentioned in the previous approach. However, the difference with the previous approach is that there, unfolding is completely integrated in abstract interpretation and the local control decisions are taken when performing analysis. The advantages over the previous approach are that there is no need to modify the analysis algorithm and that unfolding is performed once the whole analysis graph has been computed. The benefits of the availability of such better information for local control still have to be explored.

[4] Unless multivariance on successes is performed by the analysis framework.

The disadvantage is that in order to achieve as accurate information as in the previous approach it may be required to perform reanalysis in order to propagate the improved information introduced due to the additional unfolding steps, with the associated computational cost. This cost could remain reasonable by the use of incremental analysis techniques such as those presented in [HPMS95].

Example 6.5.1 Consider the following program and the goal $\leftarrow$`r(X)`

```
r(X) :- q(X),p(X).
q(a).
q(f(X)) :- q(X).
p(a).
p(f(X)) :- p(X).
p(g(X)) :- p(X).
```

The third clause for `p` can be eliminated in the specialized program for $\leftarrow$`r(X)`, provided that the call substitution for `p(X)` contains the information that `X=a` or `X=f(Y)`. The abstract domain has to be precise enough to capture, in this case, the set of principal functors of the answers.

Note that no partial evaluation algorithm based on unfolding will be able to eliminate the third clause for `p`, since an atom of form `p(X)` will be produced, no matter what local and global control is used[5]. Thus, simulating unfolding in abstract interpretation (such as methods 1, 3, and 4 above do) will not achieve this specialization either. An approach such as 2 is required. □

6.5.3 Abstract Domains and Widenings for Partial Evaluation

Once we have presented the relation between abstract domains and widening with global control in partial evaluation, we will discuss desired features for performing partial evaluation. Ideally, we would like that

- The domain can simulate the effect of unfolding, which is the means by which bindings are propagated in partial evaluation. Our abstract domain

[5]Conjunctive partial deduction [LSdW96] can solve this problem in a completely different way.

has to be capable of tracking such bindings. This suggests that domains based on term structure are required.

- In addition, the domain needs to distinguish, in a single abstract substitution, several bindings resulting from different branches of computation in order to achieve the approach 2 for local control. A term domain whose least upper bound is based on the *msg*, for instance, will rapidly lose information about multiple answers since all substitutions are combined into one binding.

Two classes of domain which have the above desirable features are:

- The domain of type-graphs [BJ92], [GdW94], [HCC94]. Its drawback is that inter-argument dependencies are lost.
- The domain of sets of depth-k substitutions with set union as the least upper bound operator. However uniform depth bounds are usually either too imprecise (if k is too small) or generate much redundancy if larger values of k are chosen.

One way to eliminate the depth-bound k in the abstract domain it to depend on a suitable widening operator which will guarantee that the set of or–nodes remains finite. Many techniques have been developed for global control of partial evaluation. Such techniques make use of data structures which are very related to the and–or graph such as *characteristic trees* [GB91], [Leu95] (related to *neighbourhoods* [Tur88]), *trace-terms* [GL96], and *global trees* [MG95], and combinations of them [LM96]. Thus, it seems possible to adapt these techniques to the case of abstract interpretation and formalize them as widening operators.

6.6 Chapter Conclusions and Future Work

We have studied the integration of traditional partial evaluation into the specialization framework presented in Chapter 5 which is based on abstract interpretation. Next we present the main conclusions which can be derived from such study. As seen in Chapter 5, a specialized program can be associated with every abstract interpretation. Abstract interpretation can be regarded as having the simple local control strategy of always performing one unfolding step. However, useful specialization can be achieved if the global control is powerful enough. The global control is closely related to the abstract domain which is used since this determines the multivariance of the analysis. If the abstract domain is finite (as is often the case), global control may simply be performed by the abstraction function of the abstract domain. However, if the abstract domain is infinite (as is required for partial evaluation), global control has to be augmented with some kind of widening operator in order to ensure termination. The strategies for global control used in partial evaluation such as those based on characteristic trees [Gallagher-Brynooghe] [LD97] and global trees [MG95] and combination of both [LM96] seem to be applicable to abstract interpretation. More powerful local unfolding strategies may be introduced in abstract interpretation, either unfolding the specialized program derived from abstract interpretation, or by incorporating unfolding into the analysis algorithm. If the latter is implemented, it can be proved that the set of atoms $\mathbf{A}_{PE}$ computed by partial evaluation is not as good an approximation of the computation as the set of atoms $\mathbf{A}_{AI}$ computed by abstract interpretation with the corresponding global and local control.

It remains as future work to experiment with the techniques presented in this chapter. We plan to do so in the context of the PLAI system. Different abstract domains and widening operators for global control should be implemented and experimented with. Efficiency of the approach as well as quality of the specialized programs should be compared to that of existing partial evaluators.

It would also be interesting to study the integration of abstract specialization with recent extensions to traditional partial evaluation such as conjunctive partial evaluation [LSdW96] of logic programs.

Chapter 7

Optimization of Dynamic Scheduling

Dynamic scheduling increases the expressive power of logic programming languages, but also introduces some overhead. In this chapter we present two classes of program transformations designed to reduce this additional overhead, while preserving the operational semantics of the original programs, modulo ordering of literals woken at the same time. The first class of transformations simplifies the delay conditions while the second class moves delayed literals later in the rule body. Application of the program transformations can be automated using information provided by compile-time analysis. We provide experimental results obtained from an implementation of the proposed techniques using the CIAO prototype compiler. Our results show that the techniques can lead to substantial performance improvement.

7.1 Introduction

Most "second-generation" logic programming languages provide a flexible scheduling in which computation generally proceeds left-to-right, but some calls are dynamically "delayed" until their arguments are sufficiently instantiated. This general form of scheduling, often referred to as *dynamic scheduling*, increases the expressive power of (constraint) logic programs. Unfortunately, it also has a significant time and space overhead.

The main objective of this chapter is to develop and evaluate high-level optimization techniques for reducing this additional overhead, while preserving the semantics of the original program. We introduce two different classes of transformations. The first class simplifies the delay conditions associated with a particular literal. The second class of transformations reorders a delayed literal closer to the point where it wakes up. Both classes of transformations essentially preserve the search space and hence the operational behaviour of the original program. The only caveat is that reordering may change the execution order of delayed literals that are woken at exactly the same time. Note that this order is system dependent and it is rare for programmers to rely on a particular ordering.

Using the CIAO prototype compiler we have built a tool which automatically optimizes logic programs with delay using the above transformations. Initial experiments suggest that simplification of delay conditions is widely applicable and can significantly speed up execution, while reordering is less applicable but can also lead to substantial performance improvements.

The promise of optimization of delay conditions using high-level program transformation was already illustrated in [MGH94]. However, optimization was performed by hand and the particular transformation rules used were not detailed. Other related work has concentrated on detecting non-suspension (e.g., [Han93]) or is restricted to the case of some particular delayed conditions (e.g., [Boy93]) usually found in functional languages, and the transformations applied do not guarantee that there will be no performance loss. In [DGB96] program segments in which no suspension occurs are identified in order to perform low-level compiler optimizations. However, no suspension behaviour optimization or reordering is performed.

7.2 Programs with Delay

A *constraint* is essentially a conjunction of predefined predicates, such as term equations or inequalities over the reals, whose arguments are constructed using predefined functions, such as real addition. We let $\overline{\exists}_W \theta$ be constraint θ restricted to the variables W.

7.2.1 Delay Declarations

In dynamically scheduled languages the execution of some literal can be delayed until a particular delay condition holds. A *delay condition*, $Cond$, takes a constraint and returns $true$ or $false$ indicating if evaluation can proceed or should be delayed. Typical primitive delay conditions are `ground(X)` which holds iff `X` is constrained to a unique value, and `nonvar(X)` which holds iff `X` is constrained to be a non-variable term. Delay conditions can be combined to allow more complex delay behaviour. They can be conjoined, written $(Cond_1, Cond_2)$, or disjoined, written $(Cond_1; Cond_2)$.

We require a delay condition $Cond$ to satisfy three properties. First, it must be *downwards closed*: for any two constraints θ, θ' s.t. $\theta' \rightarrow \theta$, if $Cond$ holds for θ, then it also holds for θ'. Second, it should not take variable names into account: for any variable renaming ρ and any constraint θ, if $Cond$ holds for θ then $\rho(Cond)$ holds for $\rho(\theta)$. Third, it should only take into account variables in the condition: for any constraint θ, $Cond$ holds for θ iff $Cond$ holds for $\bar{\exists}_{vars(Cond)}\theta$ where $vars$ returns the set of variables occurring in a syntactic object.

A *delaying literal* is of the form $delay\_until(Cond, L)$, where $Cond$ is a delay condition and L is a literal. Evaluation of L will be delayed until $Cond$ holds for the current constraint store. Delay information can be *predicate-based* and *literal-based*. In the former, the delaying literal appears as a declaration before the definition of the predicate, each instance of the predicate inheriting the delay condition. In the latter, the delaying literal appears in the body of some clause only affecting the literal L. It is straightforward to use predicate-based declarations to imitate literal-based delay, and vice versa. For simplicity, we will restrict ourselves to literal-based delay.

7.2.2 Operational Semantics

An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. A *literal* is either an atom, a delaying literal or a primitive constraint. A *goal* is a finite, non-empty sequence of literals. A *rule* is of the form H `:-` B where H, the *head*, is an atom with distinct variables as arguments and B, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program*, or *program*, is a finite set of rules. The *definition* of an atom A in program P, $defn_P(A)$, is the set of variable renamings of rules in P such that each renaming has A as a head and has distinct new local variables.

When formalizing applicability conditions for our transformations we will be interested in *annotated programs*, in which information about run-time behaviour is collected at *program points* in the initial query and program. Program points occur between literals and at the start and end of all bodies of all rules of the program. For instance, the rule $A\texttt{:-}L_1, ..., L_n$ has the program points $A\texttt{:-}⓪L_1①, ..., \text{(n-1)}L_n\text{ⓝ}$.

We are assuming that all rule heads are normalized, since this simplifies the examples and corresponds to what is done in the analyzer. This is not restrictive since programs can always be normalized. However, so as to preserve the behaviour of the original program under dynamic scheduling, the normalization process must ensure that head unifications are performed simultaneously, that is, grouped together in one primitive constraint. See for instance, the definition of `edge` in the `path` program of Example 7.2.1.

The operational semantics of a program is in terms of its "derivations" which are sequences of reductions between "states". A state $\langle A \mid \theta \mid D\rangle$ consists of the current sequence of active literals A, the current constraint θ, and the current sequence of delayed literals D. Our definition makes use of the parametric function $awoken(D, \theta)$, which returns a sequence of the delayed literals (stripped of their delaying condition) in D that are awoken by constraint θ. The order of the literals returned by *awoken* is system dependent[1]. A state $\langle L :: A \mid \theta \mid D\rangle$ can be *reduced* as follows:

1. If L is a primitive constraint and $\theta \wedge L$ is satisfiable, it is reduced to $\langle D' :: A \mid \theta \wedge L \mid D \setminus D'\rangle$ where $D' = awoken(D, \theta \wedge L)$.
2. If L is an atom, it is reduced to $\langle B :: A \mid \theta \mid D\rangle$ for some rule $(L\texttt{:-}B)$ in the definition of L.
3. If L is the delaying literal $delay\_until(C_L, L_L)$:
   - If C_L holds for θ, it is reduced to $\langle L_L :: A \mid \theta \mid D\rangle$.
   - Otherwise, it is reduced to $\langle A \mid \theta \mid D :: L\rangle$.

[1]However, it is a brave programmer indeed who makes use of such a system dependent feature when programming.

where :: denotes concatenation of sequences and we assume for simplicity that the underlying constraint solver is complete. A *derivation* from state S for program P is a sequence of states $S_0 \Rightarrow S_1 \Rightarrow ... \Rightarrow S_n$ where S_0 is S and there is a reduction from each S_i to S_{i+1}. A *derivation* from a query Q for program P is a derivation from the sequence.

The observational behaviour of a program is given by its "answers" to queries. A finite derivation from a state S for program P is *finished* if the last state in the derivation cannot be reduced. A finished derivation from a state S is *successful* if the last state has form $\langle nil \mid \theta \mid D \rangle$. The constraint $D \wedge \bar{\exists}_{vars(S) \cup vars(D)} \theta$ is an *answer to* S.

Example 7.2.1 The following program finds a path between two nodes in a directed graph.

```
path(X,Y):- X=Y.
path(X,Y):-
      delay_until(ground(Z),edge(X,Z)),
      delay_until(ground(Y),path(Z,Y)).
edge(X,Y):- head(X,Y)=head(a,b).
edge(X,Y):- head(X,Y)=head(b,c).
```

7.3 Simplification of Delay Conditions

Delay conditions may be evaluated each time a variable is touched. Simplifying such conditions can then lead to significant performance improvement. Essentially the behaviour of a delay condition is only relevant during the lifetime of the delaying literal. Hence, we can replace one delay condition by another (more efficient) condition if they are equivalent for all constraint stores that occur during the lifetime of the delaying literal.

7.3.1 Lifetime of a Delaying Literal

The lifetime of a delaying literal can be broken into three stages: *initial* states when it is first selected, *waking* states when it is woken, and *delaying* states when it sits in the collection of delayed literals. Consider the delaying literal $DL \equiv delay\_until(Cond, L)$. The *initial context* for DL, written $I(DL)$, is the set of constraints θ occurring in states of the form $\langle DL :: A \mid \theta \mid D \rangle$. The *delaying context* for DL, denoted $D(DL)$, is the set of constraints θ occurring in states of the form $\langle A \mid \theta \mid D \rangle$, where $DL \in D$. Finally, the *waking context* for DL, $W(DL)$, is the set of constraints θ such that either there is a derivation of the form $\cdots \Rightarrow \langle DL :: A \mid \theta' \mid D \rangle \Rightarrow \langle L :: A \mid \theta \mid D \rangle \Rightarrow \cdots$, or there is a derivation of the form $\cdots \Rightarrow \langle A' \mid \theta' \mid D' \rangle \Rightarrow \langle A \mid \theta \mid D \rangle \Rightarrow \cdots$ where $DL \in D' \setminus D$. We can restrict the constraints in the initial, delaying and waking contexts to the variables in DL since this does not affect the behaviour of the delay condition.

Example 7.3.1 Consider the successful derivation for query `?- Y = b, delay_until(ground(Y),path(X,Y))` and the program of Example 7.2.1. The initial, waking and delaying contexts for each of the delaying literals are:

DL	$I(DL)$	$W(DL)$	$D(DL)$
(a) `delay_until(ground(Y), path(X,Y))`	$\{Y = b\}$	$\{Y = b\}$	$\{\}$
(b) `delay_until(ground(Z),edge(X,Z))`	$\{true\}$	$\{Z = b\}$	$\{true\}$
(c) `delay_until(ground(Y),path(Z,Y))`	$\{Y = b\}$	$\{Y = b\}$	$\{\}$

7.3.2 Rules for Simplification of Delay Conditions

Given the contexts for a delaying literal, simplification can be then performed by applying the following general rule:

SIMP-EQUIV: Replace a condition C, by a more efficient one C', when they are equivalent in all contexts. If $\forall\, \theta \in (I(DL) \cup W(DL) \cup D(DL))$, C holds for θ iff C' holds for θ, then we can rewrite C with C', denoted by $C \Longrightarrow C'$.

The following are special cases of this general rule which are particularly amenable to automatic application.

CONTEXT-INDEP: The following rewriting rules of Boolean algebra can always be exhaustively applied to obtain simpler delay conditions:

1.	$(Cond, true)$	$\Longrightarrow$	$Cond$	**3.**	$(Cond; true)$	$\Longrightarrow$	$true$
2.	$(true, Cond)$	$\Longrightarrow$	$Cond$	**4.**	$(true; Cond)$	$\Longrightarrow$	$true$
5.	$(Cond, false)$	$\Longrightarrow$	$false$	**7.**	$(Cond; false)$	$\Longrightarrow$	$Cond$
6.	$(false, Cond)$	$\Longrightarrow$	$false$	**8.**	$(false; Cond)$	$\Longrightarrow$	$Cond$

Their application will often be enabled by rules 9 and 10 below.

SIMP-TRUE: From downwards closure, delay conditions satisfied in all the initial contexts, are also satisfied in all delaying and waking contexts. Thus:

$$\textbf{9}. \quad \text{If } \forall\theta \in I(DL)\ Cond \text{ holds for } \theta \ : \ Cond \implies true.$$

Finally, we can replace delay_until($true$, L) by L. Delaying literals (a) and (c) in Example 7.3.1 can be simplified in this way.

SIMP-FALSE: From downwards closure, if a delay condition is false in all waking contexts, it has been false throughout the life of a delaying literal:

$$\textbf{10}. \quad \text{If } \forall\theta \in W(DL)\ Cond \text{ does not hold for } \theta \ : \ Cond \implies false.$$

Example 7.3.2 Consider the following program, append3, which appends three lists together and the query ?- append3(X,Y,Z,[a,b,c]).

```
append3(X,Y,Z,T):- delay_until((ground(X);ground(U)), append(X,Y,U)),
                   delay_until((ground(U);ground(T)), append(U,Z,T)).
append(X,Y,Z):-head(X,Y,Z) = head([],V,V).
append(X,Y,Z):-head(X,Y,Z) = head([A|X1],Y1,[A|Z1]),
               delay_until((ground(X1);ground(Z1)),append(X1,Y1,Z1)).
```

All calls to append wake up with the first two arguments free and the last one ground. Hence, we can use rule 10 followed by rule 8 to remove the first primitive delay condition in all delaying literals. Also, the second delaying literal in append3 as well as the delaying literal in the recursive rule of append, never delay since their third argument is ground in all initial contexts. Thus, using rule 9 the delaying condition can be removed. The resulting program (with program point annotations to be used later) is:

```
append3(X,Y,Z,T):- ⓐ delay_until(ground(U), append(X,Y,U)),
                   ⓑ append(U,Z,T). ⓒ
append(X,Y,Z):- ⓓ head(X,Y,Z)=head([],V,V).ⓔ
append(X,Y,Z):- ⓕ head(X,Y,Z)=head([A|X1],Y1,[A|Z1]),
                ⓖ append(X1,Y1,Z1).ⓗ
```

SIMP-CHOOSE: Sometimes, when the delay condition contains disjunctions it is possible to use just part of the condition, discarding the rest:

$$\textbf{11}. \quad \text{if } \forall\theta \in W(DL)\ Cond \text{ holds for } \theta \ : \ (Cond; Cond') \implies Cond$$
$$\textbf{12}. \quad \text{if } \forall\theta \in W(DL)\ Cond \text{ holds for } \theta \ : \ (Cond'; Cond) \implies Cond$$

If both rule 11 and 12 can be applied to a disjunction $(Cond; Cond')$, efficiency considerations should be used to choose the best simplification.

SIMP-PRIM: Replace a primitive condition C_p by a more efficient one C'_p, if they are equivalent in all contexts:

13. If $\forall\, \theta \in (W(DL) \cup D(DL))\ C_p$ holds for θ iff C'_p holds for $\theta : C_p \Longrightarrow C'_p$.

For example, consider the `path` program. In each of the delaying and waking contexts for the delaying literal (b) the variable Z is either free or ground. Hence we could replace the primitive wakeup condition `ground(Z)` by `nonvar(Z)`, which is cheaper, obtaining the same behaviour.

Theorem 7.3.3 Let $DL \equiv delay\_until(Cond, L)$ be a delaying literal and $Cond'$ be a delay condition obtained from $Cond$ by the application of the rewriting rules 1,...,13. Then:

$$\forall \theta \in (I(DL) \cup D(DL) \cup W(DL))\ Cond \text{ holds for } \theta \text{ iff } Cond' \text{ holds for } \theta$$

Thus, application of the rewriting rules will not change the operational behaviour of a program.

7.4 Reordering Delaying Literals

If a delaying literal is known to always delay at some point, it seems worthwhile to try to move it to a later point. In particular, we would like to move the delaying literal to a point where it must wake, thus removing the delay conditions. For this work we restrict ourselves to the (seemingly simple) case of moving delaying literals in the query or rule body in which they appear.

Example 7.4.1 Unfortunately, one has to be careful when moving delaying literals to later points, since this does not always preserve the search space of the program. Consider the following example program and the query `?- delay_until(ground(Y),p(Y)), q(Y,Z).`

```
q(Y,Z) :- Y=2, long_computation(Z).
q(Y,Z) :- Y=3, Z=5.
p(Y) :- Y=3.
```

Since `Y` is initially free, `delay_until(ground(Y),p(Y))` delays. Hence, we might consider moving it after the call to `q`. If we do, we can remove the delaying condition obtaining the reordered query `?- q(Y,Z), p(Y)`. In the original query `p(Y)` is awoken before the `long_computation` occurs, it immediately fails and the second rule for `q` is tried. This succeeds waking `p(Y)`, which also succeeds. In the reordered query `long_computation` will be executed before `p(Y)` wakes up. In the extreme case, it may not terminate.

Intuitively, reordering can only be performed if in the original program the execution of `q(Y,Z)` is guaranteed to have finished by the time `p(Y)` is executed.

Example 7.4.2 Consider the simplified program of Example 7.3.2 and the query `?- append3(X,Y,Z,[a,b,c])`.

The literal $DL \equiv$ `delay_until(ground(U), append(X,Y,U))` can be reordered after `append(U,Z,T)`, even though DL does not delay until the execution of `append(U,Z,T)` is finished. This is because DL only wakes up at program point ⓔ, i.e. at the end of the execution of `append(U,Z,T)`. Hence, DL cannot affect the execution of `append(U,Z,T)`.

7.4.1 "Wakeups" and Program Points

We now formalize the transformation used in the above example. To reorder correctly we need to know at which points in the program a delaying literal can wake. We now define how to attach "wakeups" to program points. Consider the derivation:

$$\langle L :: A \mid \theta \mid D\rangle \Rightarrow \langle D' :: A \mid \theta \wedge L \mid D \setminus D'\rangle \Rightarrow^* \langle A \mid \theta' \mid D''\rangle$$

where L is a primitive constraint, $\theta \wedge L$ is satisfiable and $D' = awoken(D, \theta \wedge L)$. In this derivation the delaying literals in D' have awoken at the program point immediately after the constraint L. However, the set of delaying literals that wakeup in between L and the execution of A is $D \setminus D''$. This is, in general, a superset of D', since the execution of D' may generate new constraints which may in turn wake up other delaying literals in $D \setminus D'$. For the above derivation, we then consider the set $D \setminus D''$ as waking up at the program point after L. We define the annotation of a program P for query Q as the mapping from the program points of P to the union, for all possible derivations of Q, of the sets of waking up literals at that program point.

Example 7.4.3 In the program and query of Example 7.3.2, the wakeups attached to program point ⓔ are $\{delay\_until(ground(U), append(X, Y, U))\}$. The rest of program points have no associated wakeups.

Example 7.4.4 To see why we have to add all the delaying literals that wake up in between L and A consider the following program with query `?- g(X,Y)`.

```
g(X,Y) :-    ⓐ delay_until(ground(X), p(X,Y,Z)),ⓑ
             delay_until(ground(Y), q(Y)), ⓒ r(X).ⓓ
r(X) :-      ⓔ X = 1.ⓕ
p(X,Y,Z) :- ⓖ Y = 1,ⓗ long_computation(Z).ⓘ
q(Y) :-      ⓙ Y = 2.ⓚ
```

The annotated program points with associated wakeups are:

ⓕ $\{delay\_until(ground(X), p(X, Y, Z)), delay\_until(ground(Y), q(Y))\}$
ⓗ $\{delay\_until(ground(Y), q(Y))\}$.

If we had annotated ⓕ with the set of literals immediately awoken by `X=1`, that is D', we would only obtain the first delaying literal, `delay_until(ground(X), p(X,Y,Z))`. Thus, it would be hard to see that the second delaying literal wakes up within `r(X)`.

Information about the program points at which a delaying literal can be awoken leads to a simple methodology for reordering a delaying literal. Before we detail the transformation we need to define at which program points a delayed literal can be awoken for reordering to be allowed. We first define the set of program points for a particular goal at which delayed literals may be awoken during the evaluation of the goal. The set of *instantiating program points* for a goal ⓐGⓑ, denoted $IPP(G)$, are:

$$IPP(G) = \begin{cases} \{ⓑ\} & \text{if } G \text{ is a constraint} \\ IPP(L) & \text{if } G = delay\_until(Cond, L) \\ \bigcup_{G:-B_i \in defn_p(G)} IPP(B_i) & \text{if } G \text{ is an atom} \\ IPP(L') \bigcup IPP(G') & \text{if } G = L', G' \end{cases}$$

Now we define the subset $NIPP(G)$ of instantiating program points which are *non-final* for a goal ⓐGⓑ. A delayed literal will not be allowed to move across a goal if it wakes up at a non-final point:

$$NIPP(G) = \begin{cases} \emptyset & \text{if } G \text{ is a constraint} \\ NIPP(L) & \text{if } G = delay\_until(Cond, L) \\ \bigcup_{G \texttt{:-} B_i \in defn_p(G)} NIPP(B_i) & \text{if } G \text{ is an atom} \\ IPP(L') \bigcup NIPP(G') & \text{if } G = L', G' \end{cases}$$

The set of *final instantiating program points* for goal ⓐGⓑ, denoted $FIPP(G)$, is simply $IPP(G) - NIPP(G)$. For example the instantiating program points for ⓑ$append(U, Z, T)$.ⓒ in Example 7.4.2 are {ⓔ, ⓖ }. The final instantiating program points for ⓑ$append(U, Z, T)$.ⓒ are { ⓔ }.

7.4.2 Rules for Reordering Delaying Literals

We can now define two transformation rules which provide sufficient conditions for reordering a delaying literal across the body in which it appears, while preserving the semantics of the program. Consider a rule of the form $H : -L_1, \ldots, L_i, DL, L_{i+2}, \ldots, L_j, L_{j+1}, \ldots L_n$ where DL is a delaying literal.

DOESNT-WAKE: We can reorder DL until immediately before L_{j+1} if DL is definitely delayed before L_{i+2} and it does not wake at any instantiating program point in the conjunction of literals $L_{i+2}, \ldots, L_j$.

FINAL-WAKE: We can reorder DL until immediately after L_j if DL is definitely delayed before L_{i+2} and it does not wake at any non-final (instantiating) program point in the conjunction of literals $L_{i+2}, \ldots, L_j$. In addition if DL wakes up at a final program point together with another delaying literal DL', then DL wakes only at final program points of the literal DL'.

Example 7.4.5 Consider the program and query of Example 7.4.4. The literal `delay_until(ground(Y), q(Y))` only wakes up at ⓕ which is a final program point for `r(X)`. However, reordering such literals would result in `long_computation(Z)` being performed. This is why an additional condition is introduced in the FINAL-WAKE rule. As `delay_until(ground(X), p(X,Y,Z))` also wakes at program point ⓕ and `delay_until(ground(Y), q(Y))` wakes up at ⓗ which is a non-final program point within `p(X,Y,Z)`, FINAL-WAKE is not applicable. We can however move `delay_until(ground(X), p(X,Y,Z))` until after the conjunction `delay_until(ground(Y), q(Y)), r(X)` using the FINAL-WAKE rule, since it only wakes at ⓕ, a final program point of this conjunction. At this point it is guaranteed to wake, the delay condition can be removed, and the optimized rule is `g(X,Y):- delay_until(ground(Y), q(Y)), r(X), p(X,Y,Z).`

If we now annotate this program for the query `?- g(X,Y)`, the new annotations would show that `delay_until(ground(Y), q(Y))` could be moved after `r(X)`, using the DOESNT-WAKE rule.

The reason why DOESNT-WAKE is correct is that since DL is not awoken during evaluation of $L_{i+2}, ..., L_j$ it cannot affect the evaluation, and so can be added later. The reason why FINAL-WAKE is correct is that since DL is the last literal evaluated before returning from L_j, we can equivalently evaluate it as the first literal after returning from L_j.

Unfortunately, there is a subtle problem with this reasoning. The problem is that both reordering rules may change the order in which literals are delayed, and so may affect the system dependent order in which literal are returned by *awoken*. This is only a problem in the case when more than one literal is awoken at the same time.

Example 7.4.6 Consider the following program and query: `?- g(T)`

```
g(T):- delay_until(ground(T),p(T)),
       delay_until(ground(T),q(T)), T = 1.  ⓐ
p(T):- T = 2.
q(T):- long_computation(Z).
```

At ⓐ both delaying literals wake. If *awoken* returns `p(T)::q(T)`, the query quickly fails. There is no annotation in the body of `q(T)` which includes the delaying literal for `p(T)`, hence FINAL-WAKE is applicable for the literal. Hence, `g(T) :- delay_until(ground(T), q(T)), T =1, p(T).` is a correct reordering. But for this program the `long_computation` is executed.

However, note that behaviour of the transformed program is equivalent to that of the original program if *awoken* had returned `q(T)::p(T)` instead.

Therefore we have the somewhat weaker correctness result for the reordering rules, that the transformed program behaves equivalently to the original program for some choice of the *awoken* function. However, as noted earlier it is rare for programmers to rely on the system dependent ordering of *awoken* to prune the program search space.

7.5 Automating the Optimization

We have built a prototype automatic transformation tool which works as follows. First, the original program is analyzed, and the program annotated with the inferred information is given to the optimizer. Using this information, the optimizer first simplifies delay conditions as much as possible and then reorders those literals which are sure to delay. To reduce problems of the kind presented in Example 7.4.6, whenever more than one literal is reordered to the same program point and no information about waking order is available, the optimizer keeps the relative order in which the reordered literals appeared in the original program. In addition, reordering may enable further optimizations, as the initial contexts in the new positions will in general be more instantiated than in the original ones. Hence, another analysis–optimization iteration could be performed. In some cases the current implementation can perform further optimizations without re-analysis. Other optimizations traditionally used with fixed-scheduling constraint logic programs can also be performed after transformation. Currently we perform dead code elimination and simplification of built-ins.

Different analysis frameworks have been recently developed for logic programs with dynamic scheduling (e.g., [MGH94, DGB96, GMS95]). In our prototype we use the approach of [GMS95]. However, simplification can be performed with any analysis framework which, for a given analysis domain, approximates the initial, delaying and waking contexts for each delaying literal. For reordering, the analyzer needs to provide a description of the set of waking up literals at each program point. For the traditional optimizations, the analyzer needs to also provide a description of the constraints at each program point.

The experimental evaluation uses the information provided by three different abstract domains. The Def domain[2] [GH93] approximates *groundness* informa-

[2]This domain is a variant of the *Prop* domain [MSJ94].

tion. Thus, it can be used to infer the satisfiability of *ground* and *nonvar* tests. The ShFr domain [MH91] approximates not only groundness but also *sharing* and *freeness* information. Freeness information allows us to prove the unsatisfiability of *ground* and *nonvar* tests. The Aeq domain[3] complements ShFr with more complex modes like non-freeness, non-groundness and linearity. Non-freeness and non-groundness allow more accurate information about the behaviour of *nonvar* and *ground* tests. Linearity improves sharing and therefore the propagation of the other properties.

7.6 Experimental Results

Four different sets of benchmarks have been used in our experiments. The first set corresponds to those used in [MGH94, GMS95]. They are essentially new, reversible versions of some standard symbolic programs. The original programs used static scheduling and could only be run in one mode. In the new versions dynamic scheduling has been added to allow them to run both forwards and backwards. This first set includes `append3` (concatenates 3 lists), `nrev` (reverses a list in a naive way), `permute` (computes all permutations of a list), and `qsort` (the quick-sort algorithm). The second set corresponds to standard mathematical benchmarks in which dynamic scheduling has been added to arithmetic constraints so as to allow them to run both forwards and backwards. This set includes `fac` (factorial), `fib` (Fibonacci), and `mortgage`. The programs in the third set are programs with dynamic scheduling resulting from the automatic translation of concurrent logic programs by the Qd-Janus system [Deb93]. Dynamic scheduling is used to emulate the concurrency present in the original programs. This set includes `nand` (a nand-gate circuit designer, written by E. Tick) and `transp` (matrix transposer, written by V. Saraswat). The Qd-Janus compiler already performs analysis and optimization of its input programs and aims to produce code with little redundant concurrency. The Prolog code it produces is competitive in performance with compilers specifically designed for concurrent logic programs. The last set are NU-Prolog programs written by L. Naish which exploit rather complex dynamic scheduling for different purposes, and which have

[3]This domain is a modification of that of Mulkers et al [MSJB95]. We have added more complex modes but do not make full use of the equation modeling component.

been translated into SICStus. This set includes `nqueen` (coroutining n-queens), `slowsort` (a generate and test algorithm), `interpl` (simple interpreter for coroutining programs), and `termcompare` (term comparison).

Benchmark	Cl	Lit	DL
append3	3	3	3
nrev	4	3	3
permute	4	3	3
qsort	7	9	9
fac	8	27	3
fib	6	17	4
mortgage	8	29	5
nand	90	157	13
transp	112	180	20
nqueen	11	15	11
slowsort	9	8	8
interpl	11	10	3
termcompare	27	37	26

Table 7.1: Benchmark Characteristics.

Table 7.1 provides information regarding the complexity of the benchmarks used in our experiments. Cl is the number of clauses analyzed, Lit is the number of literals, and DL is the number of delaying literals. Since programs have been normalized the (usually high) number of term equations is not counted in Lit. DL includes all calls to predicates affected by a delay declaration.

For the first two sets of benchmarks we will consider two different versions of each program: in the first one ground conditions are used in the delaying literals (`_gr` suffix), while in the second one nonvar conditions are used (`_nv` suffix). Note, however, that in `nrev` and `qsort` nonvar conditions do not always guarantee termination. Thus a mix of ground and nonvar conditions is used in the "`_nv`" version of these benchmarks. Different rows associated to the same benchmarks indicate different queries. For the first two sets of benchmarks they perform forward and backward execution.

The programs have been implemented using `block` (SICStus predicate-based

Benchmark	Analysis (sec)			Transformation (msec)		
	Def	ShFr	Aeq	Def	ShFr	Aeq
append3_gr	0.0	0.0	0.0	7	7	10
	0.1	0.1	0.2	13	20	10
append3_nv	0.0	0.0	0.0	7	10	7
	0.1	0.2	0.4	20	20	20
nrev_gr	0.0	0.0	0.0	10	10	10
	0.1	0.1	0.2	17	17	20
nrev_nv	0.0	0.0	0.0	10	10	10
	0.4	0.1	0.2	30	10	17
permute_gr	0.0	0.0	0.0	10	10	7
	0.3	0.5	0.2	17	23	13
permute_nv	0.0	0.0	0.0	7	10	10
	0.4	2.8	5.8	20	57	50
qsort_gr	0.0	0.1	0.1	13	13	10
	3.2	2.8	12.3	47	63	93
qsort_nv	0.0	0.1	0.1	10	20	13
	23.9	1517.3	∞	243	2730	∞
fac_gr	0.0	0.0	0.0	20	30	23
	0.3	0.2	0.4	40	37	57
fac_nv	0.0	0.0	0.0	23	23	27
	0.3	0.2	0.4	37	40	50
fib_gr	0.0	0.0	0.1	23	23	27
	0.8	0.6	0.9	43	40	70
fib_nv	0.0	0.0	0.1	13	20	30
	0.6	0.6	0.9	43	40	67
mortgage_gr	0.0	0.0	0.1	27	33	40
	0.7	0.3	0.5	77	50	67
mortgage_nv	0.0	0.0	0.1	33	30	40
	0.5	0.3	0.5	57	50	70
nand	0.5	0.7	1.6	297	303	373
transp	168.5	1621.5	∞	1087	1620	∞
nqueen	0.0	0.1	0.1	30	40	40
	2.6	3.7	7.5	77	113	140
slowsort	0.0	0.0	0.1	23	33	23
	0.3	0.5	1.2	33	43	43
interpl	0.9	3.0	2.0	37	50	37
termcompare	0.2	0.3	0.3	70	90	83
	7.8	19.1	158.3	233	380	1393

Table 7.2: Efficiency Results

delay) declarations whenever possible, i.e., when only `nonvar` tests were involved. This is because they are the most efficient delay declarations in SICStus. Otherwise, `when/2` (SICStus literal-based delay) declarations were used. The only exceptions are the programs in the third class where the compiler produces literal-based `freeze` declarations.

Our first set of experiments evaluates the cost of the automatic transformation using our prototype compiler described in the previous section. Table 7.2 shows the analysis times in seconds for each of the abstract domains described in the previous section as well as the time in milliseconds required to optimize the programs using the information inferred. The times are for code run under SICStus Prolog version 3.0 on a 55MHz SPARCstation 10 with 64 MBytes of memory. An ∞ indicates that the analyzer ran out of memory because too many calling patterns were produced in the analysis.

Analysis times are generally acceptable, except for three programs: `qsort_nv`, `transp`, and `termcompare`. Their times are slow because of their complex dynamic behaviour. However, it should be remembered that the analysis of logic programs with dynamic scheduling is still in its infancy and that we are using a prototype analyzer. As this technology improves, analysis time should markedly decrease. Transformation times are very low – only when the amount of analysis information is enormous does the time reach more than one second.

Our second experiment evaluates the effectiveness of the optimizations. Table 7.3 shows the execution time in milliseconds for the original programs, and the speed-up obtained by the automatically transformed programs using simplification and then both simplification and reordering. We do this for each abstract domain. Since the information provided by Def never allows reordering, its column has been eliminated from Simp. + Reord. A blank entry in the Simplification column indicates that no delay condition was optimized, and a blank entry in the Simp. + Reord column indicates no reordering was performed and hence the speedup is the same as for simplification alone. A † indicates that no delaying literals remain in the transformed program.

Our results demonstrate that both simplification and reordering can lead to an order of magnitude performance improvement, and that they give reasonable speedups in most benchmarks. The benchmarks `nand`, `transp`, `interpl` and `termcompare` which did not exhibit any measurable speedup belong to the last

Benchmark	**Orig**	**Simplification**			**Simp. + Reord**	
		Def	ShFr	Aeq	ShFr	Aeq
append3_gr	339430	439.68 †	439.68 †	439.68 †		
	7438	1.49	1.49	1.49		2.10 †
append3_nv	816	1.06 †	1.06 †	1.06 †		
	3682	1.03	1.03	1.03		
nrev_gr	342220	1368.88 †	1368.88 †	1368.88 †		
	5864	5.55	5.55	5.55		68.19 †
nrev_nv	3682	1.03	1.03	1.03		
	1086		1.03	1.03		12.63 †
permute_gr	28982	11.13 †	11.13 †	11.13 †		
	1452	2.66	2.66	2.66		5.46 †
permute_nv	2574	1.00 †	1.00 †	1.00 †		
	836	1.02	1.02	1.02		
qsort_gr	5908	173.76 †	173.76 †	173.76 †		
	2138	2.31	2.31	2.31		
qsort_nv	818	27.27 †	27.27 †	27.27 †		
	1320		1.00	—		—
fac_gr	3268	1.54 †	1.54 †	1.54 †		
	15322	2.62	2.62	2.62		
fac_nv	2100	1.00 †	1.00 †	1.00 †		
	1830	1.07	1.07	1.07		
fib_gr	35784	61.06 †	61.06 †	61.06 †		
	37848	1.65	1.65	1.65		
fib_nv	668	1.11 †	1.11 †	1.11 †		
	722	1.06	1.06	1.06		
mortgage_gr	4202	6.55 †	6.59 †	6.59 †		
	5138	1.66	2.52	2.52	53.52 †	53.52 †
mortgage_nv	646	1.03 †	1.03 †	1.03 †		
	330	1.25	1.57	1.57	3.44 †	3.44 †
nand	464	1.00	1.00	1.00		
transp	5609	1.00	1.00	—		—
nqueen	27218	8.14	8.14	8.14		
	4684	1.39	1.39	1.39		
slowsort	1466	8.52	8.52	8.52		
	3388	1.00	1.00	1.00		
interpl	3160	1.00	1.00	1.00		
termcompare	4418	1.14	1.14	1.14		1.33
	4456	1.00	1.00	1.00		1.08

Table 7.3: Effectiveness Results

two sets of benchmarks. It is perhaps not surprising that our optimizer found programs in these classes difficult to improve since they were either produced by a rather clever transformer which tries to avoid introducing delay where it is not needed or hand-crafted by an expert in dynamic scheduling. Unsurprisingly, the more sophisticated the analysis domain, the better the speed up. In particular the extra precision of Aeq is required to gain the most benefit from reordering.

7.7 Chapter Conclusions

The experimental results obtained are promising. They show that the transformation techniques introduced in this chapter can be automated and lead to significant performance improvement. This is important because dynamic scheduling looks set to become increasingly prevalent in (constraint) logic programming languages because of its importance in implementing constraint solvers and controlling search as well as for implementing concurrency. We noted that the effect of the transformation greatly depends on the implementation of the delay declarations, and therefore on the target language. In particular, since groundness is an expensive test its simplification gives great benefits. Lesser, although still significant benefits can be obtained for other delay conditions. However this work is only a first step. Many other techniques for the automatic transformation of programs with dynamic scheduling remain to be investigated.

Part III

Program Debugging

Chapter 8

An Assertion Language for Debugging

Assertions allow expressing properties of programs. Several assertion languages have been used in the past in different contexts related to program debugging. In this chapter we propose a general language of assertions which should be useful for the different tools to be used for validation and debugging of constraint logic programs. There is clearly a trade-off between the expressive power of the language of assertions and the difficulty of dealing with it. The assertion language proposed is parametric w.r.t. the particular constraint domain, implementation, and properties of interest being used in each different tool. The language proposed is very general in that it poses few restrictions on the kind of properties which may be expressed. This results in a very rich language. However, individual tools will only use the parts of this language that are relevant to such tools. We discuss the possibility of performing compile-time checking of assertions and sketch a framework for automatic generation of programs which check assertions at run-time.

8.1 Introduction

Assertions are linguistic constructions which allow expressing properties of programs. We may be interested in expressing many different kinds of properties as assertions may be used in different contexts and for different purposes. Some contexts in which assertions have been used in the past are:

Run-time checking In imperative programming, assertions have been traditionally used to express conditions about the program which should hold at run-time. A usual example is to check that the value of a variable remains within a given range at a given program point. If assertions are found not to hold a warning is given to the user. Note that in this context, assertions express properties about the run-time behaviour of the program which *should hold* if the program is correct (see [Vet94] for an application to CLP).

Replacing the oracle In declarative debugging [Sha82], the existence of an *oracle* (normally the user) which is capable of answering questions about the intended behaviour of the program is assumed. In this context assertions have been used in order to replace the oracle as much as possible by allowing the user expressing properties which should hold if the program were correct [DNTM88, DNTM89, BDM97]. If it is possible to answer the questions posed by the declarative debugger just by using the information given as assertions, then there is no need to ask the oracle (the user). Note that here again, assertions are used to express properties which *should hold* for the program.

Compile-time checking Another use of assertions is as a means of expressing properties about the program which are checked at compile-time, usually by means of program analysis. These properties *should hold*, i.e., otherwise a bug exists in the program. An example of this are type declarations (e.g., [HL94, SHC96], functional languages, etc.), which have been shown to be useful in debugging. Generally, and in order to be able to check these properties at compile-time, the expressible properties are restricted to a statically decidable set.

Providing information to the optimizer Assertions have also been proposed as a means of providing information to an optimizer in order to perform additional optimizations during code generation (e.g., [SHC96], which also implements checking). In this context, assertions do not express properties which should hold for the program, but rather properties which *do hold* for the program at hand. Note that if the program is not correct, the properties which hold may not coincide with the properties which should hold.

General communication with the compiler In a setting where there is both a static inference system, such as an abstract interpreter [CC77, GHB+96], and an optimizer, assertions have also been proposed as a means of providing additional information to the analyzer, which it can use both to increase the precision of the information it infers and/or to perform additional optimizations during code generation [WHD88, VD92, MS92, KMM+96]. An example are assertions which state some (but not all) types in a type *inferencing* system [BCHP96]. Also, assertions can be used to represent analysis output in source form and to communicate different modules of the compiler which deal with analysis information (see chap:Analysis-Full-Lang). In this context, assertions again do not express properties which should hold for the program, but rather properties which *do hold* for the program at hand. Note that if the program is not correct, the properties which hold may not coincide with the properties which should hold.

In this work we are interested in an assertion language which *integrates all of the lines above*. Furthermore, in addition to these uses, the assertion language should serve other purposes. Although not further discussed here, we would also like to generate documentation automatically from the program source (in the "literate programming" style [D.84]) based in part on the information present in the assertions. More details can be found in [D.84, HG97].

Assertions can be classified according to many different orthogonal criteria. For example, even though they are used for different purposes, the first three contexts above have in common that assertions express properties which should hold (intended properties), while the last two ones refer to properties which actually hold (actual properties) for the program.

The aim of this chapter is to serve as a basis for the design of an assertion language which suffices for the purpose of debugging in the context of constraint logic programming (CLP) [JM94] languages, while remaining tractable. There is a clear trade-off between the expressive power of the language of assertions and the difficulty of dealing with it. The assertion language proposed is parametric w.r.t. the constraint domain and the particular CLP platform being used and thus can be used for any of them. For example, an instance of the assertion language we propose has been implemented in the CIAO system [HBGP95, Bue95]. Details can be found in [Gro97].

The structure of this chapter is the following. Section 8.2 briefly discusses the kind of properties expressible in the assertion language. Section 8.3 presents two kinds of assertions to be used for declarative properties of programs. Assertions for operational properties are presented in Sections 8.4 through 8.6. Both Sections 8.4 and 8.5 deal with predicate assertions, i.e., used to express properties of predicates. A set of basic assertions is presented in Section 8.4 and Section 8.5 presents a compound assertion which allows grouping basic predicate assertions into one. Section 8.6 introduces program-point assertions. Finally, Section 8.7 discusses the use of assertions for expressing properties of the actual program (as in the case of expressing results of analysis).

8.2 Dealing with the Multiple Objectives of Assertions

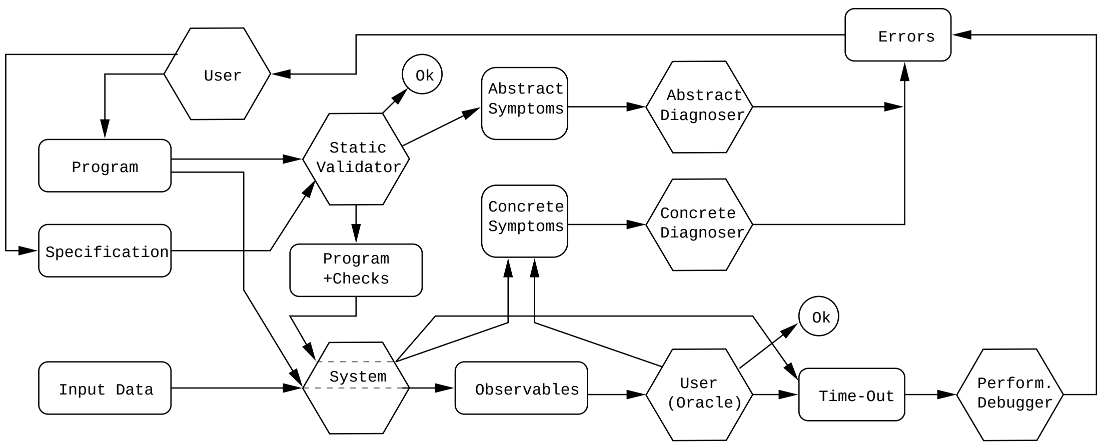

Figure 8.1: An Advanced Development and Debugging Environment

In an advanced development and debugging environment for constraint logic programs, different tools should co-exist. Figure 8.1 gives an idea of the complexity and intercommunication needs of the tools. A detailed description of a possible set of such tools can be found in [ABB+97]. As the intention in this chapter is to have a language of assertions which allows expressing any property which is of interest for any debugging (and validation) tool, it is very hard to restrict beforehand the kind of properties which can appear in assertions. Clearly, not all tools will be capable of dealing with *all* properties expressible in our assertion language. Rather than having different assertion languages for each tool, we propose the use of the same assertion language for all of them, since this will facilitate communication among the different tools and enable easy reuse of information, i.e., once a property has been stated there is no need to repeat it for the different tools. Each tool will only make use of the part of the information given as assertions which the tool *understands*.

8.2.1 Compile-time Checking of Assertions

In the context of compile-time checking, a well known example of a language for expressing properties of programs are *type systems*, which have been proved to be very useful for compile-time bug detection. Type systems allow providing high level description of program procedures. An example of these descriptions may be

```
:- type qsort(list,list).
```

Which states that both arguments of the predicate `qsort` are of the type *list*. Usually, the existence of a type checker is assumed and type declarations are checked at compile-time. Types can also be checked at run-time for input data which is not available at compile-time. The language for providing type declarations (the type system) is restricted in such a way that compile-time checking of types is (quasi) *decidable*, i.e., if the program is correct w.r.t. the given type declarations, the type checker will be (in most cases) able to prove it. If the program does not pass the type check, it is rejected and compilation aborts. However, if the program passes the type check, it is guaranteed that the program will not go wrong w.r.t. the given type declarations. Unfortunately, the fact that type systems are expected to be decidable greatly restricts the kinds of properties which are expressible in type declarations. Also, in many cases type declarations are mandatory for all program procedures and we would like to have *optional* assertions which can be incrementally added during debugging.

For other contexts which are not directly related to compile-time checking, such as replacing the oracle in declarative debugging, run-time checking, providing information to the optimizer, and general communication with the compiler, we may be interested in expressing properties which do not fall into type systems, and which are possibly undecidable at compile-time. Example of properties of interest which lay out of type systems are "The second argument of `qsort` is an ordered list", "If we execute qsort with the first argument being the list `[2,1]` on termination of the execution the second argument should be `[1,2]`", "If we call `qsort` with the first argument a ground list and the second a variable, computation should be deterministic, not fail and terminate". Thus, since it is our objective that the assertion language be common for all the above contexts, the resulting assertion language needs to be more general than type systems while at the same time include them. In this we follow the spirit of [BCHP96, Cou97]. However, we do not discard the definition and use of a decidable type system if so desired. Even though the properties given in assertions may not be decidable in general, it is our view that assertions should be checked as much as possible at compile-time via static analysis. The inference system should be able to make conservative approximations in the cases in which precise information cannot be inferred (and some assertions may remain unproven).

Note that if the properties allowed in assertions are not decidable the approach to the treatment of "don't know" when trying to statically prove a possibly undecidable assertion has to be weaker than the one used for *strong* type systems. The case that the analysis is not capable either to prove nor disprove that an assertion holds may be because we do not have an accurate enough analysis available or simply because the assertion is not statically decidable.

8.2.2 Defining Executable Assertions

In both run-time checking and replacing the oracle, the properties in assertions need to be executable. Thus, a property can be any of the following:

- **a built-in predicate or constraint**. E.g. `ground(X)`, `X>5`. Extra-logical properties may also be used, such as `var(X)`. However, for some tools not all built-in predicates may be allowed.

- **a user-defined expression in a restricted syntax**. Such restricted syntax needs to have a defined translation into (a subset of) the underlying CLP language. Usually such restricted syntax ensures that any property expressible in it is statically decidable. An example are user-defined types using regular types. E.g., `intlist ::= [] | [integer|intlist].` which is equivalent to the program
  ```
  intlist([]).
  intlist([X|T]):- integer(X), intlist(T).
  ```

- **a user-defined program**. Similar in concept to the one above but rather than in a restricted syntax, the user can define his own properties using the full underlying CLP language. As a result, the properties defined may not be statically decidable. As an example, consider defining the predicate `sort(A,B)` to check that a more involved algorithm such as `qsort(A,B)` produces correct results.

- an expression including **conjunctions, disjunctions, and/or negations** of properties.[1]

Depending on the use we make of each assertion and the particular implementation of the diagnosis tool we may restrict the set of properties treated by the particular tool. Properties which are not allowed may still be present but they will simply be ignored by such tool. For replacing the oracle we would like our executable specifications (assertions) to always terminate. In the context of run-time checking, we require:

- that the execution of the code which performs the run-time checking does not introduce non-termination into a terminating program.

- that the code for run-time tests does not modify the constraint store. This way we ensure that run-time checking will not introduce incompleteness w.r.t. the original program, i.e., for a given query, any D-atom which is an instance of an answer in the original program must also be an instance of an answer in the program with run-time tests.

[1] Except for the *Comp_prop* properties introduced in Section 8.4.4.

8.3 Assertions for Declarative Properties

Semantics associates a meaning to a given syntax (generally of a program). Foundations and examples of program semantics will be presented in Section 9.2. By now, it suffices to mention that one of the main features of Constraint Logic Programming is the existence of a *declarative semantics* which allows concentrating on *what* the program computes and not on *how* it should be computed. It is desirable to exploit this feature by the use of declarative assertions as much as possible. This is done for example in declarative debuggers.

As an example, consider the case of CLP(D), where D is the domain of values. In classical logic programming D is the Herbrand Universe and in CLP($\mathcal{R}$) D is the set of real numbers and of terms (for example lists) containing real numbers. The declarative semantics in CLP(D) associates a D-*model* to each program P which corresponds to the *meaning* of P, which is denoted $[\![P]\!]$. $[\![P]\!]$ is a set of D-atoms, where a D-atom is an expression $p(d_1, \ldots, d_n)$ with $n \geq 0$ where p is an n-ary predicate symbol and $d_i \in D$. $[\![P]\!]$ can be computed as $[\![P]\!] = \mathrm{lfp}(T_P) = \bigcup_{i=0}^{\infty} T_P^i(\emptyset)$ where T_P is the immediate consequence operator.

Rather than stating properties of the whole $[\![P]\!]$ we find it more convenient to be able to state properties of the individual predicates which are part of P. For this we define the projection of $[\![P]\!]$ over a predicate p and we denote it $[\![P]\!]_p$ as $\{p(X1, \ldots, Xn) \in [\![P]\!]\}$, i.e., the set of D-atoms in the D-model of the program which correspond to predicate p.

If the program is not correctly written, i.e., it contains bugs, $[\![P]\!]$ will not correspond to our intention. We will denote by $\mathcal{I}$ the intended D-model of the program. Thus, our program will be correct and complete iff $[\![P]\!] = \mathcal{I}$. Similarly $\mathcal{I}_p$ is defined as $\{p(X1, \ldots, Xn) \in \mathcal{I}\}$, i.e., the set of D-atoms in $\mathcal{I}$ which correspond to predicate p.

Assertions for declarative properties allow expressing properties of $\mathcal{I}_p$ for any predicate p of P. These properties can be seen as (partial) specifications of the program predicates. Ideally, it would be desirable to express for a predicate p exactly $\mathcal{I}_p$ (an exact specification), but such specification is often approximated for different reasons. Different kinds of approximations may be used (see Chapter 9for details). In our language we will consider two kinds of assertions which correspond respectively to superset (or correctness) approximations and subset (or completeness) approximations.

8.3.1 Superset (Correctness) Declarative Assertions

The most common kind of declarative assertion concerns correctness. A superset assertion A_p for a predicate p identifies a superset approximation of $\mathcal{I}_p$, i.e., $A_p \supseteq \mathcal{I}_p$. In our assertion language they are written '`:- inmodel` *Pred* `=>` *Cond*', where *Pred* is a normalized D-atom $p(x_1, \ldots, x_n), n \geq 0$ and $x_1, \ldots, x_n$ are distinct variable symbols (as in all the assertions presented in this chapter). It should be interpreted as "any D-atom $p(d_1, \ldots, d_n) \in [\![P]\!]_p$ should satisfy the property *Cond*". For example, the following assertion:

```
:- inmodel qsort(A,B) => list(B).
```

states that it is our intention that the result of ordering a list by means of the predicate `qsort` should be a list.

Superset declarative assertions are used for *correctness* debugging. Consider an assertion A_p for predicate p. If $\exists x \in [\![P]\!]_p$ s.t. $x \notin A_p$ then x is a *symptom* which indicates that the program is not correct. For example, given the above superset assertion if the D-atom `qsort(a,a)` $\in [\![P]\!]_{qsort}$ as `qsort(a,a)` $\notin A_p$ then our program is not correct.

8.3.2 Subset (Completeness) Declarative Assertions

We may also be interested in expressing that some (set of) D-atom(s) must definitely be in the model of a predicate p. A subset assertion A_p for a predicate p identifies a subset approximation of $\mathcal{I}_p$, i.e., $A_p \subseteq \mathcal{I}_p$. In our assertion language they are written '`:- inmodel` *Pred* `<=` *Cond*' (note the reversed direction in the arrow). They should be interpreted as "any D-atom $p(d_1, \ldots, d_n)$ which satisfies *Cond* should be in $[\![P]\!]_p$". For example, the following assertion where the symbol `==` stands for term identity:

```
:- inmodel qsort(A,B) <= (A == [2,1], B == [1,2]).
```

states that `[1,2]` is a correct ordering of the list `[2,1]`. Subset declarative assertions are useful for *completeness* debugging. For example the above assertion can be used to conclude that our program is incomplete if $qsort([2,1],[1,2]) \notin [\![P]\!]_{qsort}$ i.e., the existing code for `qsort` cannot be used to determine that one (in this case the only) result of ordering `[2,1]` is `[1,2]`.

Even though declarative assertions are very useful during debugging and we should exploit them as much as possible, there are properties of programs which lay out of the declarative semantics and which are of interest, for example, during performance debugging. Thus, additional assertions are required for the case of *operational properties*, i.e., those which refer to the way in which the actual computation is performed and thus are only captured using operational semantics. Indeed, the rest of assertions presented in this chapter are mainly related to operational properties. Also, they are all superset approximations.

8.4 Basic Operational Predicate Assertions

Operational predicate assertions[2] are used to express (operational) properties which concern all the invocations of the given predicate during execution of the program. We first illustrate the use of this kind of assertions with an example. Figure 8.2 presents a CIAO program [Bue95] which implements the *quicksort* algorithm together with a series of predicate assertions which express properties which the user expects to hold for the program.[3] Three assertions are given for predicate `qsort/2` (`A1`, `A2`, and `A3`) and two for predicate `partition/4` (`A4` and `A5`). The meaning of the assertions in this example is explained in detail below.

Many features may be expressed in predicate assertions. Different sorts of predicate assertions are used for different features within the execution of the predicate. Also, more than one basic predicate assertion (of the same or different kinds) may be given for the same predicate. In such a case, all of them should hold and composition of basic predicate assertions should be interpreted as their conjunction.

[2]In the remainder of this chapter we will often write "predicate assertions" when referring to "operational predicate assertions".

[3]Both for convenience, i.e., so that the assertions concerning a predicate appear near its definition in the program text, and for historical reasons, i.e., mode declarations in Prolog or entry and trust declarations in PLAI, we write predicate assertions as directives. Depending on the tool different choices could be implemented, including for example separate files or incremental addition of assertions in an interactive environment.

```
:- calls qsort(A,B) : list(A).                              % A1
:- success qsort(A,B) : list(A) => list(B).                 % A2
:- comp qsort(A,B) : (list(A),var(B)) + does_not_fail.      % A3

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

:- calls partition(A,B,C,D) : list(A).                      % A4
:- success partition(A,B,C,D)
                   :  (list(A),ground(B))
                   => (list(C),list(D)).                    % A5

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

Figure 8.2: Predicate assertions

In this section together with each kind of basic predicate assertion we will give a possible translation scheme of assertions into code which will perform run-time checking and will issue a warning message if any of the assertions does not hold. Given a predicate `p(X1,...,Xn)` for which assertions have been given, the idea is to replace the definition of `p(X1,...,Xn)` so that whenever `p/n` is executed the assertions for it are checked and the actual computation originally performed by p/n is now performed by the new predicate p_int. Given the definition of a predicate p/n as

```
p(t11,...,t1n):- body_1.
          ...
p(tm1,...,tmn):- body_m.
```

it gets translated into:

```
p(X1,...,Xn):-
      check_assertions_and_execute ‘‘p_int(X1,...,Xn)’’.

p_int(t11,...,t1n):- body_1.
           ...
p_int(tm1,...,tmn):- body_m.
```

The definition of p_int corresponds to the definition of p/n in the original program and is independent of the assertions given for p/n. The checks present in the new definition of predicate p/n depend on the existing assertions for such predicate. In what follows, $A(p/n)$ represents the set of current assertions for predicate p/n.

8.4.1 Properties of Success States

They are probably one of the most common sorts of properties which we may be interested in expressing about predicates. They are similar in nature to the *post-conditions* used in program verification. They can be expressed in our assertion language using the basic assertion ‘:- success *Pred* => *Postcond*’. It should be interpreted as, “for any call of the form *Pred* which succeeds, on success *Postcond* should also hold” . For example, we can use the following assertion in order to require that the output of a procedure (qsort) for sorting lists be a list:

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

An important thing to note is that in contrast to other programming paradigms, in (C)LP, calls to a predicate may either succeed or fail. The post-condition stated in a success assertion only refer to successful executions.

Success assertions are also relevant because declarative semantics always refers to properties of success states. In fact, any declarative assertion '`:- inmodel` *Pred* `=>` *Cond* can also be interpreted as a success assertion '`:- success` *Pred* `=>` *Cond*' due to correctness of the operational semantics (but not vice versa due to possible incompleteness of the operational semantics).

A possible translation scheme of `success` assertions into run-time tests is: let S be the set $\{Postcond$ s.t. '`:- success` $p(X1,...,Xn)$`=>` $Postcond' \in A(p/n)\}$. Then the translation is

```
p(X1,...,Xn):-
      p_int(X1,...,Xn),
      check(S).
```

Where the definition of predicate `check` is implementation dependent. Next, we give a possible implementation of such predicate. Our aim is not to provide the best possible implementation of the auxiliary predicates for run-time checking but rather to provide examples which show the feasibility of the implementation. In general it will check whether conditions given hold or not. If they hold, computation will generally continue as usual. If they do not, usually a warning will be given to the user. As usual in CLP languages, sets are implemented by means of lists.

```
check([]).
check([Cond|Conds]):-
      call(Cond),!,
      check(Conds).
check([Cond|Conds]):-
      warning(Cond),
      check(Conds).
```

No implementation is presented for the `warning` predicate. In general it will print a message informing about an assertion which does not hold.

8.4.2 Restricting Assertions to a Subset of Calls

Sometimes we are interested in properties which refer not to all invocations of a predicate, but rather to a subset of them. With this aim we allow the addition of preconditions (*Precond*) to predicate assertions as follows: '*Pred* : *Precond*'. For example, `success` assertions can be restricted and we obtain an assertion of the form '`:- success` *Pred* : *Precond* `=>` *Postcond*', which should be interpreted as, "for any call of the form *Pred* for which *Precond* holds, if the call succeeds then on success *Postcond* should also hold". Note that '`:- success` *Pred* `=>` *Postcond*' is equivalent to '`:- success` *Pred* : *true* `=>` *Postcond*'.

Assertions with a precondition are not required to hold for all calls to *Pred*, but only for those of them which satisfy the precondition *Precond*. Thus, in a sense this is a way of weakening predicate assertions. The following assertion (`A2` in Figure 8.2) requires that if `qsort` is called with a list in the first argument position and the call succeeds, then on success the second argument position should also be a list.

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

The difference with respect to the `success` assertion of Section 8.4.1 is that `B` is only expected to be a list on success of predicate `qsort/2` if `A` was a list at the call.

A possible translation scheme for `success` assertions with a precondition is: let RS be the set $\{(Precond, Postcond)$ s.t. '`:- success` $p(X1, ..., Xn)$: $Precond$ `=>` *Postcond*' $\in A(p/n)\}$. Then the translation is

```
p(X1,...,Xn):-
      collect_valid_postconds(RS,S),
      p_int(X1,...,Xn),
      check(S).
```

Where the predicate `collect_valid_postconds/2` collects the postconditions of all pairs in `RS` s.t. the precondition holds. Note that those assertions whose precondition does not hold are directly discarded. A possible implementation of such predicate is given in below.

```
collect_valid_postconds([],[]).
collect_valid_postconds([(Pre,Post)|Conds],PostConds):-
      call(Pre),!,
      PostConds = [Post|PCs],
      collect_valid_postconds(Conds,PCs).
collect_valid_postconds([_|Conds],PostConds):-
      collect_valid_postconds(Conds,PostConds).
```

8.4.3 Properties of Call States

It is also possible to use assertions to describe properties about the calls for a predicate which may appear at run-time. This is useful for at least two reasons. If we perform *Goal-dependent* analysis, (a variation of) `calls` assertions may be used for improving analysis information (see Section 8.7.2). They can also be used to check at run-time whether any of the calls for the predicate is not in the expected set of calls (the "inadmissible" calls of [Nai97]). An assertion of the kind '`:- calls` *Pred* : *Cond*' must be interpreted as "all calls of the form *Pred* should satisfy *Cond*". An example of this kind of assertion is (`A1` in Figure 8.2):

```
:- calls qsort(A,B) : list(A).
```

It expresses that in all calls to predicate `qsort` the first argument should be a list.

A possible translation scheme of `calls` assertions into run-time tests is: let C be the set $\{Cond$ s.t. '`:- calls` $p(X1, ..., Xn)$: $Cond' \in A(p/n)\}$. Then the translation is

```
p(X1,...,Xn):-
      check(C),
      p_int(X1,...,Xn).
```

8.4.4 Properties of the Computation

The predicate assertions previously presented in this section allow expressing properties about the execution state both when the predicate is called and when it terminates its execution with success. However, many other properties which refer to the computation of the predicate (rather than the input-output behaviour) are not expressible. In particular, no property which refers to (a sequence of) intermediate states in the computation of the predicate can be (easily) expressed using `calls` and `success` predicate assertions only. Examples of properties of the computation which we may be interested in are: non-failure, termination, determinacy, non-suspension, etc. In our language this sort of properties are expressed by an assertion of the kind '`:- comp` *Pred* : *Precond* + *Comp-prop*', which is interpreted as "for any call of the form *Pred* for which *Precond* holds, *Comp-prop* should also hold for the computation of *Pred*". Again, the field ':

Precond' is optional. As an example, the following assertion (A3 in Figure 8.2) requires that all calls to predicate qsort with the first argument being a list and the second a variable do not fail.

```
:- comp qsort(A,B) : (list(A) , var(B)) + does_not_fail.
```

Run-time checking of comp assertions is more difficult than that of calls and success assertions. Given a property of the computation *Comp_prop* with n parameters, it is required to define a predicate with the same name *Comp_prop* and $n+1$ arguments. The first argument of the predicate is the run-time instantiation of the call to the predicate to which the *comp* assertions relates (i.e., qsort(A,B) above) and the following n are the parameters of the property. Note that execution of the predicate and checking of the property are both performed (perhaps simultaneously) by this predicate of arity $n+1$. For example, given the property does_not_fail (with 0 parameters) which should be interpreted as "execution of the predicate either succeeds at least once or loops", we can use the following predicate does_not_fail of arity 1 for run-time checking of such property:

```
does_not_fail(Goal):-
    if(call(Goal),
       true,           %% then
       warning(Goal)). %% else
```

In this simple case, implementation of the predicate is not very difficult using the if/3 builtin predicate of SICStus prolog. However, it is not so simple to code predicates which check other properties of the computation and me may need programming meta-interpreters for it. Note however that when the properties are difficult (or even impossible) to code, it may be possible to approximate them. Care must be taken that we always stay on the safe side, i.e., the code for run-time checking may be incomplete (it does not detect that an assertion does not hold), but it must be correct (it only flags that an assertion does not hold if the assertion actually does not hold).

A possible translation scheme of comp assertions into run-time tests is: let *RC* be the set $\{(Prec, Comp\_prop)$ s.t. ':- comp $p(X1,...,Xn)$: $Prec$ + $Comp\_prop \in A(p/n)\}$. Then the translation is

```
p(X1,...,Xn):-
      collect_valid_postconds(RC,C),
      add_arg(C,p_int(X1,...,Xn),C1),
      (C1 == [] ->
        call(p_int(X1,...,Xn)) %% then
      ;
        call_list(C1)).  %% else

call_list([]).
call_list([C|Cs]):- call(C), call_list(Cs).
```

Where the predicate `add_arg/3` adds the goal `p_int(X1,...,Xn)` as the first argument to any property of the computation. A possible implementation is given below.

```
   add_arg([],_,[]).
add_arg([C|Cs],Goal,[NC|NCs]):-
      C=..[Functor|Args],
      NC=..[Functor,Goal|Args],
      add_arg(Cs,Goal,NCs).
```

Note that both `success` and `calls` assertions are in a sense special cases of `comp` assertions as properties of call and success states can also be formalized as properties of the computation. For example consider the following predicates which could be used for checking `calls` and `success` properties at run-time:

```
calls(Goal,Prop):-
      (call(Prop) ->
          true
      ;
          warning(Prop)),
      call(Goal).
```

```
success(Goal,Prop):-
      call(Goal),
      (call(Prop) ->
          true
      ;
          warning(Prop)).
```

the assertion `:- calls p(X) : ground(X)` could be written `:- comp p(X) + calls(ground(X))`. Thus, an assertion language with only the `comp` predicate assertion would suffice. However, `calls` and `success` assertions appear very often in program debugging and their treatment (at least for run-time checking) is much simpler than that of the very general `comp` assertion. Also, in our language of assertions, while conjunction, disjunction, and negation are allowed for properties of the call and success states, only conjunction (but not disjunction nor negation) are allowed in *Comp_prop* properties (see Section 8.9 for details). As a result, it is interesting to have a dedicated predicate assertion for them and only use `comp` assertions when the property is not expressible as `calls` nor `success` assertions.

8.5 Grouping Basic Assertions: Compound Assertions

In this section we introduce another kind of predicate assertions which can be used in addition to the basic ones introduced in Section 8.4, i.e., both basic and compound assertions may be given for a program. The motivation of introducing compound assertions is twofold. On the one hand, usually when more than one `success` (resp. `comp`) assertions are given for the same predicate, the set of `success` (resp. `comp`) assertions are meant to cover all the different uses of the predicate. Thus, the disjunction of the preconditions in all the `success` (resp. `comp` assertions) can usually be seen as a description of the possible calls to the predicate. Thus, it should be desirable that a `calls` assertion is automatically generated for the set of assertions, rather than having to add it manually. Second, a disadvantage of basic assertions as presented in Section 8.4 is that it is often the case that in order to express a series of properties of a predicate, several basic assertions need be written. For this reason and with the aim of making assertion writing not too tedious a task, we propose the use of a compound predicate assertion which can be used as syntactic sugar for the basic assertions. Each compound assertion is translated into 1, 2, or even 3 basic predicate assertions, depending on how many of the fields in the compound assertion are given. The syntax of compound assertions follows. Optional fields are given in square brackets.

`:- pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-prop*].

A compound assertion '`:- pred` *Pred* : *Precond* `=>` *Postcond* + *Comp-prop*' should be interpreted as "for any call of the form *Pred* which satisfies *Precond* the computation of the call should satisfy *Comp-prop* and if the predicate succeed on success of the execution *Postcond* should also hold". As usual, giving no precondition is equivalent to '`pred` *Pred* : *true*'. For example, the following assertion indicates that whenever we call `qsort` with the first argument being a list, the computation should terminate and if the computation succeeds, on termination the second argument should also be a list.

```
:- pred qsort(A,B) : list(A) => list(B) + terminates.
```

Field	Translation if given	Otherwise
`=>` *Postcond*	`success` *Pred* : *Precond* `=>` *Postcond*	$\emptyset$
`+` *Comp-prop*	`comp` *Pred* : *Precond* `+` *Comp-prop*	$\emptyset$

Table 8.1: Transforming compound into basic assertions.

Compound assertions are easily distinguished from basic ones as they always start with the keyword *Pred*, while the latter always start with one of the keywords `calls`, `success`, or `comp`. Table 8.1 presents how a compound assertion is translated into basic `success` and `comp` assertions. Generation of `calls` assertions from compound assertions is more involved. If the set of compound assertions for a predicate *Pred* is $\{A_1, \ldots, A_n\}$ and let $A_i =$ `Pred :` C_i `[=>` S_i`]` `[+` $Comp_i$`]`, then the most accurate `calls` assertion which may be generated is

$$\texttt{calls Pred :} \vee_{i=1}^{n} C_i$$

If only one compound assertion '`:- pred` *Pred* [: *Precond*] [`=>` *Postcond*] [`+` *Comp-prop*]' is given for a predicate, then we can generate the assertion '`:- calls` *Pred* : *Precond*'. If more than a compound assertion for our predicate is given, it is not correct to generate a `calls` assertion for each compound assertion. Several assertions for the same predicate are interpreted as their conjunction (according to Section 8.4), and the correct composition, as discussed above, is their *disjunction*. For example, given the two following compound assertions for predicate `qsort`:

```
:- pred qsort(A,B) : numlist(A) => numlist(B) + terminates.
:- pred qsort(A,B) : intlist(A) => intlist(B) + terminates.
```

The `calls` basic assertion which could should be generated is:

```
:- calls pred qsort(A,B) : (numlist(A) ; intlist(A)).
```

Note that when compound assertions are used, `calls` assertions are always implicitly generated. If we do not want the `calls` assertion to be generated (for example because the set of assertions available does not cover all possible uses of the predicate) basic `success` or `comp` assertions rather than compound (`pred`) assertions should be used.

8.6 Program-Point Assertions

When considering operational semantics of a program, in addition to predicates, we also have the notion of *program-points*. Thus, program-point assertions are inherently operational. Programs are sequences of clauses. Clauses are of the form H`:-`B, where B is a (possibly empty) conjunction of literals. A literal is either a call to a predicate or a constraint. By program point we refer to those places in a program in which a new literal may be added, i.e., before the first literal of a clause (if any), between two literals, and after the last literal of a clause. Program-point assertions are used to express properties which should hold whenever a given program-point is reached during execution of the program. For simplicity, we add assertions of this type to a program by adding a new literal at the corresponding program-point. This literal is a call to the predicate `check`.

Consider the following clause '`p(X):- q(X,Y), r(Y).`' Imagine for example that whenever the clause is reached by execution, after the successful execution of the literal `q(X,Y)`, `X` should be greater than `Y` and `Y` should be positive. This can be expressed by replacing the previous clause by the following one in which a program-point assertion has been added:

```
p(X):-  q(X,Y), check((X>Y,Y>=0)), r(Y).
```

8.6.1 Properties which May Appear in Program-Point Assertions

Two different kinds of properties may appear inside a `check` program-point assertion:

- **state property:** they should hold for the current execution state whenever execution reaches the corresponding program-point. Whatever happens in future computation states does not affect whether the assertion holds or not.

- **forward property:** these properties may not be decidable at the execution state which corresponds to the program point in which they appear. However, they may become decidable at a later state in the execution. Thus, in a way, this kind of properties refer to all the continuation of the execution.

Forward properties are harder to deal with than state properties in that state properties refer to a single computation state and forward properties possibly refer to a sequence of states, which in the worst case may be all the sequences of forward computations. For this reason, we only allow constraints and conjunction of constraints to appear in forward properties. However, for state properties, we can use any property defined as presented in Section 8.2.2, including user-defined programs and expressions built using conjunctions, disjunction and negation.

Checking forward properties correspond to performing an entailment check. If the property is entailed by the current store, then the assertion is true. If the current store entails the negation of the property then the assertion is false. As entailment is monotonic, if a forward property is true with a constraint store S it will also be so in any state with a constraint store S' reached in forward computation as $S' \rightarrow S$. This property can be exploited when performing run-time checking, as once an assertion is true, there is no need to check it again and it can be discarded.

An additional difficulty posed by forward properties is that they may become false (and thus a warning should be given) in an execution state which is not related to the program point at which the assertion appears. Thus, it may not be easy to identify which assertion with a forward property does not hold. For this reason, we add to program-point assertions with forward properties the field `Action` which stores a call to a procedure which is in charge of giving relevant information to the user about the assertion which does not hold. Thus, the syntax of forward properties is `fwd(Cond, Action)`. This also allows easily identifying forward conditions.

8.6.2 Execution of Program-Point Assertions

An important difference between program-point assertions and predicate assertions is that while the latter are not part of the program, program-point assertions are, as they have been introduced as new literals in some program clauses. In order to avoid program-point assertions from changing the answers of the program, we assume that the predicate `check/1` is defined as

```
check(_Prop).
```

i.e., any call to `check` trivially succeeds. This definition is overridden by the clause

```
check(Property):- call(Property).
```

when run-time checking is being performed. Note that if `Property` is `fwd(Property,Action)`, i.e., a forward property, for the execution of `Property` we need a definition of the predicate `fwd/2`. This definition is not given because it is system dependent. It can be based on delay declarations such as `when/2` or on conditionals such as `if_then_else` in CHIP.

Three cases are possible when a forward property is checked at run-time. If S entails $Cond$ then $Cond$ is satisfied. $Cond$ is not satisfied if S entails the negation of $Cond$. If S does not entail $Cond$ nor its negation, i.e., $Cond$ is consistent, but not implied by the constraint store S, then no conclusion about $Cond$ is taken in the current state and $Cond$ will be repeatedly checked in other computation states reached in forward execution.

Note that entailment check may not be complete in a given implementation of a CLP language. This means that in some cases where an assertion with a forward property does not hold (and for example a warning should be given), the system is not capable of detecting it. However, correctness of run-time checking is not an issue. If a warning is produced then the assertion is definitely false.

8.6.3 Example of Forward Property

Consider the following program fragment for which run-time checking is being performed:

```
p(X):-
      check(fwd( (X > 0), format("X not greater than 0"))),
      r(X).
r(X):-
      X > 1.
```

If we execute `p(X)` with X a finite domain variable whose domain is initially $\{-3, -2, -1, 0, 1, 2, 3\}$, when execution reaches the literal `check(...)`, the property is not decidable yet. However, during the execution of `r(X)` the domain of `X` becomes $\{2, 3\}$. Now the property becomes decidable and the assertion holds.

8.7 Assertions in Program Analysis (Actual Properties)

As opposed to all the assertions discussed in previous sections, which express expected properties of the program if it were correct (intended properties), as seen in Chapter 4, during the process of program development, validation and debugging we are often interested in expressing properties of the actual program at hand (actual properties), which may or may not satisfy the requirements. Thus, we have to distinguish between properties which we would like the final program to satisfy and properties of the actual program at hand. This greatly facilitates communicating different modules which use analysis information, reusing information and communication to/from the user.

This is achieved by simply adding in front of the assertion a tag which clearly identifies whether the property expressed by the assertion should hold in the final program or it actually holds for the program at hand. Three different types of tags are considered

`check` They are used to mark the corresponding assertion as expressing an expected property of the final program (intended property).

`true` They indicate that the property holds for the program at hand (actual property).

`trust` The property holds for the program at hand (actual property). The difference with the above is that this information is given by the user and it

```
:- entry qsort(A,B) : numlist(A).
:- true pred qsort(A,B) : numlist(A)
                       => (numlist(A),numlist(B)).

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

:- true pred partition(A,B,C,D)
                      :  (numlist(A),number(B),var([C,D]))
                      => (numlist(A),number(B),numlist(C),numlist(D)).

partition([],B,[],[]).
partition([E|R],C,[E|Left1],Right) :-
        true((ground([C,E,R]),var([Left1,Right]))),
        E<C, !,
        true((ground([C,E,R]),var([Left1,Right]))),
        partition(R,C,Left1,Right),
        true(ground([C,E,Left1,R,Right])).
partition([E|R],C,Left,[E|Right1]) :-
        true((ground([C,E,R]),var([Left,Right1]))),
        E>=C,
        true((ground([C,E,R]),var([Left,Right1]))),
        partition(R,C,Left,Right1),
        true(ground([C,E,Left,R,Right1])).
```

Figure 8.3: Analysis results expressed as assertions

may not be possible to infer it automatically.

Note that all the assertions presented in Chapter 4 have the tag `trust`.[4] Note also that the assertions presented previously in this chapter refer to intended properties. Thus, they should have the tag `check`. However, for pragmatic reasons, the tag `check` is considered optional and if no tag is given, `check` is assumed by default. For example the assertion `:- check success p(X) : ground(X)` can also be written `:- success p(X) : ground(X)`. Regarding the program-point assertions introduced in Section 8.6, they already have the keyword `check`. When we want to mark them as `true` or `trust` we simply replace `check` by the corresponding tag (see Section 8.9 for syntactic details). Examples of program-point `true` assertions can be seen in Figure 8.3.

Sometimes it is possible to compute (approximate) at compile-time properties about the run-time behaviour of a program. This process is in general tedious and automatic analysis techniques have long been used for this task. Assertions can be used for expressing the results of analysis. In this context, the assertions express properties which the program at hand satisfies.

Predicate and/or program-point assertions may be generated according to the user's choice. Figure 8.3 presents the same program as in Figure 8.2 but rather than with `check` predicate assertions, with both predicate and program-point `true` assertions which express analysis results. The results have been generated by goal-dependent type analysis. The role of the `entry` assertion is discussed in Section 8.7.2 below. Program-point assertions contain information for each program point and are literals of the `true/1` predicate. Regarding predicate assertions, for conciseness compound rather than basic predicate assertions are usually produced by the analyzer. They follow the structure of the compound assertions of Section 8.5 and have the following syntax:

`:- true pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-prop*].

8.7.1 Aiding the Analysis

Yet another kind of assertions have been introduced in Chapter 4 and are intended for use when additional information is to be provided to the analyzer in order to improve its information. There, compound assertions, as introduced in

[4] Except for `entry` assertions. The difference w.r.t. `trust` assertions is discussed in Section 8.7.2.

Section 8.5, are used. However, as mentioned above, the tag `trust` may be added to any predicate assertion (including the basic ones). An example of this kind of assertions is:

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

which states that upon success `B` is a list provided that `A` was a list on call. Note that the assertion:

```
:- check success qsort(A,B) : list(A) => list(B).
```

(where the `check` tag is optional) states that under the same conditions, `B` *should be* a list if the program were correct, while the `trust` assertion states that `B` is indeed a list.

Though similar in nature to `true` assertions, as they both refer to properties of the actual program, the main difference between them is that while `true` assertions have been generated by analysis and are automatically provable from the program at hand, `trust` assertions are often not provable (either because part of the program is not available or because analysis is not powerful enough) but the analysis is instructed to trust such assertions. When performing global analysis, a `trust` assertion for a predicate p may improve the analysis information for the predicate p if the information it contains is better than that generated by analysis. In that case it may also improve the analysis information of any other predicate p' which depends on p, i.e., p' calls p w.r.t. the analysis information available if the `trust` assertion were ignored.

Note that if analysis is *goal-dependent* (see below), the existence of `trust` assertions for a predicate does not avoid analyzing the code of the corresponding code if it is available, as otherwise the internal calls generated in this predicate could be ignored during analysis resulting in incorrect analysis information. Only after analysis of such a predicate may `trust` assertions be used to improve the analysis information obtained. Note also that if the code of the predicate is not available, the internal calls to predicates in the program that may appear during execution of the missing predicate must have been declared in `entry` assertions for soundness of the analysis. Refer to Chapter 4 for details.

It is important to mention that even though `trust` assertions are trusted by the analyzer to improve its information unless they are incompatible with the information generated by the analyzer (see Chapter 4), they may also be subject to run-time checking. The translation scheme for assertions with the tag `trust` is exactly the same as the one given in Sections 8.4 and 8.6 for assertions with the tag `check`. This should be an option when translating a program into another one with run-time tests, whether only `check` assertions or both `check` and `trust` assertions should be checked at run-time.

8.7.2 Goal-Dependent analysis

Goal-dependent analyses are characterized by generating information which is valid only for a restricted set of calls, rather than for any possible call to the predicate, as opposed to goal-independent analyses whose results are valid for any call to the predicate. As goal-dependent analyses allow obtaining results which are *specialized* (restricted) to a given context, they provide in general better (stronger) results than goal-independent analyses.

In order to improve the accuracy of goal-dependent analyses, some kind of description of the *initial* calls to the program should be given[5] With this aim, `entry` declarations have been introduced in Chapter 4. Their role is to restrict the starting points of analysis to only those calls which satisfy the assertion '`:- entry` *Pred* : *Precond*'. For example, the following assertion informs the analyzer that at run-time all initial calls to the predicate `qsort/2` have a list of numbers in the first argument position:

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

The possibly more accurate information generated by a goal-dependent analyzer using the above assertion is valid for any execution of `qsort/2` with the first argument being a numeric list, but may be incorrect for other executions.

Both `entry` and `trust` assertions have in common that they are provided by the user and are assumed to be true. Thus, we may tend to think that the assertion '`:- entry` *Pred* : *Precond*' is syntactic sugar for '`:- trust calls` *Pred* : *Precond*'. However, there is a subtle difference between the two sorts of assertions above. '`:- entry` *Pred* : *Precond*' states that *Precond* holds (only) for

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

all the initial calls to *Pred*, while '`:- trust calls` *Pred* : *Precond*' states that *Precond* holds for all (both initial and internal) calls to *Pred*.

Thus, `entry` assertions allow providing more precise descriptions of initial calls (as the properties expressed do not need to hold for the internal calls) and also they are easier to provide by the user (which does not need to understand the internal behaviour of the program). This is possible because goal-dependent analysis is capable of automatically computing (approximating) a description of all the internal calls. Consider for example the following program with an entry assertion.

```
:- entry p(A) : ground(A).
p(a).
p(X):- p(Y).
```

If instead of the entry we had written '`:- trust calls p(A) : ground(A)`' then such assertion would be incorrect. For example the execution of `p(b)` produces calls to `p` with the argument being a free variable.

`entry` assertions may also be checked at run-time. As mentioned in Section 8.7.1, this should be an option of the compiler when introducing run-time tests in the program. The translation scheme is analogous to that performed for `calls` assertions but is only applied to initial calls to the program.

8.8 Compile-time Checking of Assertions

The traditional approach to the use of `check` assertions is to check them at run-time. As we have seen in previous sections, there is a great similarity between the `check` assertions and the analysis information, which can also be expressed as assertions.

An idea is then to use global program analysis to anticipate the results of run-time checking of assertions. Of course, not always the analysis information allows determining whether a given `check` assertion holds at run-time or not. If a `check` assertion is proved not to hold, then we have detected a *symptom* which indicates that the program is not correct and should be diagnosed. An important thing to mention is that this not only allows finding bugs, but also it is possible to do it at compile-time, and without having to run the program. This may be very important in cases in which either there are not many test cases or running the program may be very expensive.

If the assertion is proved to hold, then there is no need to check whether it holds or not, with the associated improvement in run-time performance. Note that if the analysis is goal-dependent it may be possible to ensure that a (set of) `calls` assertion for a predicate hold. However, if the analysis is goal-independent it is not possible in general.

Another reason why compile-time checking of assertions is important is the existence of properties of the computation which are not checkable by means of run-time tests, but which could be proved (or disproved) at compile-time by program analysis. An example of this is the property *terminates*. Clearly, this property is semi-decidable, but not decidable in general and it is not possible to introduce simple run-time tests which give a warning if it does not hold.

For the case of `success` and `comp` assertions, they can be proved to hold both with goal-independent and goal-dependent analysis. However, if the analysis is goal-dependent, two sub-cases may be distinguished:

- *Postcond* (resp. `Comp-prop`) holds for any call which satisfies *Precond*, or
- *Postcond* (resp. `Comp-prop`) holds for any call which analysis has computed (approximated) as possible during run-time and such call satisfies *Precond*.

The first case is less problematic and the `check` assertion may be directly converted into a `true` assertion. In the second case it is not possible to transform the `check` into a `true` assertion as the assertion holds in our context, i.e., for the class of calls described by our `entry` assertion as seen in Section 8.7.2. However, if we would like to analyze the program for a new `entry` assertion, it could not be used in the same way as the `true` assertions generated by the analyzer, which hold for any additional `entry` assertion.

8.9 Syntax of Assertions

In this section the grammar of the assertion language is presented for reference.

declarative-assert	::=	`:- inmodel` *pred approx state-prop-exp*
approx	::=	`=>`

```
                    | <=
predicate-assert  ::=  :- eflag pred-assert
pred-assert       ::=  calls
                    | success
                    | comp
calls             ::=  calls pred
                    | calls pred : state-prop-exp
success           ::=  success pred => state-prop-exp
                    | success pred : state-prop-exp => state-prop-exp
pred              ::=  pred-name
                    | Pred-name(args)
args              ::=  Var
                    | Var, args
comp              ::=  comp pred + comp-exp
                    | comp pred : state-prop-exp + comp-exp
state-prop-exp    ::=  state-prop-exp , state-prop-exp
                    | state-prop-exp ; state-prop-exp
                    | \+(state-prop-exp)
                    | State-prop
comp-exp          ::=  comp-exp , comp-exp
                    | Comp-prop
eflag             ::=  check
                    | true
                    | trust
                    | ϵ
literal-assert    ::=  flag ( Goal , comp-exp )
flag              ::=  check
                    | true
                    | trust
prog-point-assert ::=  flag ( prog-point-cond )
prog-point-cond   ::=  state-prop-exp
                    | fwd-cond
fwd-cond          ::=  fwd( fcond , Action )
fcond             ::=  Constraint
```

compound	::=	*eflag Pred precond postcond computation*
precond	::=	`:` *state-prop-exp*
		| ϵ
postcond	::=	`=>` *state-prop-exp*
		| ϵ
computation	::=	`+` *comp-exp*
		| ϵ

There are some non-terminals in the grammar which are not defined. This is because they are constraint-domain and/or platform dependent. They can be easily distinguished in the previous grammar because their name starts with a capital letter:

Pred-name As we are interested in having an assertion language which looks homogeneous with the CLP language used, we admit as *Pred-name* any valid name for a predicate in the underlying CLP language. Usually, non-empty strings of characters which start with a lower-case letter.

Var It corresponds to the syntax for variables in the CLP language. Usually, non-empty strings of characters which start with a capital letter.

State-prop As seen in Section 8.2.2, as built-in predicates and user programs may be used as *State-prop*, their syntax depends on the syntax of the language and the set of built-ins of the system.

Comp-prop They correspond to the properties of predicate computation introduced in Section 8.4.4.

Goal It corresponds to the syntax for goals in the CLP language. Usually, non-empty strings of characters which start with a lower-case letter.

Constraint Their syntax depend on the language and the particular implementation.

Action It is a call to a given procedure which will handle assertions which do not hold. Additionally they are assumed to succeed in finite time.

8.10 Chapter Conclusions

Assertions allow expressing properties of programs. Many kinds of properties may be expressed and assertions may be used in many different contexts. Thus, many kinds of assertions exist. We have proposed an assertion language for constraint logic programs which is general in that it can be used in many different contexts and for different purposes and which is as uniform as possible. This allows communication among the user and the different tools and facilitates the implementation of an integrated development and debugging environment in which different tools co-exist. Also, the high-level nature of constraint logic programming allow expressing properties in the underlying programming language. This may contribute to encourage the programmer to actually write and use specifications as this can be done in a language the programmer is familiar with.

In order to show the feasibility of run-time checking and to help clarify the semantics of assertions, a scheme for run-time checking is presented which given a program and a (partial) specification given in the proposed language of assertions produces a new program which when run may detect that some assertions do not hold. In such case a warning is given to the user. Diagnosis should then be performed in order to detect and correct the error in the program.

Chapter 9

The Role of Semantic Approximations in Program Debugging

During program development and debugging, several tools may be used which deal with (approximations of the) program semantics in one way or the other. Example of such tools are automatic validation tools, declarative debuggers, program analyzers, etc. These tools also have in common that both the semantics of the current program and the semantics of the program we aim at achieving are often compared. In the work presented in this chapter we give a uniform formulation of the "problems" which have to be dealt with in a wide set of such tools. The formulation is very general and it is only assumed that the program semantics used is in the class of fixpoint semantics. It is then studied the effect of using approximations rather than the exact program semantics when solving such problems. The case of using abstract interpretation in order to approximate program semantics is studied in detail. Finally, we propose an advanced architecture of tools for program development and debugging capable of dealing with approximations of program semantics. The (approximations) of program semantics in such environment will be expressed using the assertion language proposed in Chapter 8.

9.1 Introduction

A central problem in program development is obtaining a program which satisfies the user's expectations. When considering a given program, a natural question is then whether or not it fulfills expectations of some kind (requirements). To be able to formulate this question, some formal or informal way of specifying such requirements is needed. That is, a (formal or informal) program *semantics* is needed, in which what the program computes and what it is required to compute can be expressed.

It may then be possible either to *verify* that the program satisfies the requirement for every computation (in the considered class), or to show a specific computation where the requirement is violated. The process of identifying the part of the program responsible for the violation is referred to as *diagnosis*. The program then needs to be modified to correct the error. Since the requirement documentation is often not complete, the user's requirements are often given as *approximations*, i.e., safe specifications of (parts of) the intended semantics of the program. The process of *debugging* consists of the study of the program semantics, observation of error symptoms, localization of program "errors" and their correction until no symptom can be observed anymore and the program is considered correct.

Semantic approximations have been used in program validation, in declarative diagnosis, and in program analysis. This chapter gives a common view of these techniques from the perspective of debugging. The objective is to explore possible uses of approximations for debugging purposes. The presentation is organized as follows. First, some notions on program semantics are given in Section 9.2, mainly by means of examples. Then, validation problems, diagnosis by proofs, and declarative diagnosis are described in terms of set-theoretic relations in Section 9.3. Next, the effect of using approximations rather than the exact sets for the intended semantics is studied in Section 9.4. In Section 9.5 such relations on set approximations are reformulated for the special case of abstract interpretation. Finally, Section 9.6 proposes an advanced architecture of tools for program validation and debugging.

We keep the basic discussion quite general, in that we only impose some restrictions on the way the different semantics are formalized. We illustrate the general discussion by very simple examples referring to Constraint Logic Programming (CLP) [JM94].

9.2 Actual and Intended Semantics

Semantics associate a *meaning* to a given syntax (generally of a program). A particular semantics captures some features of the computations of a program (sometimes called the "observables") while hiding others. Different kinds of semantics can be used depending of the features to be described.

In this chapter we restrict ourselves to the important class of semantics referred to as *fixpoint semantics*. In this approach a (monotonic) semantic operator (which we refer to as S_P) is associated with each program P. This S_P function operates on a semantic domain which is generally assumed to be a complete lattice or, more generally, a chain complete partial order. The meaning of the program (which we refer to as $[\![P]\!]$) is defined as the least fixpoint of the S_P operator, i.e., $[\![P]\!] = \text{lfp}(S_P)$. A well-known result is that if S_P is continuous, the least fixpoint is the limit of an iterative process involving at most ω applications of S_P and starting from the bottom element of the lattice.

Example 9.2.1 An example of a set-based, fixpoint semantics for (constraint) logic programs is the traditional least model semantics [JM94], briefly discussed in Section 8.3. The semantic objects in this case are so called D-atoms. A D-atom is an expression $p(d_1, \ldots, d_n)$ where p is an n-ary predicate symbol, $d_1, \ldots, d_n \in D$ and D is the domain of values. For example, in classical logic programming D is the Herbrand universe; for CLP(R) D is the set of real numbers and of terms (for example lists) containing real numbers[1].

The semantic operator for program P is T_P (the immediate consequence operator) and $[\![P]\!] = \text{lfp}(T_P) = \bigcup_{i=0}^{\infty} T_P^i(\emptyset)$. An important property is that $[\![P]\!]$ is the least D-model of the program. Any ground instance[2] of a computed answer

[1] Usually it is assumed that D is given together with a fixed interpretation of the symbols that can occur in constraints. For instance for CLP(R), $+$ is interpreted as addition and $>$ as the "greater than" relation on reals.

[2] In CLP, by a ground instance of a constrained atom $A \leftarrow c$ we mean any D-atom $A\theta$ such that $c\theta$ is true; here A is an atom, c a constraint and θ is a valuation assigning elements of D to variables.

(for an atomic query) is a member of $[\![P]\!]$.

For example, given the following CLP program, over the domain of integers:

```
sorted([]).
sorted([Y]).
sorted([H1,H2|T2]) :- H1 > H2, sorted([H2|T2).
```

we have that $[\![P]\!] = \{\,sorted([\,])\,\} \cup \{\,sorted([X]) \mid X \in D\,\} \cup \{\,sorted([X_1,\ldots,X_n]) \mid n \geq 2,\ X_1 > \cdots > X_n\,\}$. So for instance $[\![P]\!]$ contains $sorted([7])$, $sorted([a])$, $sorted([[\,]])$, $sorted([2,1,0])$ and does not contain $sorted([0,2])$, $sorted([2,1,a])$.

Example 9.2.2 Another example of a fixpoint semantics is the traditional "call-answer operational semantics" for CLP programs (see, e.g., [GHB$^+$96]). The semantic objects in this case are pairs of constrained atoms. The program is assumed to contain a query or "entry point". $[\![P]\!]$ contains all the call-answer pairs that appear during program execution for the given query or entry point. For example, given the CLP program above and the query "$\leftarrow X = [1,Y], sorted(X)$", and, assuming standard left-to-right, depth-first control, we have $[\![P]\!] = \{(sorted(X) \leftarrow X = [1,Y], sorted(X) \leftarrow X = [1,Y] \wedge Y < 1), (sorted(X) \leftarrow X = [Y], sorted(X) \leftarrow X = [Y])\}$.

Both program validation and diagnosis, to be discussed more precisely later, compare the *actual semantics* of the program, i.e., $[\![P]\!]$, with an *intended semantics* for the same program. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. The nature of the requirements considered in validation and diagnosis is very wide. For example, one can discuss *declarative* diagnosis/validation (when the requirements concern the relation specified by the program), diagnosis/validation of *dynamic properties* (when the requirements concern properties of the execution states), *performance* diagnosis/validation (when the requirements concern the efficiency of execution), etc. Thus, different kinds of user's expectations require different kinds of semantics in order to be able both to adequately express the requirements and to extract relevant meaning from the program to compare with the requirements.

Example 9.2.3 In CLP, requirements regarding characteristics of the computed answers of a program can in general be expressed and checked using the least D-model semantics of Example 9.2.1, whereas if the requirements also refer to characteristics of the calls that occur during execution then the operational semantics of Example 9.2.2 (using sets of pairs of constrained atoms) would need to be used.

We focus here on the common case in which the actual semantics $[\![P]\!]$ of a program corresponds to a set and the semantic domain is the lattice of sets with ordering being set inclusion. A natural question is thus how the user's intention can be represented. For the time being, let us assume that $\mathcal{I}$ belongs to the same semantic domain used for $[\![P]\!]$. The semantic object $\mathcal{I}$ can be seen as the corresponding semantics of an intended program. But this program does not exist (neither as program, nor in mind) in general. Thus, usually there is no expression of $\mathcal{I}$, but rather partial descriptions of it.

Example 9.2.4 If the program of Example 9.2.1 is intended to compute all integer lists that are sorted, the programmer can approximate this intention with:

$$\mathcal{I}_1 = \{sorted([X]) \mid X \text{ is an integer}\}$$

$$\mathcal{I}_2 = \{sorted(L) \mid L \text{ is an integer list}\}$$

Obviously, $\mathcal{I}_1$ represents a subset of the programmer's intention, since it represents only sorted integer lists of length one. Similarly, $\mathcal{I}_2$ represents a superset of the programmer's intention; it does not require that the lists are sorted.

9.3 Validation and Diagnosis in a Set Theoretic Framework

This section summarizes well-known notions related to program validation (see, e.g., [Der93, DM93]), diagnosis by proof, and declarative diagnosis [Sha82, Fer87]. The problems found in these disciplines are summarized and discussed in a set theoretic framework for clarity. They can also be formulated in a lattice theoretic setting, but the set theoretic presentation simplifies the discussion.

9.3.1 Validation

Validation aims at proving certain properties of a program which are formally defined as relationships between a specification $\mathcal{I}$ and the actual program semantics $[\![P]\!]$. Table 9.1 lists validation problems in a set theoretic formulation.

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

Table 9.1: Set theoretic formulation of validation problems

Note that we do not assume that $\mathcal{I}$ is unique. We simply denote specifications as $\mathcal{I}$, but it can very well be the case that different specifications are given for verifying different properties. In particular, when dealing with partial correctness, $\mathcal{I}$ describes a property which should be satisfied by all elements of the semantics $[\![P]\!]$. In other words, $\mathcal{I}$ corresponds to expected properties of all results or all behaviours of the program (depending of the kind of semantics). When dealing with completeness $\mathcal{I}$ characterizes a set of elements which should be in the semantics $[\![P]\!]$, i.e., $\mathcal{I}$ describes some expected results or behaviours of P. Proving incorrectness and incompleteness is also of interest, as it indicates that the program does not satisfy the specifications and diagnosis of incorrectness or incompleteness should be performed.

9.3.2 Diagnosis by Proof

The existing proof methods for correctness and completenes are usually based on some kind of induction. Table 9.2 presents well-known sufficient conditions which can be used for program verification and diagnosis.

In the table (*) stands for an additional requirement. The sufficient condition for completeness of P w.r.t. $\mathcal{I}$, requires not only co-inductiveness of $\mathcal{I}$ for P but also that S_P has a unique fixpoint. This last condition holds for a large class of programs (e.g., the acceptable programs in [AP93]).

Property	Definition	Implies
$\mathcal{I}$ inductive for P	$S_P(\mathcal{I}) \subseteq \mathcal{I}$	P partially correct w.r.t. $\mathcal{I}$
$\mathcal{I}$ co-inductive for P	$\mathcal{I} \subseteq S_P(\mathcal{I})$	P complete* w.r.t. $\mathcal{I}$
$\mathcal{I}$ not inductive for P	$S_P(\mathcal{I}) - \mathcal{I} \neq \emptyset$	
$\mathcal{I}$ not co-inductive for P	$\mathcal{I} - S_P(\mathcal{I}) \neq \emptyset$	

Table 9.2: Set theoretic formulation of diagnosis by proof problems

Failures in proving the conditions may possibly indicate that the program has an error. An *incorrectness error* is a part of the program that is the reason for $S_P(\mathcal{I}) - \mathcal{I} \neq \emptyset$. An *incompleteness error* is a part of the program that is the reason for $\mathcal{I} - S_P(\mathcal{I}) \neq \emptyset$. The operator S_P in any kind of semantics is defined in terms of the constructs of the program P. Thus, it makes it possible to define precisely what is meant by the informal statement "is the reason". For CLP programs, an incorrectness error is a program clause and an incompleteness error is a program procedure (a set of the clauses defining a certain predicate symbol).

If the program is incorrect or incomplete, then it includes a corresponding error. One can try to make a proof that $\mathcal{I}$ is inductive (or co-inductive) w.r.t. the program. For an incorrect or incomplete program some constructs will be identified where the corresponding conditions cannot be proved. These constructs are possible error locations. As the conditions presented in Table 9.2 are not necessary, a fragment of the program localized as erroneous may or may not correspond to a bug in the program.

Example 9.3.1 We show two examples for which a proof of partial correctness is impossible. In both cases the specification is not inductive for the program. In the first case the program is incorrect w.r.t. the specification. In the second, the program is correct but a correctness error is detected because of a too weak specification. The operator S_P is the immediate consequence operator T_P for logic programs.

Consider the program P from Example 9.2.1 and the specification $\mathcal{I}_2$ from Example 9.2.4 (so the arguments of *sorted* are required to be integer lists). An attempted correctness proof fails, $\mathcal{I}_2$ is not inductive w.r.t. P, the reason is the clause $sorted(X) \leftarrow X = [Y]$, as $sorted([a]) \in S_P(I_2)$ and $sorted([a]) \notin \mathcal{I}_2$. This clause is also the reason that the program is not partially correct w.r.t. $\mathcal{I}_2$.

Consider the following CLP program Q, over the domain of integers. It is basically the program from Example 9.2.1 in which the new predicate *order/2* has been added.

```
sorted([]).
sorted([Y]) :- Y > Z.
sorted([H1,H2|T2]) :- order(H1,H2), sorted([H2|T2]).
order(X,Y)  :- X > Y.
```

Assume that a partial specification requires the argument of *sorted/1* to be a list of integers. Nothing is required about predicate *order/2*. This means that, in our set-theoretical setting, $\mathcal{I}$ contains all the D-atoms of the form $order(X, Y)$ $(X, Y \in D)$ and all the atoms of the form $sorted(L)$, where L is a list of integers. Notice that Q is correct w.r.t. $\mathcal{I}$. However, $\mathcal{I}$ is not inductive w.r.t. Q (as $S_Q(\mathcal{I})$ contains for instance $sorted([a, 1])$). The second clause is the reason. Strengthening the specification for *order/2* is necessary to obtain a correctness proof. We add a requirement that both arguments of *order/2* are integers and obtain $\mathcal{I}'$, which is inductive w.r.t. Q.

Note that the situation of weak correctness requirements presented above is equivalent to having an incomplete but correct program which presents a correctness error using conditions of Table 9.2 (or vice versa). However, the experience with type checking of logic programs (see, e.g., [AM94, HL94]) shows that failure in proving local validation conditions for a clause is often a good indication that the clause is erroneous.

9.3.3 Declarative Diagnosis

In contrast to diagnosis by proof, the declarative diagnosis concerns the case when a particular (test) computation does not satisfy a requirement.

We learn that a program P is incorrect (i.e., not partially correct w.r.t. $\mathcal{I}$) when we find out that it produces a result x such that $x \not\subseteq \mathcal{I}$. Such a result x is called an *incorrectness symptom*. Similarly, a program P is incomplete when it does not produce some expected result, in other words when there exists some $x \subseteq \mathcal{I}$ such that $x \not\subseteq [\![P]\!]$. Such x is called an *incompleteness symptom*.

Example 9.3.2 In the program of Example 9.2.1 with the specifications of Example 9.2.4, note that $sorted([a]) \in [\![P]\!]$ but $sorted([a]) \notin \mathcal{I}_2$. Therefore, such an atom is an incorrectness symptom w.r.t. $\mathcal{I}_2$. If in that program the first clause was missing then $sorted([]) \in I_1$ would be an incompleteness symptom w.r.t. $\mathcal{I}_1$, since, without that clause, $sorted([]) \notin [\![P]\!]$.

Briefly, declarative diagnosis starts with a symptom of incorrectness (resp. insufficiency) and aims at localizing an erroneous fragment of the program. A declarative diagnoser localizes an error by comparing elements of the actual semantics involved in computation of the symptom at hand with user's expectations. The diagnoser will re-explore computations of symptoms obtained w.r.t. $\mathcal{I}$, and identify errors related to such symptoms, i.e., parts of the program which explain why $S_P(\mathcal{I}) \not\subseteq \mathcal{I}$ (resp. $\mathcal{I} \not\subseteq S_P(\mathcal{I})$). The erroneous fragment of the program localized in that way depends on the nature of S_P.

Example 9.3.3 Consider (constraint) logic programming and its logical semantics. So $\mathcal{I}$ and $[\![P]\!]$ are interpretations over some domain. In the case of incorrectness, if there exists an x s.t. $x \in S_P(\mathcal{I})$ and $x \notin \mathcal{I}$ then there exists a clause $H \leftarrow B$ of the program P which is not valid in $\mathcal{I}$ (for some valuation, H is false and B is true). It can be proved that an incorrectness diagnoser finds such a erroneous clause for any incorrectness symptom. In the program of Example 9.2.1, with $\mathcal{I}_2$ as in Example 9.2.4, we have:

$$\begin{aligned} T_P(\mathcal{I}_2) \;&= \{sorted([\,])\} \cup \\ &\{sorted([X]) \mid X \in D\} \cup \\ &\{sorted([X, Y|L]) \mid X, Y \text{are integers, } X > Y,\ [Y|L] \text{ is an integer list } \} \end{aligned}$$

in which $sorted([a])$ is included. The clause responsible for this symptom is the second one in the program.

In the case of incompleteness, if there exists an y s.t. $y \in \mathcal{I}$ and $y \notin S_P(\mathcal{I})$ then for each clause $H \leftarrow B$ of P if y is a value of H under some valuation ν (an instance of H) then $\nu(B)$ is false in $\mathcal{I}$. So the erroneous fragment found in this case is a set of clauses (which begin with the same predicate symbol).

In the process of diagnosing, the actual semantics of the program $[\![P]\!]$ is compared with the user's expectations $\mathcal{I}$. This is achieved by asking queries about elements of both $[\![P]\!]$ and $\mathcal{I}$ to an oracle. In practice the oracle is usually the programmer, although an executable specification may also be used (we will come back to this issue later).

Three families of queries are considered: one used in incorrectness error search and two used in incompleteness error search. A *universal query* asks whether a given subset Q of $[\![P]\!]$ is correct w.r.t. $\mathcal{I}$ (i.e. whether $Q \subseteq \mathcal{I}$). In the case of CLP, where $\mathcal{I}$ is a set of D-atoms, Q is usually the set of ground instances of a given constrained atom. An example universal query is:

Is $sorted([X, 1]) \leftarrow X > 2$ correct?

The answer is YES, assuming that $\mathcal{I} = \{sorted(L) \mid L \text{ is a sorted integer list}\}$. Under the same assumption, the answer to the universal query about $sorted([X, 1]) \leftarrow X \geq 0$ is NO.

An *existential query* asks whether a given set Q has an element in $\mathcal{I}$ (i.e. whether $Q \cap \mathcal{I} \neq \emptyset$). If Q is the set of ground instances of a constrained atom $A \leftarrow C$, then $Q \cap \mathcal{I} \neq \emptyset$ is equivalent to satisfiability of the formula $C \wedge A$ in the interpretation $\mathcal{I}$. Here is an example existential query (in which the constraint C is empty):

Is $sorted([X, Y])$ satisfiable?

A *covering query* asks if a given set Q' contains all the elements of a given set Q that are in $\mathcal{I}$ (so it asks whether $Q \cap \mathcal{I} \subseteq Q'$). It is a generalization of an existential query (when $Q' = \emptyset$). An example:

Do $\{sorted([2, 1]), sorted([3, 1])\}$ cover all correct instances of $sorted([X, 1]) \leftarrow X < 4$?

Table 9.3 shows for all possible pairs of query/answer used in a declarative diagnoser the corresponding problem in a set theoretic setting.

9.4 Approximating the Intended Semantics

Using the exact intended semantics for automatic validation and diagnosis is in general not realistic, since the exact semantics can be only partially known, infinite, too expensive to compute, etc. In this section we consider the debugging

Query	Answer	Definition
Universal	yes	$Q \subseteq \mathcal{I}$
	no	$Q \not\subseteq \mathcal{I}$
Existential	yes	$Q \cap \mathcal{I} \neq \emptyset$
	no	$Q \cap \mathcal{I} = \emptyset$
Covering	yes	$(Q \cap \mathcal{I}) \subseteq Q'$
	no	$(Q \cap \mathcal{I}) \not\subseteq Q'$

Table 9.3: Set theoretic formulation of problems in a declarative diagnoser

process in terms of *approximations* of the intended semantics. Approximations of the actual program semantics will be considered in the following sections.

An over-approximation of a value A (a "superset" if the semantic domain consist of sets), denoted A^+, satisfies $A \subseteq A^+$. Similarly, two other types of approximation are frequently considered, under- (or "subset") approximation, denoted A^-, $A^- \subseteq A$, and "existential" approximation, denoted $A^!$, $A^! \cap A \neq \emptyset$. In what follows, a prime symbol will used to distinguish an approximation A' from the exact value A.

Notice that if A_1^+ and A_2^+ are over-approximations of A then also $A_1^+ \cap A_2^+$ is an over-approximation of A. Moreover, it is a better approximation than either A_1^+ or A_2^+. A similar property holds for under-approximations w.r.t. $\cup$. However, existential approximations do not enjoy this property.

Example 9.4.1 Consider the CLP program given in Example 9.2.1, and its specification in Example 9.2.4. We have that $\mathcal{I}_1$ is an under-approximation of the intended semantics $\mathcal{I}$, and $\mathcal{I}_2$ is an over-approximation of it. Therefore, $\mathcal{I}_1$ (resp. $\mathcal{I}_2$) is a specification of kind $\mathcal{I}^-$ (resp. $\mathcal{I}^+$), and used in proving properties w.r.t. $\mathcal{I}$. A different thing is that while trying to prove properties w.r.t. $\mathcal{I}_1$ (resp. $\mathcal{I}_2$) we may try to use also approximations of the form $\mathcal{I}_1^-$ (resp. $\mathcal{I}_2^+$) or $\mathcal{I}_1^+$ (resp. $\mathcal{I}_2^-$).

If we approximate $\mathcal{I}$, when dealing with partial correctness, approximations of the type $\mathcal{I}^+$ should be used, as $[\![P]\!] \not\subseteq \mathcal{I}^+ \Rightarrow [\![P]\!] \not\subseteq \mathcal{I}$, i.e., the program is definitely incorrect w.r.t. $\mathcal{I}$. When dealing with completeness, we should use approximations of either type $\mathcal{I}^-$ or $\mathcal{I}^!$ as both $\mathcal{I}^- \not\subseteq [\![P]\!]$ and $\mathcal{I}^! \cap [\![P]\!] = \emptyset$

imply that $\mathcal{I} \not\subseteq [\![P]\!]$ i.e., the program is definitely not complete w.r.t. $\mathcal{I}$. respectively. However, no interesting conclusion can be drawn if either $\mathcal{I}^-$ or $\mathcal{I}^!$ are used for correctness or $\mathcal{I}^+$ is used for completeness. We now discuss the use of approximations in program diagnosis.

9.4.1 Replacing the Oracle in Declarative Diagnosis

As seen in Section 9.3.3, in declarative diagnosis the existence of an oracle is assumed and the user is repeatedly asked questions about the intended semantics of the program. An idea is then to provide the system with (an approximation of) the intended semantics which can be used to automatically answer some of the oracle queries. When no sufficient conditions for a given query are satisfied, then the query cannot be answered automatically and the answer has to be provided by the user.

It is very seldom the case that there exists a formal specification $\mathcal{I}$ which completely describes the user's intention. Even less realistic is to expect that there exists such an executable specification. However, it is feasible to have formal/executable specifications which are approximations $\mathcal{I}^+$, $\mathcal{I}^-$ or $\mathcal{I}^!$ of the intended semantics. Such approximate specifications for declarative debugging of logic programs were introduced in [DNTM89], where four kinds of approximations were used. In our terminology those approximations were $\mathcal{I}^-$, $(\overline{\mathcal{I}})^!$, $\mathcal{I}^!$ and $\overline{\mathcal{I}^+}$ or, equivalently, $(\overline{\mathcal{I}})^-$ (where $\overline{S}$ denotes the complement of set S). That paper reported on experiments performed with a prototype implementation which was used to automate the answering of queries (except covering queries). User's answers are stored as an executable partial specification, which can then be used if the query is repeated. Actually, in many cases it is also possible to answer other queries. Table 9.4 presents a series of sufficient conditions which can be used by a declarative diagnoser to automatically answer some of the questions and avoid asking the user.

Example 9.4.2 Assume the query $\{sorted([a])\} \subseteq \mathcal{I}$ posed during incorrectness diagnosis. An approximation $\mathcal{I}^+$ containing all the atoms of the form $sorted(V)$ where V is an integer list (such as $\mathcal{I}_2$ of Example 9.2.4) is sufficient to obtain a negative answer.

Name	Property	Sufficient condition
Universal	$Q \subseteq \mathcal{I}$	$Q \subseteq \mathcal{I}^-$
	$Q \not\subseteq \mathcal{I}$	$Q \not\subseteq \mathcal{I}^+$, or $Q \cap \mathcal{I}^+ = \emptyset \ \wedge \ Q \neq \emptyset$
Existential	$Q \cap \mathcal{I} = \emptyset$	$Q \cap \mathcal{I}^+ = \emptyset$
	$Q \cap \mathcal{I} \neq \emptyset$	$Q \cap \mathcal{I}^- \neq \emptyset$, or $Q \subseteq \mathcal{I}^-$, or $\mathcal{I}^! \subseteq Q$

Table 9.4: Sufficient conditions for oracle queries

9.5 Approximating the Actual Semantics

The methods of program analysis allow computing approximations of the actual semantics $[\![P]\!]$, thus automate validation of programs w.r.t. a priori chosen properties.

The idea of using abstract interpretation for validation and diagnosis is not new. Its use for debugging of imperative programs has been studied by Bourdoncle [Bou93], and for debugging of logic programs by Comini et al. [CLMV96b]. Both approaches focus on some specific semantics and specific programming languages. It has also been used in abstract assertion checking proposed in Chapter 4. This section outlines the use of abstract interpretation for verification and diagnosis in a general setting of arbitrary fixpoint (set) semantics. For the time being, we assume that specifications are written as $\mathcal{I}_\alpha$ (i.e., the abstract domain is used as the language to write specifications). Thus, we discuss proving properties w.r.t. I_α, and only approximations of the actual model are considered.

9.5.1 Abstract Interpretation

In this section we recall the framework of abstract interpretation, already introduced in previous chapters and give a somewhat more detailed description of some additional properties which will be used in this chapter. An abstract semantic object is a finite representation of a, possibly infinite, set of actual semantic objects in the concrete domain (D). The set of all possible abstract semantic values represents an *abstract domain* (D_α) which is usually a complete lattice or cpo which is ascending chain finite. However, for this study, abstract interpretation is restricted to complete lattices over sets both for the concrete $\langle D, \subseteq \rangle$ and abstract $\langle D_\alpha, \sqsubseteq \rangle$ domains. As usual, the concrete and abstract domains are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : D \mapsto D_\alpha$, and *concretization* $\gamma : D_\alpha \mapsto D$, such that

$$\forall x \in D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y. \tag{9.1}$$

Note that in general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall \lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$), and is not equal to set inclusion. In an abuse of notation, however, we will usually write $\subseteq$ both for the concrete and abstract domain. Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of D in some precise sense. Again, in an abuse of notation, we will use $\cup$ and $\cap$, respectively (although they are in general not equal).

By monotonicity, the mappings α and γ (denoted f in what follows) satisfy:

$$x \subseteq y \ \Rightarrow \ f(x) \subseteq f(y). \tag{9.2}$$

We will also assume in some cases the following properties for α and γ:

$$f(x) \cap f(y) = \emptyset \Rightarrow \ x \cap y = \emptyset \quad \text{and} \quad f(x \cap y) = f(x) \cap f(y). \tag{9.3}$$

The abstract domain D_α is usually constructed with the objective of computing approximations of the semantics of a given program. Thus, all operations in the abstract domain also have to abstract their concrete counterparts. In particular, if the semantic operator S_P can be decomposed in lower level operations, and their abstract counterparts are locally correct w.r.t. them, then an abstract semantic operator S_P^α can be defined which is correct w.r.t. S_P. This means that $\gamma(S_P^\alpha(\alpha(x))$ is an approximation of $S_P(x)$ in D, and consequently, $\gamma(lfp(S_P^\alpha))$ is an approximation of $[\![P]\!]$. We will denote $lfp(S_P^\alpha)$ as $[\![P]\!]_\alpha$. The following relations hold:

$$\forall x \in D : \ \gamma(S_P^\alpha(\alpha(x)) \supseteq S_P(x) \tag{9.4}$$

$$\gamma([\![P]\!]_\alpha) \supseteq [\![P]\!] \ \text{ equivalently } \ [\![P]\!]_\alpha \supseteq \alpha([\![P]\!]). \tag{9.5}$$

An abstract operator S_P^α is said to be *precise*, if instead it satisfies that

$$\gamma([\![P]\!]_\alpha) = [\![P]\!] \ \text{ equivalently } \ [\![P]\!]_\alpha = \alpha([\![P]\!]). \tag{9.6}$$

Note that the construction presented allows obtaining over-approximations of $[\![P]\!]$. When (9.1) holds, the construction is termed a *Galois insertion.* If $\subseteq$ is used in (9.1) instead of $\supseteq$, we obtain a dual construction, termed a *reversed Galois insertion.* The dual relations of (9.4) and (9.5) also hold in this case.

In practice, the abstract domains should be sufficiently simple to allow effective computation of semantic approximations of programs. For example, Herbrand interpretations of some alphabet may be mapped into an abstract domain where each element represents a typing of predicates in some type system. For a given program P the abstract operator S_P^α would allow then to compute a typing of the predicates in the least Herbrand model of P.

Example 9.5.1 A simple example of abstract interpretation in logic programming can be constructed as follows. The concrete semantics (least Herbrand model) of a program P is $[\![P]\!] = lfp(T_P)$. So the concrete domain is $D = \wp(B_P)$ (where B_P is the Herbrand base of the program).

We consider over-approximating the set of "succeeding predicates", i.e those whose predicate symbols appear in $[\![P]\!]$. A possible abstraction is as follows. The abstract domain is $D_\alpha = \wp(B_P^\alpha)$, where B_P^α is the set of predicate symbols of P. Let $pred(A)$ denote the predicate symbol for an atom A. We define the abstraction function:

$$\alpha : D \to D_\alpha \;\text{ such that }\; \alpha(I) = \{pred(A) \mid A \in I\}.$$

The concretization function is defined as:

$$\gamma : D_\alpha \to D \;\text{ such that }\; \gamma(I_\alpha) = \{A \in B_P \mid pred(A) \in I_\alpha\}.$$

For example,

$$\alpha(\{p(a,b), p(c,d), q(a), r(a)\}) = \{p/2, q/1, r/1\}$$
$$\gamma(\{p/2, q/1\}) = \{p(a,a), p(a,b), p(a,c), \ldots, q(a), q(b), \ldots\}.$$

Note that $(D_\alpha, \gamma, D, \alpha)$ is a Galois insertion. The abstract semantic operator $T_P^\alpha : D_\alpha \to D_\alpha$ is defined as:

$$T_P^\alpha(I_\alpha) = \{pred(A) \mid \exists\, (A \leftarrow B_1, \ldots, B_n) \in P \;\forall\, i \in [1,n] : pred(B_i) \in I_\alpha\}.$$

Since D_α is finite and T_P^α is monotonic, the analysis (applying T_P^α repeatedly until fixpoint, starting from $\emptyset$) will terminate in a finite number of steps n and $[\![P]\!]_\alpha = T_P^\alpha {\uparrow} n$ approximates $[\![P]\!]$. For example, for the following program P,

```
p(X,Y) :- q(X), r(Y).
t(X) :- l(X).
m(X) :- s(X).
q(a). q(b).
r(a). r(c). r(X).
```

we have $B_P^\alpha = \{p/2, q/1, r/1, s/1, t/1, l/1, m/1\}$, and:

$$T_P^\alpha(\emptyset) = \{q/1, r/1\} \qquad T_P^\alpha(\{q/1, r/1\}) = \{q/1, r/1, p/2\}$$
$$T_P^\alpha(\{q/1, r/1, p/2\}) = \{q/1, r/1, p/2\}$$

So $T_P^\alpha\!\uparrow\!2 = T_P^\alpha\!\uparrow\!3 = \{q/1, r/1, p/2\} = [\![P]\!]_\alpha$

9.5.2 Abstract Diagnosis

The technique of abstract diagnosis [CLMV96b, CLMV96a] is based on the use of *observables* which correspond roughly to the abstraction functions α used in abstract interpretation with some additional properties. Observables (in a similar way to semantics) allow extracting the properties of interest from the execution of a goal, while hiding details which are not relevant. The intended semantics with respect to the observable α is denoted $\mathcal{I}_\alpha$ and is assumed to be an exact description.

Abstract diagnosis searches for incorrectness and incompleteness errors as defined in Section 9.3.2, using the sufficient conditions given in Table 9.2. The semantic operator S_P is replaced by S_P^α, in a similar way to abstract interpretation. However, and unlike abstract interpretation, no fixpoint computation is needed and $lfp(S_P^\alpha)$ is not computed.

Two different kind of observables are considered in [CLMV96a]. *Complete* observables provide stronger results but are often not practical because the specification of the intended semantics $\mathcal{I}_\alpha$ is infinite and diagnosis would not terminate. Such complete observables correspond to the precise abstract operators of Section 9.5.1. The second kind of observables considered in [CLMV96a] are called *approximate* observables and their corresponding operator S_P^α is correct but not precise (as is usually the case in abstract interpretation).

9.5.3 Validation using Abstract Interpretation

Abstract diagnosis localizes suspected program constructs following the diagnosis by proof principle. The proof attempt may succeed in which case the program satisfies the requirement $\mathcal{I}$ (expressed as $\mathcal{I}_\alpha$), and abstract diagnosis works as validation.

An alternative way of validation is to compute abstract approximations $[\![P]\!]_\alpha$ of the actual semantics of the program $[\![P]\!]$ and then use the definitions given in Table 9.1 instead of the sufficient conditions of Table 9.2 (on which abstract diagnosis is based). This is reasonable if one considers that usually program analyses are performed in any case to use the information inferred for optimizing the code of the program.

For now, we assume that the program specification is given as a semantic value $\mathcal{I}_\alpha \in D_\alpha$. Comparison between actual and intended semantics of the program should be done in the same domain. Thus, for comparison we need in principle $\alpha([\![P]\!])$. However, using abstract interpretation, we can compute instead $[\![P]\!]_\alpha$, which is an approximation of $\alpha([\![P]\!])$, and can be compared with $\mathcal{I}_\alpha$. We will use the notation $[\![P]\!]_{\alpha^+}$ to represent that $[\![P]\!]_\alpha \supseteq \alpha([\![P]\!])$. $[\![P]\!]_{\alpha^-}$ indicates that $[\![P]\!]_\alpha \subseteq \alpha([\![P]\!])$. Table 9.5 gives sufficient conditions for correctness and completeness w.r.t. $\mathcal{I}_\alpha$ which can be used when $[\![P]\!]$ is approximated.

Property	**Definition**	**Sufficient condition**
P is partially correct w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^+} \subseteq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \subseteq [\![P]\!]_{\alpha^-}$
P is incorrect w.r.t. $\mathcal{I}_\alpha$	$\alpha([\![P]\!]) \not\subseteq \mathcal{I}_\alpha$	$[\![P]\!]_{\alpha^-} \not\subseteq \mathcal{I}_\alpha$, or $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset \wedge [\![P]\!]_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathcal{I}_\alpha$	$\mathcal{I}_\alpha \not\subseteq \alpha([\![P]\!])$	$\mathcal{I}_\alpha \not\subseteq [\![P]\!]_{\alpha^+}$

Table 9.5: Validation problems using approximations

The following conclusions can be drawn from Table 9.5. Analyses which use a Galois insertion (α^+, γ^+), and thus over-approximate the actual semantics (i.e., those denoted as $[\![P]\!]_{\alpha^+}$) are specially suited for proving partial correctness and incompleteness with respect to the abstract specification $\mathcal{I}_\alpha$. It will also be sometimes possible to prove incorrectness in the extreme case in which the semantics inferred by the program is incompatible with the abstract specification, i.e., when $[\![P]\!]_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset$. Note that it will only be possible to prove completeness if the abstraction is precise. According to Table 9.5 only $[\![P]\!]_{\alpha^-}$ can be used with this end, and in the case we are discussing $[\![P]\!]_{\alpha^+}$ holds. Thus, the only possibility is that the abstraction is precise.

On the other hand, if a reversed Galois insertion is used (α^-, γ^-), and then analysis under-approximates the actual semantics (the case denoted $[\![P]\!]_{\alpha^-}$), it will be possible to prove completeness and incorrectness. Partial correctness and incompleteness can only be proved if the analysis is precise.

Note that the results obtained for direct Galois insertions (α^+, γ^+) are in essence equivalent to the ones presented for abstract diagnosis [CLMV96a] for *approximate* observables. In the case of precise abstractions, also completeness may be derivable, and this corresponds to the complete observables of [CLMV96a].

Example 9.5.2 If the abstract interpretation tells that in $[\![P]\!]_{\alpha^+}$ the type of a predicate p with just one argument position is *intlist* and the user has declared it in $\mathcal{I}_\alpha$ as *list*, then under some natural assumptions about ordering in the abstract domain we conclude that $[\![P]\!]_{\alpha^+} \subseteq \mathcal{I}_\alpha$, i.e., the program is correct w.r.t. the declared $\mathcal{I}_\alpha$ (or more precisely w.r.t. $\gamma(\mathcal{I}_\alpha)$). However, the program may still be incorrect w.r.t. the precise intention $\mathcal{I}$, which is not given by the declaration.

Example 9.5.3 Assume now that $[\![P]\!]_{\alpha^+} \not\subseteq \mathcal{I}_\alpha$. We cannot conclude that P is correct w.r.t. $\mathcal{I}_\alpha$. We cannot conclude the contrary either. For example if the abstract interpretation tells that the type of the predicate p with one argument position is *list* while the user declares it as *intlist* then P may still be correct w.r.t. the declaration. This can be due to the loss of accuracy introduced by the abstraction. In any case it may be desirable to localize a fragment of the program responsible for this discrepancy. A more careful inspection would then be needed to check whether the fragment is erroneous w.r.t. the declaration, or not.

If analysis information allow us to conclude that the program is incorrect or incomplete w.r.t. $\mathcal{I}_\alpha$, an (abstract) symptom has been found which ensures that the program does not satisfy the requirement. Thus, a diagnosis should be performed to locate the program construct responsible for the symptom. We are studying the possibility of using for that purpose the conditions in Table 9.2, in a similar way as done in abstract diagnosis [CLMV96b].

9.6 Towards an Integrated Validation and Diagnosis Environment

In the previous sections we have addressed the problem of validation and diagnosis of a program with respect to incomplete requirements. We have hopefully contributed to clarifying how known verification and debugging techniques can be combined to support the process of program development, specially in the case in which approximations are used. This final section discusses the design of an environment integrating validation and diagnosis tools making an extensive use of semantic approximations.

9.6.1 Some Practical Aspects of the Debugging Process

An important aspect of debugging is that in practice the process of program construction is often iterative, and the iterations update incrementally not only the program but also the requirements. This is related to the observation that user's expectations concerning a program are rarely fully described. At each stage of development we have a (possibly empty collection of) subset approximation(s) $\mathcal{I}^-$ of the intended semantics and a (possibly empty collection of) superset approximations[3] $\mathcal{I}^+$ which together represent the specification. The program in hand should be complete w.r.t. $\mathcal{I}^-$ and partially correct w.r.t. $\mathcal{I}^+$. In the previous sections we mentioned some well-known proof methods used for checking that.

If the proof fails, the failure points to some fragments of the program, which may possibly be erroneous. The failure may be due to:

1. an error in the program causing violation of the specification in hand,

2. the specification is too weak, or

3. incompleteness of the prover.

Note that if an error exists then it can only be due to the fragments identified. The user should inspect them in order to identify the reason of failure. If the user identifies that the reason is an error then the program has to be corrected.

[3]The other kinds of approximations may also be present but, for simplicity, we will consider these two in this discussion.

If, on the other hand, the user does not identify an error then alternatively it may be possible to strengthen the specification in such a way that a proof can be achieved.[4]

If the proof succeeds we may:

1. stop the development process, or

2. update the specification.

In particular, the latter is needed if the behaviour of the program w.r.t. the first specification is not acceptable and the user wants to clarify why.

Note that even if the proof succeeds, as specifications may be partial, some bugs may still be hidden in the program. For example, if the techniques presented in Section 9.5 are used, some bugs may not be captured by the abstract semantics. Thus, if during testing or execution of the program some unexpected behaviour is found, diagnosis should start for it. The well-known technique of declarative diagnosis is then applicable, which, as we have seen, can also rely on approximations of the intended semantics.

9.6.2 Which tools are needed

We believe that an integrated environment incorporating the techniques described so far (as well as other techniques, such as procedural debugging and visualization, which are beyond the scope of the work presented in this chapter) can be of great help in speeding up the code development process. In this section we propose some tools to be included in the environment. Figure 8.1 presented a possible arquitecture of tools for such an environment. The intention is to detect bugs as early as possible, i.e., during compilation or even editing. This can only be achieved by (semi-) automatic analysis of the (not necessarily completely developed) program in the presence of some (approximate) specifications. An example of such techniques is type checking, which proved to be useful for that purpose. Our approach puts a framework for working with properties that may be more general than classical type systems.

The common integrating concept for the tools proposed is the notion of semantic approximation which is involved in

[4] An example of weak specification is given in Example 9.3.1.

- describing user's intentions,
- program analysis,
- comparing the results of program analysis with the user's intentions,
- verification,
- debugging.

Semantic approximations will be expressed by means of assertions, for example using the assertion language proposed in Chapter 8. The fundamental technique mentioned in this context is that of abstract interpretation which allows automatic synthesis of semantic approximations, for abstract verification and for abstract debugging.

To support the above mentioned activities we may need the following tools:

- A program analyzer: it takes the program and the selected abstract domain(s) and generates an approximation of the actual semantics of the program. In the case of CLP programs standard analysis techniques can be used for this purpose.
- An assertion translator: if the language for assertions is the underlying programming language or an abstract domain different from that used internally by the tool, this translator is in charge of transforming the intended semantics into the abstract domain to be used by the analyzer. An intelligent translation scheme would be able to select the best among a set of abstract domains depending on the requirements expressed by the user in the intended model.
- A comparator: it would compare the user requirements and the information generated by the analysis. It can produce three different results:
  - The requirement is verified.
  - The requirement does not hold. An *abstract symptom* has been found and diagnosis should start.

  - None of the above. We cannot prove that the requirement holds nor that it does not hold. Run-time tests could be introduced which would make sure that the requirements hold. Clearly, this introduces an important overhead and could be turned on only during program testing.

- A diagnoser based on abstraction: the diagnoser tries to localize the program construct responsible for the abstract symptom. It would use algorithms based on the sufficient conditions of Table 9.2. Thus it will locate possible error sources.

- A declarative (concrete) diagnoser: it would be used once all abstract symptoms have been diagnosed and eliminated from the program in order to underpin all subsequent bugs in the program which appear during program testing and execution. As in Section 9.4.1, the program would store approximations of the intended semantics to avoid asking the user whenever the question can be solved using such approximations.

Partial prototypes of the component tools are mentioned above are currently being developed. For example, the assertion language of Chapter 8 together with an analyzer, and a comparator has been incorporated in the CIAO system which works on the domains of moded types, definiteness, freeness, and grounding dependencies for CLP programs.

9.7 Chapter Conclusions

We have seen how approximations can be used in program diagnosis and validation. First, different techniques have been recalled and the properties which they aim at proving have been formulated in a set-theoretic setting. Then we have seen some cases in which it is still possible to prove properties when approximations of the exact values are used. Then we have presented examples of existing and future tools which use approximations of program semantics and can perform verification and/or diagnosis tasks.

Chapter 10

Conclusions and Future Work

10.1 Conclusions

This thesis has presented several novel techniques for the analysis, optimization, and debugging of (constraint) logic programs. believe that many of the results presented are also applicable to other other high level programming languages, and, in general, to declarative languages. Most of the proposed techniques have been implemented in the CIAO and &-Prolog systems and evaluated experimentally. Some of the fundamental conclusions which may be derived this work are:

- It is possible to perform static program analysis in an incremental way without loss of accuracy and with important efficiency improvements. A generic analysis algorithm has been formalized which is parametric with respect to the analysis strategy used. The possible modifications to a program have been classified and in each case one or several algorithms for performing the related incremental analysis have been given. Such algorithms are expressed in terms of the generic algorithm mentioned above.

- Taking the generic analysis algorithm as a starting point, we have identified a class of analysis strategies which are especially efficient. We have shown experimentally that these strategies are very appropriate for the incremental case and that even for the non-incremental case they are as good or even better than the strategies used in non-incremental analyzers.

- It is possible to analyze the full ISO Prolog language. We have developed a series of analysis techniques which together with other existing ones can be used to analyze programs with all the impure features of real-life languages such as Prolog.

- Multiple specialization is not hard to implement and allows obtaining relevant performance improvements. We have developed a framework for multiple specialization based on abstract interpretation which is as powerful as the best of the frameworks proposed previously, but with the advantage of requiring only small modifications to existing abstract interpreters. The optimizations performed by this specializer are based on the notion of abstract executability, which has been formalized in this work. Abstract multiple specialization has been implemented and integrated in a compiler for the first time and its effect quantified. The experimental results show that it is an effective technique.

- There is a strong relationship between partial evaluation and abstract interpretation, and their integration may improve the results of both. We have clarified for the first time the relation between such techniques. We have proposed an algorithm to obtain specialized programs directly from analysis information and which allows optimizations which are not achievable by traditional partial evaluation. We have also identified the modifications which are required in the abstract interpretation framework in order to obtain all the specializations which partial evaluation is capable of performing.

- The cost of dynamic scheduling can be reduced using specialization techniques which use information obtained by static analysis. We have performed a study which has led us to identify a series of techniques for program transformation which allow reducing the cost of dynamic scheduling without modifying the operational semantics of the program. This is important in order to avoid obtaining optimized programs which are less efficient than the original one.

- It is possible to extend (constraint) logic languages with a uniform set of assertions which are useful in a large number of debugging and compilation tools. We have designed an assertion language which allows communicating properties and requirements of the program between the user and the different tools which may exist in an advanced program development environment. We have provided a scheme for the run-time checking of assertions and we have also studied the possibility of compile-time checking of assertions by the use of static analysis (abstract interpretation).

- It is possible to perform mixed debugging between compile- and run-time in (constraint) logic languages by means of analysis based on approximations. We have identified in a systematic way a set of sufficient conditions which allow obtaining conclusions about some relevant questions for program validation and debugging, such as the correctness and completeness of a program, when both the requirements (for example due to incomplete specifications) and the information available about the program (for example, that obtained by static program analysis) are approximated.

10.2 Future Work

Even though most of the compilation techniques presented in this work have been implemented in the CIAO and &-Prolog systems, and experiments have been conducted which show their relevance in practice, it would be interesting to evaluate their applicability in other contexts and their effectiveness for other sets of programs. Simultaneously, some of the techniques presented would benefit from the study of other related techniques.

In spite of having pointed out possible avenues for future work in each chapter, we summarize here the most important ones. Concretely,

- It remains as future work to study and implement a compiler in which not only analysis, but other (all) compilation phases, such as, for example, optimization, are performed incrementally.

- Even though the analysis algorithms presented in Chapter 3 are very efficient, it could be interesting to experiment with different analysis strategies within the efficient class of those which preserve the strongly connected components of the analysis graph.

- Regarding the analysis of full languages, some of the different existing alternatives could be compared experimentally. For example, the importance of providing additional information when analysis is not capable of generating accurate information could be explored.

- The multiple specialization framework presented in Chapter 5 is capable of minimizing the number of versions in the specialized program and thus it avoids versions which do not allow further optimization. However, the final program will have as many versions as necessary in order to obtain all possible optimizations. It would be interesting to experiment with other strategies which allow collapsing versions when the benefit achieved by keeping them separate does not pay off with respect to the increase in size of the specialized program.

- For the integration of partial evaluation in the specialization framework presented in Chapter 5 it would be interesting to experiment with different abstract domains and local control strategies. Also, even though the proposed integration allows performing optimizations which are not achievable by traditional partial evaluation, it would be interesting to perform an experimental comparison of the efficiency of the proposed specializer with that of the existing partial evaluators for the cases in which partial evaluation suffices.

- The optimization of dynamic scheduling has produced very promising experimental results. It remains as future work to search for transformation strategies with somewhat weaker preconditions. That way, they would be applicable in some cases in which the proposed techniques are not, even though it would be possible to reduce the cost of dynamic scheduling for them.

- The proposed assertion language has been implemented in CIAO. However, it would be interesting to experiment with it in other environments and with other tools for the development of (constraint) logic programs. This could be useful in order to determine whether the assertion language is expressive enough and whether its complexity is acceptable.

- Finally, the theoretical study in Chapter 9 is of relevance in order to determine under which circumstances it is possible to obtain conclusions about questions such as program correctness. It would be interesting to evaluate experimentally how often it is possible to prove each of the sufficient conditions proposed in the real-life program development and debugging process.

Bibliography

[ABB$^+$97] A. Aggoun, F. Benhamou, F. Bueno, M. Carro, P. Deransart, W. Drabent, G. Ferrand, F. Goualard, M. Hermenegildo, C. Lai, J.Lloyd, J. Maluszynski, G. Puebla, and A. Tessier. CP Debugging Tools: Clarification of Functionalities and Selection of the Tools. Technical Report D.WP1.1.M1.1-2, DISCIPL Project, June 1997.

[AM94] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6):743–765, 1994.

[AMSS94] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In Springer-Verlag, editor, *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 266–280, Namur, Belgium, September 1994.

[AP93] K. R. Apt and D. Pedreschi. Reasoning about termination of pure PROLOG programs. *Information and Computation*, 1(106):109–157, 1993.

[AU77] A. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Mass., 1977.

[BCHP95] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Data–flow Analysis of Standard Prolog Programs. In *ICLP95 WS on Abstract Interpretation of Logic Languages*, Japan, June 1995.

[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.

[BDD+97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linkoping, Sweden, May 1997. U. of Linkoping Press.

[BDM97] J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linkoping, Sweden, May 1997. U. of Linkoping Press.

[BGCH93] F. Bueno, M. García de la Banda, D. Cabeza, and M. Hermenegildo. The &–Prolog Compiler System — Automatic Parallelization Tools for LP. Technical Report CLIP5/93.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, June 1993.

[BGH94a] F. Bueno, M. García de la Banda, and M. Hermenegildo. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *Parallel Symbolic Computation*, pages 63–73. World Scientific Publishing Company, September 1994.

[BGH94b] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[BJ88] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *Fifth International Conference and Symposium on Logic Programming*, pages 669–683, Seattle, Washington, August 1988. MIT Press.

[BJ92] M. Bruynooghe and G. Janssens. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2&3):205–258, 1992.

[BLM94] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19–20:443–502, July 1994.

[Bou93] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55, 1993.

[Boy93] J. Boye. Avoiding dynamic delays in functional logic programs. In *Programming Language Implementation and Logic Programming*, number 714 in LNCS, pages 12–27, Estonia, August 1993. Springer-Verlag.

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

[Bue95] F. Bueno. The CIAO Multiparadigm Compiler: A User's Manual. Technical Report CLIP8/95.0, Facultad de Informática, UPM, June 1995.

[Bul84] M.A. Bulyonkov. Polivariant Mixed Computation for Analyzer Programs. *Acta Informatica*, 21:473–484, 1984.

[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.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.

[CC94] J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

[CD93] C. Consel and O. Danvy. Tutorial Notes on Partial Evaluation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, pages 493–501, Charleston, South Carolina, 1993. ACM.

[CDG93] M. Codish, S. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, pages 451–464, Charleston, South Carolina, 1993. ACM.

[CDMV93] B. Le Charlier, O. Degimbe, L. Michael, and P. Van Hentenryck. Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In *Workshop on Static Analysis*, pages 15–26. Springer-Verlag, September 1993.

[CDY91] M. Codish, D. Dams, and E. Yardeni. Derivation and Safety of an Abstract Unification Algorithm for Groundness and Aliasing Analysis. In *Eighth International Conference on Logic Programming*, pages 79–96, Paris, France, June 1991. MIT Press.

[CF92] A. Cortesi and G. File. Abstract interpretation of prolog: the treatment of the built-ins. In *Proc. of the 1992 GULP Conference on Logic Programming*, pages 87–104. Italian Association for Logic Programming, June 1992.

[CGBH94] M. Codish, M. García de la Banda, M. Bruynooghe, and M. Hermenegildo. Goal Dependent vs Goal Independent Analysis of Logic Programs. In F. Pfenning, editor, *Fifth International Conference on Logic Programming and Automated Reasoning*, number 822 in LNAI, pages 305–320, Kiev, Ukraine, July 1994. Springer-Verlag.

[CK93] C. Consel and S.C. Koo. Parameterized partial deduction. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, July 1993.

[CLMV96a] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. Submitted for publication, 1996.

[CLMV96b] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996.

[Col87] A. Colmerauer. Opening the Prolog-III Universe. In *BYTE Magazine*, August 1987.

[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.

[Cou97] P. Cousot. Types as Abstract Interpretations. In *Symposium on Principles of Programming Languages*, pages 316–331. ACM Press, January 1997.

[CRV94] B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework Which Accurately Handles Prolog Search–Rule and the Cut. In *International Symposium on Logic Programming*, pages 157–171. MIT Press, November 1994.

[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.

[D.84] Knuth D. Literate programming. *Computer Journal*, 27:97–111, 1984.

[Deb89a] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

[Deb89b] S.K. Debray. Flow analysis of dynamic logic programs. *Journal of Logic Programming*, 7(2):149–176, September 1989.

[Deb92] S. Debray, editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, volume 13(1–2). North-Holland, July 1992.

[Deb93] S. K. Debray. QD-Janus : A Sequential Implementation of Janus in Prolog. *Software—Practice and Experience*, 23(12):1337–1360, December 1993.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard (Reference Manual).* Springer, 1996.

[Der93] P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118:99–166, 1993.

[DGB96] S. Debray, D. Gudemann, and P. Bigot. Detection and optimization of suspension-free logic programs. *Journal of Logic Programming*, 29(1–3):171–195, October–December 1996.

[DGT96] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation.* Number 1110 in LNCS. Springer, February 1996. Dagstuhl Seminar.

[DM93] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* The MIT Press, 1993.

[dMSC93] Vítor Manuel de Morais Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, University of Bristol, August 1993.

[DNTM88] W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.

[DNTM89] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In (H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.

[FCH92] M. Fernández, M. Carro, and M. Hermenegildo. IDeal Resource Allocation (IDRA): A Technique for Computing Accurate Ideal

Speedups in Parallel Logic Languages. Technical report, T.U. of Madrid (UPM), June 1992.

[Fer87] G. Ferrand. Error diagnosis in logic programming. *J. Logic Programming*, 4:177–198, 1987.

[Gal92] J.P. Gallagher. Static Analysis for Logic Program Specialization. In *Workshop on Static Analysis WSA'92*, pages 285–294, 1992.

[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.

[GB90] J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialization. In *1990 International Conference on Logic Programming*, pages 732–746. MIT Press, June 1990.

[GB91] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(1991):305–333, 1991.

[GCS88] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6(2–3):159–186, 1988.

[GDL92] Roberto Giacobazzi, Saumya Debray, and Giorgio Levi. A generalized semantics for constraint logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 581–591, ICOT, Japan, 1992. Association for Computing Machinery.

[GdW94] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[GGL94] M. Gabbrielli, R. Giacobazzi, and G. Levi. Goal independency and call patterns in the analysis of logic programs. In *ACM Symposium on Applied Computing*. ACM Press, 1994.

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

[GH93] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 437–455. MIT Press, Cambridge, MA, October 1993.

[GHB+96] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, 1996.

[GL96] J. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110, pages 115 – 136. Springer Verlag Lecture Notes in Computer Science, 1996.

[GMS95] 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.

[Gr94] R. Glueck and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *International Symposium on Programming Language Implementation and Logic Programming, PLILP'94*, volume 844 of *LNCS*, pages 165–182, Madrid, Spain, 1994. Springer Verlag.

[Gro97] The CLIP Group. Program Assertions. Technical Report CLIP4/97.1, Facultad de Informática, UPM, August 1997.

[Han93] M. Hanus. Analysis of Nonlinear Constraints in CLP(R). In *Tenth International Conference on Logic Programming*, pages 83–99. MIT Press, June 1993.

[HBC+96] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: A Demo and Status Report. In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation Technology*. Computer Science Department, Technical University of Madrid, September 1996.

[HBGP95] M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995.

[HCC94] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179 – 210, 1994.

[HG91] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HG97] M. Hermenegildo and The CLIP Group. `pl2texi`: An Automatic Documentation Generator for (C)LP – Reference Manual. Technical Report CLIP5/97.1, Facultad de Informática, UPM, August 1997.

[HL94] P. Hill and J. Lloyd. *The Goedel Programming Language.* MIT Press, Cambridge MA, 1994.

[HP96] M. Hermenegildo and G. Puebla. Applying Multiple Abstract Specialization to Program Parallelization (abstract). In O. Danvy, R. Glueck, and P. Thiemann, editors, *Partial Evaluation– Dagstuhl Seminar Report*, number 134. IBFI – Sloss Dagstuhl, February 1996.

[HPMS95] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

[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.

[HWD92] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[JB92] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

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

[JL88] D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.

[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.

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

[JM86] N. D. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal function graphs. In *Thirteenth Ann. ACM*

Symp. Principles of Programming Languages, pages 296–306. St. Petersburg, Florida, ACM, 1986.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[Jor94] N. Jorgensen. Finding Fixpoints in Finite Function Spaces Using Neededness Analysis and Chaotic Iteration. In *International Static Analysis Symposium*, 1994.

[JS87] N. Jones and H. Sondergaard. A semantics-based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis-Horwood, 1987.

[KB95] A. Krall and T. Berger. Incremental global compilation of prolog with the vienna abstract machine. In *International Conference on Logic Programming*. MIT Press, June 1995.

[KMM+95] A. Kelly, A. Macdonald, K. Marriott, H. Sondergaard, P. Stuckey, and R. Yap. An optimizing compiler for CLP(R). In *Proceedings of the Conference on Constraint Programming CP'95*. LNCS, Springer-Verlag, 1995.

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

[KMSSar] A. Kelly, K. Marriott, H. Søndergaard, and P.J. Stuckey. A practical object-oriented analysis engine for CLP. *Software: Practice and Experience*, to appear.

[Kom92] J. Komorovski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta Programming in Logic, Proceedings of META'92*, volume 649 of *LNCS*, pages 49–69. Springer-Verlag, 1992.

[Kow74] R. A. Kowalski. Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pages 569–574, 1974.

[Kow80] R. A. Kowalski. Logic as a computer language. *Proc. Infotec State of the Art Conference, Software Development: Management*, June 1980.

[LD97] M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. Technical Report CW 250, Departement Computerwetenschappen, K.U. Leuven, Belgium, June 1997. Accepted for Publication in New Generation Computing.

[LDMH93] B. Le Charlier, O. Degimbe, L. Michel, and P. Van Henteryck. Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In *International Workshop on Static Analysis*. Springer-Verlag, 1993.

[Leu95] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, 1995.

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

[LM95] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. Technical Report CW 220, Departement Computerwetenschappen, K.U. Leuven, Belgium, December 1995.

[LM96] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996.

[LNS82] J. L. Lassez, V. L. Nguyen, and E. A. Sonnenberg. Fixed point theorems and semantics: a folk tale. *Information Processing Letters*, 14(3):112–116, 1982.

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

[LS96] Michael Leuschel and De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996.

[LSdW96] M. Leuschel, D. De Schreye, and D. A. de Waal. A conceptual embedding of folding into partial deduction: towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint Int,. Conf. and Symp. on Logic Programming (JICSLP'96)*. MIT Press, 1996.

[Mel86] C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 463–475. Springer-Verlag, July 1986.

[MG95] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing fl exible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Japan, June 1995. MIT Press.

[MGH94] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[MH89a] K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.

[MH89b] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[MH90a] 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.

[MH90b] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 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. Originally published as Technical Report FIM 59.1/IA/90, Computer Science Dept, Universidad Politecnica de Madrid, Spain, August 1990.

[MJMB89] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, June 1989.

[MMRS55] John McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon. A proposal for the dartmouth summer research project on artificial intelligence. Report, manuscript, MITAI, MITad, August 1955.

[MS89] K. Marriott and H. Sondergaard. Abstract interpretation, 1989. 1989 SLP Tutorial Notes.

[MS92] K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *19th. Annual ACM Conf. on Principles of Programming Languages*. ACM, 1992.

[MSJ94] K. Marriott, H. Søndergaard, and N.D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[MSJB95] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the Practicality of Abstract Equation Systems. In *International Conference on Logic Programming*. MIT Press, June 1995.

[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.

[Nai97] L. Naish. A three-valued declarative debugging scheme. In *8th Workshop on Logic Programming Environments*, July 1997. ICLP Post-Conference Workshop.

[Neu90] G. Neumann. Transforming interpreters into compilers by goal classification. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-programming in Logic*, pages 205–217, Leuven, Belgium, 1990. K. U. Leuven.

[oBLdM92] University of Bristol, Katholieke Universiteit Leuven, and Universidad Politécnica de Madrid. Interface between the prince prolog analysers and the compiler. Technical Report KUL/PRINCE/92.1, Katholieke Universiteit Leuven, October 1992.

[PBH97] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997.

[PGH+96] G. Puebla, M. García de la Banda, M. Hermenegildo, K. Marriott, and P. Stuckey. Automatic Optimization of Logic Programs with Dynamic Scheduling. In *Workshop on Abstract Interpretation of Logic Languages*, Jerusalim, December 1996. The Hebrew University.

[PGH97] G. Puebla, J. Gallagher, and M. Hermenegildo. Towards Integrating Partial Evaluation in a Specialization Framework based on Generic Abstract Interpretation. In *Proceedings of the ILPS'97 Workshop on Specialization of Declarative Programs*, October 1997. Post ILPS'97 Workshop.

[PGMS97] G. Puebla, M. García de la Banda, K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on Logic Programming*, pages 93–107, Leuven, Belgium, June 1997. MIT Press, Cambridge, MA.

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

[PH96a] G. Puebla and M. Hermenegildo. Abstract Specialization and its Application to Program Parallelization. In J. Gallagher, editor, *V International Workshop on Metaprogramming and Metareasoning in Logic*, 1996.

[PH96b] G. Puebla and M. Hermenegildo. Automatic Optimization of Dynamic Scheduling in Logic Programs. In *Programming Languages: Implementation, Logics, and Programs*, number 1140 in LNCS, Aachen, Germany, September 1996. Springer-Verlag. Poster.

[PH96c] 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.

[PH96d] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *II Workshop on Verification and Analysis of Logic Languages*, September 1996.

[PH97a] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *Journal of Logic Programming*, 1997. Submitted for publication.

[PH97b] G. Puebla and M. Hermenegildo. Abstract Specialization and its Application to Program Parallelization. In J. Gallagher, editor, *VI International Workshop on Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.

[PRO94] International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England. *PROLOG. ISO/IEC DIS 13211 — Part 1: General Core*, 1994.

[PRO95] International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England. *PROLOG. Working Draft 7.0 X3J17/95/1 — Part 2: Modules*, 1995.

[RN95] Stuart Russel and Peter Norvig. *Artificial Intelligence - A Modern Approach.* Prentice-Hall, Englewood Cliffs, 1995.

[RR93] G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, Charleston, South Carolina, 1993. ACM.

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

[SCWY91] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[Sha82] E. Shapiro. *Algorithmic Program Debugging.* ACM Distiguished Dissertation. MIT Press, 1982.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.

[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.

[Tar72] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1:140–160, 1972.

[Tay90] A. Taylor. LIPS on a MIPS: Results from a prolog compiler for a RISC. In *1990 International Conference on Logic Programming*, pages 174–189. MIT Press, June 1990.

[Tur88] V. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

[VD90] P. Van Roy and A. M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *North American Conference on Logic Programming*, pages 501–515. MIT Press, October 1990.

[VD92] P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

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

[Vet94] E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Constraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.

[VWL94] B. Vergauwen, J. Wauman, and J. Levi. Efficient Fixpoint Computation. In *International Static Analysis Symposium*. Springer-Verlag, 1994.

[WHD88] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International*

Conference and Symposium on Logic Programming, pages 684–699. MIT Press, August 1988.

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