Data Flow And Control Flow Analysis Of Logic Programs

Renganathan Sundararajan

CIS-TR-94-08
March 1994

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Abstract

This dissertation studies two related problems in data flow and control flow analysis of logic programs and provides efficient solutions.

The first part of the dissertation proposes a static analysis based on abstract interpretation to compute the data dependencies in a program; specifically, it derives precise information about sharing, freeness, and groundness of variables. This information is used in parallel execution and optimization of logic programs. The analysis has polynomial time complexity in the number of clauses and clause literals, and exponential worst-case complexity in the number of clause variables. In practice, the number of iterations required for computing the fixed-point appears to be independent of the height of the abstract domain and is usually less than five. A widening operation is used to accelerate convergence when computing the minimal function graph semantics of programs.

The second part of the dissertation provides a practical solution to the problem of simultaneous data flow and control flow analysis of logic programs. All analyses of logic programs to date have assumed that either control flow or data flow is known. We develop a framework for simultaneous data flow and control flow analysis. It is shown that a sub-problem in deriving control flow, namely, finding the set of all minimal permissible modes, is computationally intractable. We define a practical approximation algorithm and study its usefulness and complexity. The approximation algorithm derives minimal permissible modes for many non-trivial programs in polynomial time. A control flow for each clause and each entry substitution is then derived using the proposed framework and the approximation algorithm.

The simultaneous derivation of data flow and control flow in logic programs has many advantages. It results in a flexible control flow, uncovers more parallelism at compile-time than previous proposals, and enables other program analysis such as compile-time memory reuse strategies. Above all, it moves logic programming a step closer towards the ideal of separating the logic and control aspects of a program.

"Data Flow and Control Flow Analysis of Logic Programs," a dissertation prepared by Renganathan Sundararajan in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

Chair of the Examining Committee

Date

Commitee in charge: Dr. John Conery, Chair
Dr. Evan Tick
Dr. William Clinger
Dr. Richard Koch

Accepted by:

Vice Provost and Dean of the Graduate School

Acknowledgements

I wish to thank my advisor John Conery for his support throughout this research. He encouraged my interest in abstract interpretation and data flow analysis. I am thankful to Evan Tick for many hours of discussion, and for suggesting improvements to the thesis. Will Clinger helped me learn denotational semantics and corrected many of my misunderstandings and I am thankful to him. I thank Professor Richard Koch for serving on my committee and for showing interest in my work.

I am grateful for the financial support from Hewlett-Packard, Motorola, Tektronix, OACIS, and the National Science Foundation during various stages of my research.

Michael Codish, Saumya Debray, Manuel Hermenegildo, Harald Søndergaard, and Will Winsborough answered many of my questions and I thank them for their help.

Apple Oxman and family provided a home away from home and made my stay in Eugene enjoyable. My wife Renuka has been a constant source of strength. I am grateful to my parents for their love and affection.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Decades of research in automatic theorem proving and the desire to use logic in natural language processing led to the development of the symbolic computer programming language Prolog, originally meant to be an acronym for *Programming in Logic*. Prolog is based on a subset of first order predicate calculus, known as Horn Clause Logic. Horn clauses may be thought of as implications with 0 or 1 consequent. (Exact definitions will be given in Chapter 2.) Prolog has been found to be suitable for a wide spectrum of applications including hardware verification systems, automated theorem provers for various logics, VLSI chip design and layout tools, and expert systems for polymer design, among others [60].

An important advantage of declarative computer programming languages such as those based on Horn Clause Logic is that the mathematical semantics of declarative languages imposes no a *priori* ordering on or sequencing of the evaluation of the sub-expressions of a program. This allows us to execute in parallel the sub-computations in a logic program subject to the constraint of maintaining a consistent value for a variable within the same scope. Many models of computation have been proposed for exploiting the inherent parallelism in logic programs. These generally fall under the categories of *Or* parallel models and *And* parallel models or a combination of them [20]. Roughly speaking, in *Or* parallel models, multiple solutions to a query are searched in parallel. In *And* parallel models, the computation of a single solution to a query is performed in parallel.

In the Independent-And models, two body goals may be solved in parallel if they do not bind variables in arguments to conflicting terms. This requires an analysis of the data dependencies which can be done either at run-time or statically at compile-time.

The following is a fragment of the well-known quicksort algorithm. Given a list of numbers as the first argument, the program quicksort splits the list into two sub-lists `Small` and `Large` about a pivot element `Pivot`, recursively sorts the two sub-lists and uses append to concatenate the sorted sub-lists. The two recursive goals in the body of quicksort may be solved in parallel without interfering with each other, if we know that the arguments `Small`, `Large`, `SS`, and `LS` do not share non-ground terms *i.e.*, terms that have variables.

```
quicksort([], []).
quicksort([Pivot|Unsorted], Sorted) :-
    split(Unsorted, Pivot, Small, Large),
    quicksort(Small, SS),
    quicksort(Large, LS),
    append(SS, [Pivot|LS], Sorted).
```

The simplest way to verify that the two recursive calls to quicksort are independent is to test at runtime whether the two goals have unbound variables in common. However, run-time tests for independence of terms can be costly: $O(m+n)$ where m and n correspond to the size of terms. Run-time analyses of programs, in general, can have significant space and time overheads every time the program is executed. Compile-time or static analyses, on the other hand, incur the space time overheads just once during the compilation process. If programs are compiled only a few times but executed many times, static analyses are preferable to run-time analyses.

Three types of static analysis may help us determine at compile time that two goals will be independent:

groundness A ground term does not contain any variables and hence it is independent of any other term.

sharing Two nonground terms are independent if they have no variables in common, *i.e.,* the same variable does not occur in both terms.

freeness Two goals that share a variable may be solved in parallel if one goal does not bind the variable to a nonvariable term.

Static analysis for groundness, sharing, and freeness can be performed using an abstract interpretation framework based on an operational or a denotational semantics of programs. Abstract interpretation provides a new, more abstract semantics for the language under consideration and shows how the meaning of a program in the new semantics is an abstraction of the meaning of the program in standard semantics. This is usually done by providing an abstract domain of computation which captures the properties of interest. Operations over the abstract domain are analogs of the standard operations over the standard domain.

1.1 Contributions

This dissertation provides solutions to two problems related to the parallel execution of logic programs.

The first issue deals with deriving data dependencies in logic programs. We define an abstract domain to express sharing, freeness, and groundness of variables and use this domain within an abstract interpretation framework to derive these properties for a given program. The second issue is the simultaneous derivation of data flow and control flow analysis. All flow analyses of logic programs to date assumed that either control flow or data flow information is available. We propose a framework for combined derivation of control flow and data flow information. The framework is instantiated with a simple domain of input output modes and experimentally evaluated. Note that although the definition of input output modes used for experimentation is restrictive, it is not a limitation of the framework itself. More expressive mode systems can be coupled to the framework easily. We now discuss the two issues of precise sharing analysis and simultaneous data flow and control flow analysis in more detail.

Many researchers have proposed sharing analyses that treat sharing as a transitive relation mainly to realize worst-case polynomial time complexity. In sharing analysis of logic programs, one approach to a tractable analysis is to treat sharing as transitive. This loses precision considerably. Our approach starts with a domain proposed by Jacobs and Langen [44] where sharing is treated as transitive. We then add information about linearity and freeness of terms. A term is considered *linear* if it does not contain multiple occurrences of the same variable. This enables sharing to be treated as a non-transitive relation, thus leading to a more precise abstract unification algorithm. Our analysis has polynomial time complexity for a class of programs that induce what we call "stable call substitutions." We point out the sources of combinatorial explosion in the case of programs with unstable substitutions, and specify a widening operation and a compact representation. The result is graceful degradation, automatic and selective loss of precision only where needed, and reasonable running times.

The increased precision of our data dependency analysis has a beneficial consequence. It actually results in faster convergence of fixed-point computations, contrary to the usual expectation that increased precision can only be obtained from more expensive analyses. The number of fixed-point iterations, in general, is proportional to the height of the abstract domain. For our domain, it is exponential in the number of variables in a clause. In practice, however, the number of iterations is observed to be independent of the number of variables and bounded by five. This is reinforced by the experimental results of other researchers such as Van Hentenryck [39], and Hermenegildo, Warren and Debray [88]. Ullman [2] reports similar results for intra-procedural analysis of imperative programs.

Since there are programs which cause the analyzer to converge slowly, we use a *widening* operation proposed by Cousot and Cousot [21] that accelerates convergence when computing that part of the fixed-point which is reachable in top-down computations. Its usefulness is illustrated in the case of programs whose analyses tend to wander around the abstract space. The proposed widening operation to accelerate the convergence of fixed-point computations is very general and independent of the language and the abstract domain. Our domain-independent, optimized abstract interpreter and the proposed sharing, freeness, and groundness analysis together with the widening operation perform as well as and in many cases better than other sharing, groundness and freeness analyses that have been proposed, both in terms of the precision of the analysis and the time taken for analysis.

All data flow analyses to date assume a fixed control flow, *i.e.*, they assume that the sub-computations will be performed in a certain sequence and accordingly derive the data dependencies among the sub-computations. These dependencies are in turn used to relax the total ordering among the sub-computations. However, assuming an a *priori* fixed control flow is too strong a restriction. It precludes many possible computation orderings and leads to ill-moded programs. Informally, an ill-moded program is one where the control flow induces a data flow that does not satisfy input-output restrictions. Reordering ensures the well-modedness of programs. Until now, there has been no framework for deriving data flow and control flow at the same time. We define a novel framework for simultaneous derivation of data flow and control flow in logic programs. The cyclic dependency between data flow and control flow is broken by first defining the notion of permissible modes in a way that does not depend on a fixed control flow and then deriving control flow based on permissible modes.

Our research on a combined data and control flow analysis is motivated by an apparent circular dependency between these two analyses. Consider the quicksort example again and assume that the control flow is known; for example, body goals are solved left-to-right. If quicksort is always called from other parts of a program with the first argument bound to a ground term, data flow analysis will show that when the subgoal split is solved, the two sub-lists Small and Large will be bound to ground terms, and that the variables SS and LS will remain unbound and independent of each other. It is easy to see that the second and third subgoals (recursive calls to quicksort) depend on the first goal split. Likewise, the fourth goal (append) depends on the second and third subgoals.

This information is derived assuming left-to-right execution. The control flow can now be redefined and so the two recursive calls may be solved in parallel. Given the control flow, one can perform a forward analysis of the data flow through the program, *i.e.*, propagate any information about the entry point to other points in the program.

Likewise, if data flow information is known, the derivation of control flow is not difficult. In the quicksort example, assume that quicksort is to be called always with the first argument ground, split with the first and second arguments ground and finally append is to be called with the first and second argument ground. Given this information, one can derive the same control flow for quicksort even if the body goals are permuted. In the general case, however, a given data flow may be realized by more than one control flow.

An execution of a logic program may bind variables in a goal to values or terms constructed from variables, constant symbols, and function symbols. An association between a variable and its value is known as a binding. An environment binds variables to values. The environment in which a goal or a procedure call is executed is known as the call substitution or environment. The following example illustrates the need for different orderings of body goals of a clause depending on the environment of the call.

Consider the permutation relation between two lists. An empty list [] is a permutation of itself. The list [Y|Ys] is a permutation of a list [X|Xs], if there exists a list Zs and an element Y of [X|Xs] such that Zs has the same elements as [X|Xs] but without Y and the list Ys is a permutation of Zs. This is a declarative reading of the relation perm.

```
1.  perm([], []).
2.  perm([X|Xs], [Y|Ys]) :-
        delete([X|Xs], Y, Zs),
        perm(Zs, Ys).
3.  delete([A|Rest], A, Rest).
4.  delete([A,B|Bs], X, [A|Rest]) :-
        delete([B|Bs], X, Rest).
```

Assume that body goals are executed left-to-right. If perm is called with the first argument bound to a list of terms, it will succeed and unify the second argument to a permutation of the first. For example, the goal perm([a,b,c,d],A) will succeed and bind A to [a,c,d,b] as one of many answer substitutions.

Unifying the goal perm([a,b,c,d],A) with the head of the second clause of perm binds the head variables X and Xs to the terms a and [b,c,d] respectively, *i.e.*, the head literal *generates* or *produces* X and Xs. After head unification, solving the first body goal delete([a,b,c,d],Y,Zs) binds Y to a and Zs to [b,c,d]. Delete generates Y and Zs, where Y is *consumed* by the head literal and Zs by the second body goal perm. Finally, the second body goal perm([b,c,d],Ys) binds Ys which is consumed by the head literal.

The data flow induced by the the left-to-right control flow and the initial goal perm([a,b,c,d],A) is shown in Figure 1.1. In a data flow graph, each node is a literal and

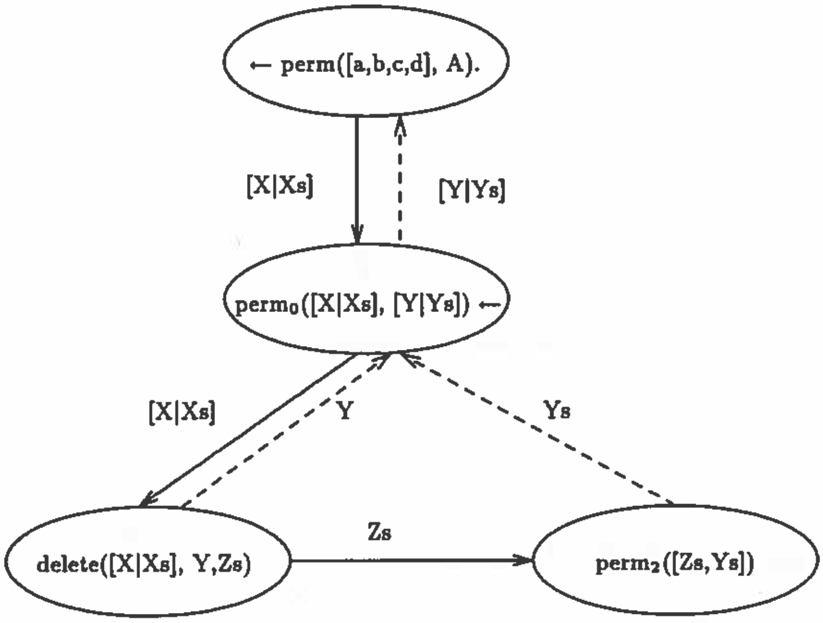

Figure 1.1: Data Flow Graph for **Perm** Induced by Left-to-Right Control Flow

there is directed edge from a node i to another node j labeled with variable X iff literal i produces a binding for the variable X which is consumed by j. We have two types of edges in Figure 1.1, solid and dashed. They both denote the producer-consumer relationship. Dashed edges signify that the variable is produced by a body goal and consumed by the head or passed by the head literal to the caller's environment. Analogously, a solid edge signifies the data flow into and within a clause. They provide a visual clue about the directionality of data flow into and out of a clause.

Although no input-output directionality is implied by the declarative reading, from an operational viewpoint we say that the first argument of **perm** has *input mode*, *i.e.,* the first argument of **perm** will be bound to a ground term at call time. Similarly, the second argument of **perm** has *output mode*, *i.e.,* the second argument will be bound to a ground term if and when **perm** succeeds. We use *perm*(+,−) to indicate that the first argument of **perm** is input to the procedure and the second argument is output. Notice that, with the left-to-right control flow, the mode *perm*(+,−) induces the mode *delete*(+,−,−) for **delete** and *perm*(+,−) for the recursive call in the body of **perm** which is consistent with the mode for the initial call to **perm**.

In Chapter 2, we will define SLD-trees which show all computations of a program and a goal given a control flow. With a left-to-right strategy for solving body goals, the SLD-tree for the above program and the goal `perm([a,b,c,d],A)` does not have any infinite branches. The order in which the clauses of **delete** are used does not affect the number or set of solutions, declaratively speaking. In terms of SLD-trees, the number of successful paths or derivations remain the same. Operationally, a strategy for constructing or exploring an SLD-tree may miss some solutions, *i.e.,* the strategy may be incomplete.

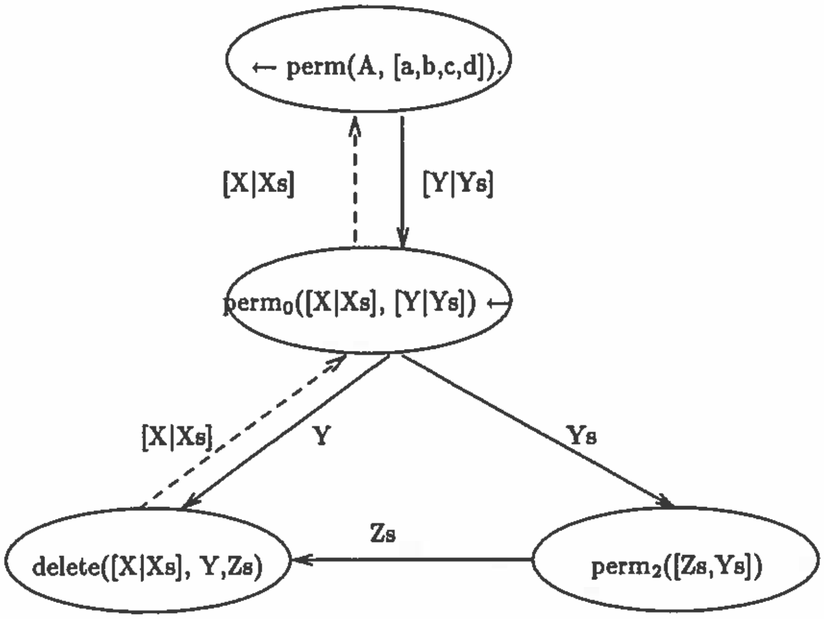

Figure 1.2: Data Flow Graph for Perm Induced by Right-to-Left Control Flow

Since permutation is a symmetric relation, one may be tempted to call perm with the second argument bound to a list of terms and leave the first one free and expect perm to bind the first argument to a permutation of its second argument. Unfortunately, there is a problem. Assume the same control flow as in the first case. If the clauses of delete are used in some fixed order, some solutions will be missed. This is because there are infinite computation paths and any depth-first search (such as the one induced by trying the clauses of delete in some fixed order) will miss some solutions.

After perm produces the first solution, delete will go into an infinite loop, if the alternative clauses of a predicate are tried as they appear in the program. This is because solving the body goals of perm in left-to-right order causes delete to be called with its first and third arguments being unbound variables; solving delete with left-to-right control flow and with the arguments one and three unbound causes it to loop indefinitely, given the assumptions.

However, if we use a right-to-left control flow, an initial call to perm with the mode perm(−,+) induces the same mode for the recursive call to perm and the mode *delete*(−,+,+) for delete. When delete is called with the second and third arguments bound to ground terms, it terminates and produces a value for the first argument. delete(A,B,C) is used to insert B somewhere in the list C producing A in the mode *delete*(−,+,+). The data flow for the second clause of perm induced by the right-to-left control flow is shown in Figure 1.2.

Note that delete(A,B,C) was used to delete some element B from A producing C in the mode *delete*(+,−,-). Thus, delete may be used with the first argument in input mode and the other two in output mode, or the other two in input mode and the first in output mode. These two modes of delete were induced by the two modes of perm, each of which needed a different control strategy. Thus, it is clear that control flow depends on data flow.

The unsolvability of the halting problem implies that in general there is no algorithm for choosing statically the right order to ensure the termination of a logic program. Hence, we will not make termination of programs an objective of our control flow analysis. Instead, the objective is to derive a control flow that induces well-modedness for a clause. A clause is well-moded iff there is a partial ordering of the body goals that satisfies the input output restrictions of the body goals. How do we know what the input and output positions of a body goal should be? We introduce the notion of permissible modes, based on some restrictions on the arguments of procedures. The permissible modes do not define what input output modes are; the definition of modes is one of the parameters of the framework discussed below.

In the second part of this dissertation, we propose a framework for deriving data flow and control flow simultaneously. The framework first derives permissible modes using properties not dependent on execution order. The permissible modes may then be used to derive control flow. We define a sound approximation algorithm for computing a set of permissible input output modes which induce well-moded clauses. The worst-case time complexity of the algorithm is bounded by a polynomial in the number of literals in a clause and the arity of the head predicate.

The space of permissible modes of a procedure may be partially ordered consistent with the constraints imposed by the modes. Some permissible modes are less restrictive than others, *i.e.,* impose fewer constraints. Minimal permissible modes are those that do not impose more constraints than any other permissible modes. We show that deriving the set of minimal permissible modes of a clause is NP-hard. Our permissible mode derivation algorithm is an approximation algorithm in the sense that it tries to derive minimal permissible modes but some times it may derive permissible modes that are not minimal and may not derive all the permissible modes.

We suggest an approach similar in spirit to polymorphic type inferencing and reconstruction used in languages such as Standard ML [59]. The programmer may omit permissible mode declarations most of the time and let the compiler derive them automatically. When the compiler fails to derive a desired permissible mode for a procedure, the programmer must add the missing ones. Empirical results and analysis of the time complexity of the permissible mode derivation algorithm show this approach to be tractable and useful. Furthermore, the analysis of a number of programs shows that our method derives all minimal permissible modes of most procedures. For example, our algorithm finds the two minimal permissible modes for the perm program and the right order of body goals associated with those permissible modes.

The definition of permissible modes and the framework for control and data flow analysis do not depend on the particular definition of modes we use. Equating input output modes with ground terms is done solely for experimentation.

1.2 Thesis Organization

In the rest of this chapter, we cover mathematical preliminaries, and discuss the denotational approach to defining semantics of programming languages.

In chapter two, we start with a discussion of the syntax and semantics of Horn Clause

Logic Programming. We then discuss an operational semantics (based on SLD-resolution) and a denotational semantics for logic programs. The denotational semantics will be used as the basis for deriving an abstract interpretation framework in chapter three. Finally, we present abstract interpretation as a semantics-based framework for data flow analysis.

In chapter three, we discuss the issues involved in deriving sharing, groundness, and freeness information, propose an abstract domain and operations for that purpose and prove soundness.

In chapter four, we study the efficiency of the proposed data flow analysis. We derive the worst-case time complexity of our analysis, and compare it with the theoretical bounds. A standard algorithm to compute the meaning of semantic functions would wastefully compute the value of the function at all points. We study Minimal Function Graph (MFG) semantics wherein the values of the semantic functions are computed only at the needed points, which constitute a tiny fraction of the total number of points.

Cousot and Cousot [21] proposed a widening operation that accelerates the convergence by comparing (and widening, if needed) successive iterates of fixed-point computations. As mentioned before, the number of fixed-point iterations is usually very small (less than five) and hence the time complexity of our analysis is not dominated by the number of iterations in practice. It is dominated by the analysis time for one iteration, which in our case can be exponential in the size of a clause. We define a widening operation to be used when computing the MFG semantics. For a large number of programs, our analysis reaches convergence with a small number of fixed-point iterations. These programs are well-behaved with respect to our abstract domain in the sense that an abstract substitution (or environment) for an initial call to a recursively defined predicate and the abstract substitutions for the recursive calls derived from it are the same when restricted to the variables in the goal. This means that the fixed-point computations can be shared. Referring to the quicksort example, a call to quicksort together with an abstract substitution in which the first argument is ground, induces recursive calls with the same property.

Our proposed widening operation achieves convergence much faster in the case of not so well-behaved programs. Indeed, without the widening operation the analysis of such programs tends to wander around the abstract space, *i.e.*, an initial call to a recursive predicate together with its abstract environment induce calls to the same predicate with incomparable abstract environments. Analyses of programs such as self-adjusting binary search trees exhibit this behavior and hence benefit from the widening operation.

In chapter five, we discuss the problems in deriving control flow and illustrate the inter-dependency of control flow and data flow analyses. After defining the notion of well-modedness and permissible modes, we show that the problem of deriving minimal permissible modes is NP-hard. We propose a framework and algorithm for deriving permissible modes, and prove its soundness. Our algorithm may derive non-minimal permissible modes but has polynomial time worst-case complexity. Thus, we trade off minimality for tractability. To the best of our knowledge, ours is the first formal framework which defines and derives permissible modes *without* assuming that either data flow or control flow is known.

We then discuss the derivation of control flow given the permissible modes for all procedures and suggest heuristics for those cases where the permissible modes induce more than one ordering of body goals.

The last chapter summarizes the contributions, discusses the limitations of our approach and suggests directions for further research.

1.3 Mathematical Preliminaries

In this section, we provide a brief introduction to domain theory and define some of the terms used in later sections.

Definition 1.1
A function $f : A \rightarrow B$ is *injective* (one-to-one) iff $\forall x, y \in A.\ f(x) = f(y) \Longrightarrow x = y$. Function $f : A \rightarrow B$ is *surjective* (onto) iff $\forall y \in B.\ \exists x \in A.\ f(x) = y$. f is *bijective* iff it is both injective and surjective. □

Definition 1.2
A relation $\sqsubseteq$ is a *partial ordering* on a set D iff $\sqsubseteq$ is:

- reflexive: $\forall x \in D.\ x \sqsubseteq x$.
- antisymmetric: $\forall x, y \in D.\ x \sqsubseteq y$ and $y \sqsubseteq x \Longrightarrow x = y$.
- transitive: $\forall x, y, z \in D.\ x \sqsubseteq y$ and $y \sqsubseteq z \Longrightarrow x \sqsubseteq z$.

□

For $x, y \in D$, we read $x \sqsubseteq y$ as saying "x approximates y." A set equipped with a partial ordering is usually referred to as a poset or partially ordered set. The notation $\sqsubseteq_D$ indicates that we are talking about the partial ordering defined on D, but when there is no room for confusion, the subscript D will be omitted.

Definition 1.3
Let $(D, \sqsubseteq)$ be a poset, $X \subseteq D$ and $u, l \in D$. Then,

- $X \sqsubseteq u$, if $\forall x \in X.\ x \sqsubseteq u$.
- $l \sqsubseteq X$, if $\forall x \in X.\ l \sqsubseteq x$.

□

The element u (l) is said to be the *upper bound* (*lower bound*) of the set X.

Definition 1.4
Let $(D, \sqsubseteq)$ be a poset. An element $x \in D$ is the *least upper bound* (lub) of a subset X of D iff:

- $X \sqsubseteq x$ and
- $\forall y \in D.\ X \sqsubseteq y \Longrightarrow x \sqsubseteq y$.

□

The least upper bound of X is denoted as $\sqcup X$. The *greatest lower bound* (glb) is denoted as $\sqcap X$ and may be defined analogously to the least upper bound. A lub (glb) is unique if it exists. When we take the lub of a set containing two elements, we write $x \sqcup y$ (read as x join y) instead of $\sqcup\{x, y\}$. Likewise, we write $x \sqcap y$ (read x meet y) instead of $\sqcap\{x, y\}$.

Definition 1.5

Let $(D, \sqsubseteq)$ be a poset. A subset X of D is a *chain* iff X is non-empty and $\forall x, y \in X.\ x \sqsubseteq y$ or $y \sqsubseteq x$. □

All elements of a chain are related to each other, *i.e.*, a chain is totally ordered.

Definition 1.6

A poset $(D, \sqsubseteq)$ is a *complete partial order*, cpo, iff $\forall$ chain $X \subseteq D.\ \sqcup X$ exists in D. □

Definition 1.7

A poset D is a *pointed cpo* iff D is a cpo and $\exists \bot \in D.\ \bot \sqsubseteq D$. □

Definition 1.8

A poset $(D,\sqsubseteq)$ is a *join semi-lattice* iff for all subsets X of D, the lub of X exists in D. D is a *meet semi-lattice* iff for all subsets X of D, the glb of X exists in D. D is a *complete lattice* iff D is both a join semi-lattice and a meet semi-lattice. □

Definition 1.9

For cpos A and B, a function $f : A \rightarrow B$ is *continuous* iff for any chain $X \subseteq A$, $f(\sqcup X) = \sqcup\{f(x) | x \in X\}$. □

In other words, the image (under f) of the least upper bound of a chain X contains the same information as the least upper bound of the images (under f) of the elements of X. Continuous functions preserve the limits of chains.

Definition 1.10

A function $f : A \rightarrow B$ where A and B are posets is *monotonic* iff $\forall x, y \in A.\ x \sqsubseteq_A y \Longrightarrow f(x) \sqsubseteq_B f(y)$. □

Continuity implies monotonicity. When the underlying set is finite, monotonicity and continuity are equivalent. Continuity allows us to work with infinite objects in terms of their finite approximations. We will say more about continuity shortly.

Definition 1.11

A *domain* is a pointed cpo over which computable functions are continuous. □

One can construct complex domains out of simpler ones. We now define a few domain constructions that will be needed later.

Any set D can be equipped with a discrete partial ordering $\sqsubseteq$ defined as: $\forall x, y \in D.\ x \sqsubseteq y$ iff $x = y$. Such a set is a cpo and is known as a flat cpo. For a poset $(A, \sqsubseteq_A)$, its *lifting* $A_\bot$ is the set $A \cup \{\bot\}$, partially ordered by the relation $\sqsubseteq$ such that $\forall x, y \in A_\bot.\ x \sqsubseteq y$ iff either $x = \bot$ or $x \sqsubseteq_A y$. The newly added element $\bot$ is known as the *improper* element and all other elements are *proper*.

For posets $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$, their product $A \times B$ is the set $\{(a, b) \mid a \in A\text{and} : b \in B\}$ partially ordered by the relation $\sqsubseteq$ such that $(a, b) \sqsubseteq (a', b')$ iff $a \sqsubseteq_A a'$ and $b \sqsubseteq_B b'$. Domain construction by products can be generalized to products of n domains, where $n > 2$. We write $(x_1, x_2, \ldots, x_n)$ to denote an element of $A_1 \times A_2 \times \ldots \times A_n$. In the special case of $A_1 = A_2 = \ldots = A_n$, we write the domain as A^n. For a product of n domains, we use $\downarrow i$ as the projection operation: $(x_1, x_2, \ldots, x_n) \downarrow i = x_i$.

For posets $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$, define a function space $A \rightarrow B$ (read "*A to B*") to be the set of all continuous functions with domain A and co-domain B. If A and B are cpos, then $A \rightarrow B$ is a cpo. These functions are partially ordered by the relation $\sqsubseteq$ such that for $f, g : A \rightarrow B$,

$$f \sqsubseteq g \quad \text{iff} \quad \forall a \in A \;\; f(a) \sqsubseteq_B g(a).$$

We always use the pointwise ordering on function space, *i.e.*, for functions $f, g : X \rightarrow Y$, $f \sqcup g$ is a function $h : X \rightarrow Y$ such that $\forall x \in X.h(x) = f(x) \sqcup g(x)$. This follows from the definition of $\sqsubseteq$ above.

If D is a domain, D^* denotes the domain of finite sequences of $d \in D$. If D is a set, then its power set $\mathcal{P}(D)$ is a domain with subset relation as the partial ordering, unless noted otherwise.

Least Fixed-Points

In the previous section, we defined domains and some domain constructors and the continuity of functions. Continuity is useful when computing with infinite objects and necessary from the point of view of computability. In the denotational definition of the semantics of a programming language, the concepts of continuity and fixed-points of functions play an important role. The meanings of recursively specified functions (and domains) will be defined in terms of least fixed-points.

For a function $f : D \rightarrow D$, an element $d \in D$ is a fixed-point of f if $d = f(d)$. If D is a domain and $f : D \rightarrow D$ is a continuous function, then there always exists a $d \in D$ such that d is the least fixed-point of f, *i.e.*, $f(d) = d \land \forall e \in D.\; e = f(e)$ implies $d \sqsubseteq e$. Let f^i denote f composed with itself i times, *i.e.*, $f^i(x) = f(f(f \ldots (f(x)) \ldots))$ and $\perp$ denote the least element of the domain D. Then the least fixed-point of f, *fix* f, is defined as

$$\mathit{fix}\, f = \sqcup\{f^i(\perp) \mid i \geq 0\}$$

The existence of the least fixed-points is known as Kleene's fixed-point theorem [76].

The construction of least fixed-points is illustrated now. Consider a recursive specification of the factorial function.

$$\mathit{fac} : N \rightarrow N_\perp$$

$$\mathit{fac} = \lambda n.\, n = 0 \rightarrow 1 \mid n > 0 \rightarrow n * \mathit{fac}(n-1)$$

In this definition, the occurrence of *fac* on the right hand side of the equation is *free*, *i.e.*, neither a parameter of the function, nor a local variable, in programming language terms. In all recursive function definitions which associate a name with a function and use that name in the definition of the function (such as the above), the name of the function will be free in the definition.

The meaning of a function f is given as the least fixed-point of an associated functional F. We derive a functional F from the recursive definition of f by abstracting out f, *i.e.,* by converting f into a parameter. For example, the meaning of *fac* is defined to be the least fixed-point of the following functional F, obtained by abstracting out *fac* from the specification.

$$F : (N \to N_\perp) \to (N \to N_\perp)$$

$$F = \lambda f.\, \lambda n.\, n = 0 \to 1 \mid n > 0 \to n * f(n-1).$$

What is $F(\mathit{fac})$? Applying F to *fac*, we get

$$\lambda n.\, n = 0 \to 1 \mid n > 0 \to n * \mathit{fac}(n-1).$$

We see that $F(\mathit{fac}) = \mathit{fac}$ and we have recovered the original definition of factorial as a fixed-point of F.

Continuity is useful in computing with infinite objects. The meaning of a recursive function, an infinite object, is solely determined by the meanings of its finite approximations, *i.e.,* as the lub of successive, finite approximations. We can compute the finite approximations as much as needed. The following example shows, in part, how continuity is useful in computing with infinite objects.

Consider the functional F associated with the factorial function. We list below the successive approximations $F^0, F^1, F^2, \ldots$ of the factorial function.

$$\begin{array}{lcll}
F^0 & = & \perp & \text{by definition of } F^0 \\
F^1 & = & F(F^0) & = F(\perp) = \lambda n.\, n = 0 \to 1 \mid n > 0 \to \perp \\
F^2 & = & F(F^1) & = F(\lambda n.\, n = 0 \to 1 \mid n > 0 \to \perp) \\
 & & & = \lambda n.\, n = 0 \to 1 \mid n > 0 \to n * ((\lambda n.\, n = 0 \to 1 \mid n > 0 \to \perp)(n-1))
\end{array}$$

The $F^0, F^1, F^2, \ldots$ are finite mappings from N to $N_\perp$ and correspond to unfolding the recursive definition of *fac* 0, 1, 2, ... times respectively. F^0 maps all natural numbers to $\perp$. F^1 maps 0 to 1 and all other natural numbers to $\perp$. Likewise, F^2 is defined for 0 and 1 and undefined for all other numbers. We can represent these finite functions by their graphs *i.e.,* sets of pairs of arguments and results. The graphs of these approximations are as follows. We do not show the pairs of the form $(n, \perp)$.

$$F^0 = \perp, F^1 = \{(0,1)\},\ F^2 = \{(0,1),(1,1)\},\ F^3 = \{(0,1),(1,1),(2,2)\}, \ldots$$

Given the partial ordering on function space, it is easy to see that $F^0 \sqsubseteq F^1 \sqsubseteq F^2 \ldots$ forms an infinite chain. The meaning of factorial is the lub of this chain. The application of the factorial function to some $n \in N$ involves computing the finite approximation F^{n+1} and applying F^{n+1} to n. Operationally, it amounts to $n+1$ unfoldings of *fac*.

The need for cpos arises from the fact that the successive approximations of a recursively defined function form an infinite chain and the meaning of such a function is the least upper bound of the chain. We want the lubs of chains to exist and hence domains must be complete partial orders. We also want the least element to exist so that successive approximations can be computed starting with the least element. Hence, domains are defined to be pointed cpos.

We will not go into the details of computability of continuous functions but refer the reader to an excellent introductory book by Manna [53]. Our review of domain theory is necessarily incomplete. Domain theory, developed by Dana Scott, is general enough to model computation, including recursion and self-application. For a detailed discussion of domain theory, please see [76, 72, 80]. With this brief review of domain theory, we are ready to discuss a small example of denotational definition.

1.4 Denotational Semantics

The semantics of a programming language may be defined using operational, axiomatic or denotational approaches. In all approaches, the meaning of a program in a given language is defined in terms of the meanings of the language constructs.

In the denotational approach to semantics, the meanings of language constructs are defined by *valuation* or *semantic* functions. Valuation functions map syntactic constructs to semantic domains. The meaning of a construct is defined in terms of the meanings of its proper sub-parts. We need a notation to define the valuation functions and semantic domains. Lambda calculus is the language most often used for this purpose.

A denotational definition of a language consists of three parts: abstract syntax, semantic domains, and valuation functions that map syntactic elements to their meanings in the semantic domain.

Abstract syntax defines the structure of the sentences of a language in terms of its components which may be other syntactic categories or tokens. The rules for grouping individual alphabet symbols or terminals into words and the terminals themselves are omitted, since these details are irrelevant when defining the semantics. For example, the structure of an arithmetic expression may be defined by rules $\langle aexp \rangle$ ::= $\langle aexp \rangle$ *plus* $\langle aexp \rangle$ and $\langle aexp \rangle$ ::= $\langle numeral \rangle$. The structure of the tokens *numeral* and *plus* is unimportant.

Examples of semantic domains include integers, truth values, ordered pairs, and continuous functions over these spaces.

As an example, we define the denotational semantics of binary numerals. We have to specify three things, namely, the abstract syntax, the semantic domains and the valuation functions. The abstract syntax of binary numerals is given by the following BNF grammar, where *0* and *1* are the numerals, B stands for the syntactic category Binary Numeral and D for Binary Digit. Since a binary digit does not have any structure, there is no difference between the abstract syntax and the concrete syntax.

B:	Binary Numeral	::=	BD
B:	Binary Numeral	::=	D
D:	Binary Digit	::=	*0*
D:	Binary Digit	::=	*1*

For the semantic domain, we use the natural numbers N together with the operations *times* and *plus*. We now have to show the correspondence between syntactic objects and semantic domain elements and use semantic functions for this purpose. The semantic functions are given by a set of equations, one for each production defining the syntactic category such as B or D. The semantic function corresponding to a syntactic category B is denoted by boldface **B**. The semantic functions have the following types:

B: Binary Numeral → N
D: Binary Digit → N

Before specifying the semantic functions, we describe a few conventions about their form. Let f be a function of two arguments. Then its curried version is a function f' such that $(f'(x))(y) = f(x, y)$, *i.e.*, f', a function of one argument, when applied to x returns an unnamed function, which when applied to y yields a result that is equivalent to $f(x, y)$. We use functions in their curried form, *i.e.*, the application of f to x and y will written as $f\,x\,y$ instead of $f(x, y)$ and the type of a function $f : X \times Y \rightarrow Z$ will be written as $f : X \rightarrow Y \rightarrow Z$. A semantic function may have parameters from syntactic and semantic domains. It is customary to enclose the syntactic parameters of a semantic function in brackets **[ ]**. We now define the semantic functions **B** and **D**:

$$\begin{aligned}
\mathbf{B}[\![\mathrm{BD}]\!] &= (\mathbf{B}[\![\mathrm{B}]\!] \times 2) + \mathbf{D}[\![\mathrm{D}]\!] \\
\mathbf{B}[\![\mathrm{D}]\!] &= \mathbf{D}[\![\mathrm{D}]\!] \\
\mathbf{D}[\![\mathit{0}]\!] &= 0 \\
\mathbf{D}[\![\mathit{1}]\!] &= 1
\end{aligned}$$

Note that the meaning of a syntactic object is defined either directly (*i.e.*, mapped to an object in the semantic domain, as in the case of D) or as the result of some function(s) applied to the meanings of its sub-parts (as in the case of B).

1.5 Summary

This chapter discussed our contributions, provided an outline of the dissertation, and recalled some mathematical definitions. The next chapter presents the syntax and semantics of Horn Clause Logic Programming, and introduces abstract interpretation of logic programs.

Chapter 2

Syntax and Semantics of Logic Programs

This chapter provides an introduction to the syntax and semantics of logic programs. We discuss an operational semantics for logic programs based on SLD-resolution and illustrate some potential problems with the selection of a control strategy. Since the operational semantics is not convenient for the types of analyses we are interested in, we define a denotational semantics for logic programs. The denotational semantics is used as the basis for an abstract interpretation framework. We use a parallel composition operator in both the operational and denotational semantics. The parallel composition operator replaces the standard notion of composition and is convenient to use for a variety of reasons.

2.1 Logic Programming

The model of computation represented by logic programming is radically different from procedural programming based on the von Neumann model. A logic program is a set of axioms of a first order theory and computation may be viewed as an attempt to prove that a given formula logically follows from the given set of axioms.

Logic Programming owes its development to Kowalski and Colmerauer [48]. Kowalski showed that a formula of the form $p \leftarrow q_1, \ldots, q_n$ has a procedural interpretation. The set of clauses defining p is seen as defining the procedure p and the goals $q_1, \ldots, q_n$ are viewed as procedure calls. Parameter passing, data structure construction and selection and returning results are all achieved using one mechanism, namely, unification. Colmerauer developed the logic programming language Prolog and an interpreter for it as part of a research effort in natural language processing.

A computational model of a logic programming system may be split into two parts: a declarative part which represents the constraints of the system being modeled and a procedural part which manipulates these constraints. Such a separation of declarative knowledge and procedural knowledge occurs naturally in many domains and Prolog makes it easier to represent and reason about such knowledge. Prolog has been used successfully in both academia and industry for a variety of purposes.

The field of logic programming reaches far beyond the language Prolog. To mention a few, the fields of deductive data bases [40], non-monotonic reasoning [65], constraint logic programming [19, 35], committed-choice logic languages [73], and higher-order logic programming [43, 63, 58, 15], owe their development to research in logic programming and shortcomings in Prolog. The convenience and expressive power of unification has led researchers to explore various combinations of functional and logic programming paradigms [7, 24, 37].

2.2 Syntax and Semantics

Logic programming languages use a subset of the sentences of First Order Predicate Calculus together with one or more inference rules to derive logical consequences.

In predicate calculus, a *literal* is an atomic formula (*atom*) or the negation of an atomic formula. For example, $p(t_1, \ldots, t_n)$ is an atomic formula, and $\neg p(t_1, \ldots, t_n)$ is a negated atom. In an atom $p(t_1, \ldots, t_n)$, p is the *predicate symbol* and $t_1, \ldots, t_n$ are its *arguments*, which are *terms*. A term t is either a *variable* or of the form $f(t_1, \ldots, t_k)$ where f is a *function symbol* of arity k and $t_1, \ldots, t_k$ are all terms. When k is zero, we omit the parentheses, and f is called a *constant*.

A term (atom) is a ground term (atom) if and only if no variable occurs in it. Constants, function and predicate symbols are denoted by identifiers starting with lower case letters, and a variable by an identifier starting with an upper case letter. We use a ranked, countably infinite and mutually disjoint alphabet of predicate symbols and function symbols. The rank of a predicate is its arity. Note that a predicate symbol such as p may have only one arity. This is not a restriction because a consistent renaming of the predicate symbols in a program will ensure that a predicate symbol has associated with it only one arity.

A *clause* is a disjunction of literals, where all the variables are universally quantified in prenex form. For example, $\forall X. \neg q_1(X) \vee \neg q_2(X) \vee p(X) \vee r(X)$ and $\forall X. \neg p(X)$ are clauses. A clause with at most one positive literal is a *Horn Clause*. The clause $\forall X. \neg q_1(X) \vee \neg q_2(X) \vee p(X) \vee r(X)$ is not a Horn clause since it has two positive literals. A clause with exactly one positive literal is also known as a *definite clause* or *program clause* and a clause with zero positive literals is a *goal clause*. For example, $\forall X.\, \neg q_1(X) \vee \neg q_2(X) \vee p(X)$ is a program clause and $\forall X.\, \neg p(X)$ is a goal clause.

Hereafter, we use the term clause to mean a Horn clause. Since all variables in a clause are universally quantified, we omit the quantifiers. Furthermore, since $a \rightarrow b \equiv \neg a \vee b$, the clause $p \vee \neg q_1 \vee \neg q_2 \vee \ldots \vee \neg q_n$ will be written as $p \leftarrow q_1, q_2, \ldots, q_n$, where $\leftarrow$ is read as "if" and comma stands for conjunction. The positive literal of a clause is known as its *head*, and the negative literals are collectively known as the *body*.

When the body of a clause is empty, the clause is known as a *unit clause*. A goal clause will be written as $\leftarrow b_1, \ldots, b_n$ and a unit clause as $p \leftarrow$. A *procedure* p is a set of n-ary clauses whose heads have the same n-ary predicate symbol p. The literals in the body of a clause are often referred to as *goals* or *subgoals*.

Example 2.1

Consider the following program which defines natural numbers.

```
1.  nat(0) ←
2.  nat(succ(X)) ←
        nat(X).
3.  ← nat(succ(succ(succ(0)))).
```

□

In the above example, nat(0) is an atom where nat is a predicate symbol of arity one and 0 is a constant term. The atom nat(0) is a ground atom since no variables occur in it. The atom nat(X) is not ground since it contains a variable term X. The atom nat(succ(0)) is also ground. Its argument is a term whose function symbol is succ of arity one. We may some times refer to terms such as `succ(0)` as a compound term.

The clause `nat(succ(X))` $\leftarrow$ `nat(X)` is a Horn clause. The clauses `nat(0)` and `nat(succ(X))` $\leftarrow$ `nat(X)` together form a procedure `nat` which defines the notion of a natural number. The clause `nat(0)` $\leftarrow$ is a unit clause, since its body is empty. The positive literal `nat(succ(X))` of the clause `nat(succ(X))` $\leftarrow$ `nat(X)` is known as its head, and the negative literal `nat(X)` as its body; `nat(X)` is also the only subgoal of the second clause. The clause $\leftarrow$ `nat(succ(succ(succ(0))))` is a goal clause.

A logic program is a finite set of Horn clauses. It can be given meaning in three different ways. The clause $p \leftarrow q_1, \ldots, q_n$ can be read *procedurally* as follows. "To prove p, prove $q_1, q_2, \ldots,$ and q_n." This proof process is a constructive one, *i.e.,* when it succeeds it produces values for the variables of the clause. In the procedural reading, the meaning of an n-ary procedure p in a program is defined to be a relation: a set of n-tuples of terms such that the relation is provable, given the clauses of the program as axioms.

From a *model-theoretic* point of view, the clause can be read as: "p is true if $q_1, \ldots,$ and q_n are." In this view, the meaning of a procedure p in a program is a set of of n-tuples such that any model of the program is also a model of the relation denoted by p.

Yet another view is possible with the use of least fixed-point semantics. In this view, one associates a transformation T_P with a program P. This transformation maps a set of ground atoms to a set of ground atoms. The meaning of a program is then defined to be the least fixed-point of this transformation.

Van Emden, Kowalski, and Apt [38, 5] show that these three notions of semantics coincide for Horn clause logic programs (when the inference rule associated with operational semantics is complete). A more detailed account can be found in Lloyd's book [52].

2.3 Operational Semantics of Logic Programs

Operational semantics specifies, in an implementation independent way, not only what the results of a computation should be but also how the results of a computation are obtained. The operational semantics of a language is usually given in terms of a *transition system* which consists of configurations (or states) and transitions between configurations. *Natural semantics* focuses on how the overall results of a computation are obtained whereas *structured operational semantics* (SOS) specifies the meaning in terms of individual execution steps.

For Horn clause logic programming, a proof procedure known as SLD-resolution provides an operational semantics [52]. SLD-resolution may be seen as an SOS-style semantics [69] in that it relates a configuration consisting of a goal and a substitution to the next configuration via resolution.

2.3.1 Substitutions

The notion of binding a variable is captured by a mapping from variables to terms. A substitution is an infinite mapping (from the set of variables to the set of terms) which is the identity everywhere except at a finite number of points. For example, $\{X \mapsto f(a), Y \mapsto Z\}$ is the non-trivial portion of a substitution which maps variable X to $f(a)$ and variable Y to Z. It maps all other variables to themselves. We can therefore represent the non-trivial portion of a substitution θ : *Subst* = *Var* $\rightarrow$ *Term* by a finite set of bindings $\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ where the v_i are distinct variables, the t_i are terms, the v_i do not occur in any of the t_j.

A *renaming substitution* is a bijective substitution from variables to variables and which, by definition, has an inverse.

An element $v_i \mapsto t_i$ of a substitution is called a *binding*. The definition of substitution can be extended naturally from *Var*$\rightarrow$*Term* to other objects such as terms and literals. Hereafter, substitutions mean either the original definition or its extension to terms, atoms, etc.

The term $t\theta$ is obtained by applying substitution θ to a term t, *i.e.*, replacing every variable v_i of t with the corresponding t_i for which there is a binding $v_i \mapsto t_i \in \theta$.

We use *Vars*(T) to mean the set of variables in T, where T is a syntactic object. The domain of a substitution *dom*(θ) denotes the finite set of variables v_i such that $v_i\theta \neq v_i$. The range of a substitution *range*(θ) is the set of variables X such that there is a variable $v_i \in dom(\theta)$, $v_i\theta = t$ and $X \in Vars(t)$. Note that the *dom* and *range* are defined so that they refer only to the non-trivial portion of a substitution, and $Vars(\theta) = dom(\theta) \cup range(\theta)$.

Example 2.2

If $\theta = \{X \mapsto f(2, Z_1), Y \mapsto Z_2\}$, then $dom(\theta) = \{X, Y\}$, $range(\theta) = \{Z_1, Z_2\}$, and $Vars(\theta) = \{X, Y, Z_1, Z_2\}$. The restriction of the domain of a substitution θ to a set of variables V will be denoted as $\theta|_V$. □

Sequential Composition

The composition of substitutions θ_i and θ_j, *i.e.*, $\theta_i \circ \theta_j$, is defined as $\lambda x.(x\theta_i)\theta_j$. In other words, for all terms t, $t(\theta_i \circ \theta_j) = (t\theta_i)\theta_j$. Substitutions, as we have defined them, are idempotent, *i.e.*, $\theta \circ \theta = \theta$, since *dom*$(\theta)$ and *range*(θ) are disjoint. Applying an idempotent substitution one or more times to a term produces the same result.

Composition of idempotent substitutions does not necessarily yield an idempotent substitution. For example, consider $\theta_i \circ \theta_j$ where $\theta_i = \{X \mapsto f(Y)\}$ and $\theta_j = \{Y \mapsto f(X)\}$. However, whenever we compose two idempotent substitutions $\theta_i \circ \theta_j$, it will be the case that $dom(\theta_j) \subseteq range(\theta_i)$ and $range(\theta_j) \cap dom(\theta_i) = \emptyset$. In the above example, this condition is not satisfied by θ_i and θ_j. The composition of substitutions satisfying the above condition yields an idempotent substitution. In this dissertation, we consider idempotent substitutions only.

Equivalence Classes of Substitutions

Composition allows us to define a natural pre-ordering of substitutions: $\theta_1 \leq \theta_2$ iff $\exists \theta_3 \in$ *Subst* such that $\theta_2 = \theta_1 \circ \theta_3$. Intuitively speaking, θ_1 is less specific than θ_2.

Example 2.3

Let $\theta_1 = \{X_2 \mapsto g(Z)\}$ and $\theta_2 = \{X_2 \mapsto g(R)\}$. Then, $\theta_2 \leq \theta_1$ since θ_1 can be obtained from θ_2 by composing θ_2 with $\theta_3 = \{R \mapsto Z\}$. Likewise, $\theta_1 \leq \theta_2$ since θ_2 can be obtained from θ_1 by composing θ_1 with $\theta_4 = \{Z \mapsto R\}$. □

This notion allows us to define equivalence classes of substitutions. We say that two substitutions are equivalent if they can be obtained from each other through renaming substitutions. Define an equivalence relation $\sim$ on substitutions, induced by $\leq$ as: $\sigma \sim \theta$ *iff* $\theta \leq \sigma \wedge \sigma \leq \theta$. For example, the substitutions $\theta_1 = \{X_2 \mapsto g(Z)\}$ and $\theta_2 = \{X_2 \mapsto g(R)\}$ form an equivalence class $\{\theta_1, \theta_2, \ldots, \}$ and the substitutions $\theta_3 = \{X_2 \mapsto g(f(Z))\}$ and $\theta_4 = \{X_2 \mapsto g(f(S))\}$ also form an equivalence class $\{\theta_3, \theta_4, \ldots, \}$. The equivalence class of a substitution θ is denoted by $[\theta]$. The set of equivalence classes of substitutions is denoted as $Subst_{\sim}$. The relation $\leq$ induces a partial order on $Subst_{\sim}$, and the partial order is also denoted by $\leq$. In the above example, $[\theta_1] \leq [\theta_3]$.

2.4 Parallel Composition

Composition is asymmetric in its two arguments. When two substitutions are composed, the bindings in the two substitutions should represent equality constraints on variables, and these constraints should hold simultaneously. This idea of reconciling the information in two substitutions is inherently symmetric. However, composition as we have defined it is not symmetric, since $\theta \circ \sigma \neq \sigma \circ \theta$ in general. To emphasize this fact, this form of composition is known as sequential composition.

Consider the parallel execution of two body goals a_i and a_j given current substitution θ. In general, they may produce different bindings for the variable terms that may occur in both. The notion of ensuring compatible bindings is absent in sequential composition. Since unification of terms may be viewed as solving equations, a parallel composition operator that has many nice algebraic properties [68] may be defined as solving the simultaneous equations implied by the two substitutions. We now discuss this viewpoint, along with a parallel composition operator.

A set of equations $\{X_1 = t_1, \ldots, X_n = t_n\}$ is in *solved form* iff the X_i are distinct variables and they do not occur in the right-hand side of any equation. It is clear from the definition that an idempotent substitution $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ may be viewed as a set of equations in solved form and vice versa. The equation set of a substitution θ will be denoted $Eqn(\theta)$. The equivalence class of equations corresponding to the equivalence class of idempotent substitutions $[\theta]$ will be denoted by $[Eqn(\theta)]$.

A substitution θ is a *unifier* of terms t_1 and t_2 if $t_1\theta \equiv t_2\theta$, where $\equiv$ is syntactic identity. The set of most general unifiers (*mgu*) of two terms form an equivalence class under the relation $\sim$. For this reason, it is often referred to as *the* most general unifier. A substitution θ unifies a set of term equations $\{s_1 = t_1, \ldots, s_n = t_n\}$ iff $\{s_1\theta = t_1\theta, \ldots, s_n\theta = t_n\theta\}$. The equivalence class of most general unifiers of an equation set E is denoted by *unif(E)*. An equation set E is unifiable iff there exists a substitution θ that unifies E. Two sets of equations E_1, E_2 are equivalent iff *unif*$(E_1) =$ *unif*(E_2). The unification algorithm is defined next, followed by the parallel composition operation.

Unification of a set of equations is the process of continually rewriting the equations until they are in solved form [50]. Figure 2.1 shows a set of equivalence preserving rewrite rules that are used to perform the transformation.

To find the mgu of two terms (atoms) t_1 and t_2, we begin by forming the singleton set of equations $\{t_1 = t_2\}$. The unification algorithm transforms a set of equations into solved form if the set is solvable or halts with failure. Each step of the algorithm involves choosing

- $f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n)$. Replace the equation with the equations $t_1 = s_1,\ldots,t_n = s_n$.
- $f(t_1,\ldots,t_n) = g(s_1,\ldots,s_m)$. If $f \neq g$ or $m \neq n$, halt with failure.
- $x = x$. Delete the equation.
- $t = x$ where t is not a variable. Replace it with $x = t$.
- $x = t$ where x is variable, $x \not\equiv t$ and x has another occurrence in other equations. If x occurs in t, halt with failure. If not, replace x by t in all other equations.

Figure 2.1: Equivalence Preserving Transformation Rules

an equation to which one of the rules of Figure 2.1 apply and applying the rule. Each rule transforms a set of equations into an equivalent one. The algorithm is deterministic in the sense that for each equation, only one (or none) of the rules in Figure 2.1 is applicable. It is non-deterministic in the sense that the rules may be applied in any order. However, the most general unifiers of a set of equations form an equivalence class and hence the unification algorithm is confluent, *i.e.*, it produces the most general unifier which is unique modulo variable renaming.

Parallel composition may be defined in terms of solving the union of two sets of simultaneous equations, thus capturing the symmetric nature of reconciliation.

Recall that the set of equivalence classes of idempotent substitutions $Subst_{\sim}$ is a partially ordered set, with the partial order being $\leq$ and the equivalence class of a substitution θ is denoted by $[\theta]$. In the discussion following the definition, we will not differentiate between an idempotent substitution and its equivalence class.

The result of reconciling two idempotent substitutions θ and σ is defined to be the most general unifier of the union of σ and θ considered as sets of equations.

Definition 2.1 [Parallel Composition]
$\uparrow$: $Subst_{\sim} \times Subst_{\sim} \rightarrow Subst_{\sim} \cup \{fail\}$
$[\theta] \uparrow [\sigma] = mgu(Eqn(\theta) \cup Eqn(\sigma))$ if it exists, *fail* otherwise □

Notice that the definition is symmetric with respect to the substitutions being reconciled.

Example 2.4
Let $\theta = \{X \mapsto g(Y,a), Z \mapsto g(b,Y)\}$ and $\sigma = \{X \mapsto g(a,Q), Z \mapsto g(P,Y)\}$. Then

$$\begin{aligned} \theta \uparrow \sigma &= mgu(\{X = g(Y,a), Z = g(b,Y), X = g(a,Q), Z = g(P,Y)\}) \\ &= \{X \mapsto g(a,a), Z \mapsto g(b,a), Y \mapsto a, P \mapsto b, Q \mapsto a\} \end{aligned}$$

On the other hand, the sequential composition $\theta \circ \sigma$ is equal to $\{X \mapsto g(Y,a), Z \mapsto g(b,Y)\}$ where the effect of the substitution σ is lost. This is because applying

θ to a term t will replace all occurrences of X in t by $g(Y,a)$ and all occurrences of Z in t by $g(b,Y)$. Hence applying σ to $t\theta$ will have no effect. In other words, $dom\sigma \cap Var(t\theta) = \emptyset$. A similar problem occurs if σ is applied first. □

We will not go into the details, but the poset $(Subst_{\sim}, \leq)$ of equivalence classes of idempotent substitutions is a complete lattice (with a new top element *fail* added to it) and $\uparrow$ is the lub operation. $\uparrow$ is commutative, associative and idempotent. Hereafter, we use "substitution" to mean the set of equivalence classes of idempotent substitutions $Subst_{\sim}$.

2.4.1 SLD-Resolution

SLD-resolution stands for Linear resolution with a Selection function for Definite clauses. SLD-resolution solves a goal G, consisting of a sequence of atomic formulas $a_1, \ldots, a_n$, in the context of a program P and a current substitution θ (initially empty). An atom a_m is selected using a *computation rule*; then a *search rule* is used to find a clause $b \leftarrow b_1, \ldots, b_k$ in P such that b unifies with $a_m\theta$ using substitution θ_i (after renaming the variables of the clause). The body literals replace a_m in the goal and the process continues after composing the current substitution with θ_i, *i.e.*, the new goal is $a_1, \ldots, a_{m-1}, b_1, \ldots, b_k, a_{m+1}, \ldots, a_n$ and the new current substitution is $\theta \circ \theta_i$. This process repeats until the goal is empty, in which case the initial goal has been successfully solved, or there is an atom in the goal that does not unify with the head of any of the clauses. When the goal is empty, the current substitution (restricted to the variables in the initial goal) is known as an *answer substitution*.

Consider the following program P and the initial goal G.

```
1.  p(X,Z) ← q(X,Y), r(Y,Z).
2.  r(X,X).
3.  q(a,b).
← p(A,b).
```

Suppose that the computation rule always selects the leftmost atomic formula in a goal and the search rule selects clauses in order from 1 to 3. The goal clause defines an initial goal sequence `p(A,b)`. To solve `p(A,b)`, we unify it with the head of clause 1, using the substitution $\{A \mapsto X, Z \mapsto b\}$. The new goal is `q(X,Y), r(Y,Z)` and the current substitution θ_0 is $\{A \mapsto X, Z \mapsto b\}$. From the new goal, the computation rule selects the atom `q(X,Y)` and then the selection rule selects the only clause for q. The unifying substitution θ' is $\{X \mapsto a, Y \mapsto b\}$. The new current substitution $\theta_1 = \theta_0 \circ \theta'$ is $\{A \mapsto a, X \mapsto a, Y \mapsto b, Z \mapsto b\}$. Since the body of clause three is empty, the new goal is `r(Y,Z)`. The computation rule selects the only atom in the new goal; the selection rule selects the only clause `r(X,X)` which is renamed to `r(X',X')` and then unified with $\theta_1 r(Y,Z)$. The unification succeeds with $\theta'' = \{X' \mapsto b\}$ as the unifier. The new current substitution $\theta_2 = \theta_1 \circ \theta'' = \{A \mapsto a, X \mapsto a, Y \mapsto b, Z \mapsto b, X' \mapsto b\}$.

Definition 2.2

Let $\langle G, \theta \rangle$ be a state where G is a goal clause $\leftarrow a_1, \ldots, a_n$, and θ is the current substitution. Let C be a program clause $a \leftarrow b_1, \ldots, b_k$. The state $\langle G', \theta' \rangle$ is *derived* from the state $\langle G, \theta \rangle$ and C using mgu σ if the following conditions hold:

- a_m is an atom, called the *selected* atom, in G. $(1 \leq m \leq n)$
- σ is an mgu of $a_m\theta$ and a.
- G' is the goal $\leftarrow a_1, \ldots, a_{m-1}, b_1, \ldots, b_k, a_{m+1}, \ldots, a_n$ and $\theta' = \theta \circ \sigma$.

$G'\theta'$ is called the *resolvent* of $G\theta$ and C. □

Definition 2.3

Let P be a program and G be a goal and ε be the identity substitution. An *SLD-derivation* of $P \cup G$ consists of a finite or infinite sequence $S = \langle G, \varepsilon \rangle, S_1, \ldots,$ of states, a sequence $C_1, C_2 \ldots,$ of renamed versions of program clauses of P and a sequence $\sigma_1, \sigma_2, \ldots,$ of mgu's such that each S_{i+1} is derived from S_i and C_{i+1} using σ_{i+1}. □

Note that since state $S_{i+1} = \langle G_{i+1}, \theta_{i+1} \rangle$ is derived from state $S_i = \langle G_i, \theta_i \rangle$ using a substitution σ_{i+1}, $\theta_{i+1} = \theta_i \circ \sigma_{i+1}$. The clause C_i used in step i is a renamed version of the corresponding program clause that does not have any variables in common with the derivation so far. SLD-derivations may be finite or infinite. A finite derivation is successful if it ends in a state $\langle \Box, \theta \rangle$, where □ denotes the empty goal. A successful derivation is known as a *refutation*.

Definition 2.4

An *SLD-refutation* of $P \cup G$ is a finite SLD-derivation of $P \cup G$ which has the empty clause □ as the goal in the last state of the derivation. If G_n is □, then the refutation has length n. □

A derivation has finitely failed if it ends in a state with a non-empty goal clause whose selected atom does not unify with the head of any program clause.

Definition 2.5

A *computation rule* is a function that maps a goal clause to an atom in that goal, called the selected atom. □

A *search rule* selects a clause that will unify with the selected atom. A search rule is defined with reference to an SLD-tree shortly.

For example, a computation rule may always select the leftmost atom in a goal clause. Likewise, a search rule may select clauses in the order in which they appear in a program.

It is important to note that the choice of a computation rule does not affect the results. If $P \cup G$ is unsatisfiable, then for any computation rule R, there exists an SLD-refutation of $P \cup G$ via R. This is known as the *independence of the computation rule* and is due to Apt and Van Emden [5].

An SLD-tree is a graphical representation of all SLD-derivations of a program and goal via a given computation rule R. The computation rule determines the shape and size of the SLD-tree. A search rules merely explores the SLD-tree in a certain order. Since a computation rule determines the search space of a program (via the construction of an SLD-tree), it is more important than a search rule. However, in the presence of infinite branches, the search rule also becomes important. An unfair search rule such as a depth-first search rule (e.g. searching clauses in the order they appear in the program) makes the search incomplete, *i.e.,* does not always find all the solutions.

The construction of an SLD-tree starts with the initial state $\langle G, \varepsilon \rangle$ at the root. Given a node with state S suppose some atom a_m is the selected atom. Then each derived state S' of state S and clause C_i $(1 \leq i \leq n)$ will be a child node of the given node. Each path in the SLD-tree is an SLD-derivation.

Definition 2.6

Let P be a program and G a goal. An *SLD-tree* for $P \cup G$ is a tree satisfying the following conditions.

- The root node is the state $\langle G, \varepsilon \rangle$.
- Each node of the tree is a state with a possibly empty goal.
- Let $\langle \leftarrow a_1, \ldots, a_k, \theta \rangle$ be a node and $a_m, (1 \leq m \leq k)$ be the selected atom. Then for each input clause $a \leftarrow b_1, \ldots, b_n$ in the program such that a and $a_m\theta$ are unifiable with an mgu σ, the node has a child $\langle \leftarrow a_1, \ldots, a_{m-1}, b_1, \ldots, b_n, a_{m+1}, \ldots, a_k, \theta \circ \sigma \rangle$.
- Nodes with an empty goal in their states have no children.

□

Definition 2.7

A *search rule* is a strategy for searching SLD-trees to find success branches. An SLD-refutation procedure is specified by a computation rule together with a search rule. □

A program and a goal may have widely differing SLD-trees depending on the computation rule. Figures 2.2 and 2.3 show SLD-trees for the same program but using two different computation rules that select the leftmost and rightmost atoms respectively.

The SLD-tree in Figure 2.2 is built using a computation rule that selects the leftmost literal in a goal. The atom chosen by the computation rule in each goal node is underlined. The labels on the edges denote the number of the clause used in resolving the selected atom of the parent node. This tree shows three SLD-derivations, corresponding to the three paths from the root to the leaves. Two of them are successful and one is a failure derivation. The leaves of the successful derivations are labeled with the answer substitution restricted to the variables of the initial goal.

The same program and goal have a very different SLD-tree when the rightmost literal selection rule is used, as shown in Figure 2.3. For the same program, the computation rule that selects the rightmost literal in a goal creates an SLD-tree that has infinite branches. A depth-first search will enter into an infinite loop and miss the solutions to the right of the infinite path.

The number of success branches is the same for all SLD-trees of a goal and a program. A fair search rule will be able to find all the successful derivations even in the presence of infinite paths, whereas an unfair rule such as depth-first search will miss solutions to the right of an infinite path. A parallel search rule, likewise, will be able to find all successful derivations.

1. $p(X,Z) \leftarrow q(X,Y), p(Y,Z)$.
2. $p(X,X)$.
3. $q(a,b)$.

$\leftarrow p(A,b)$.

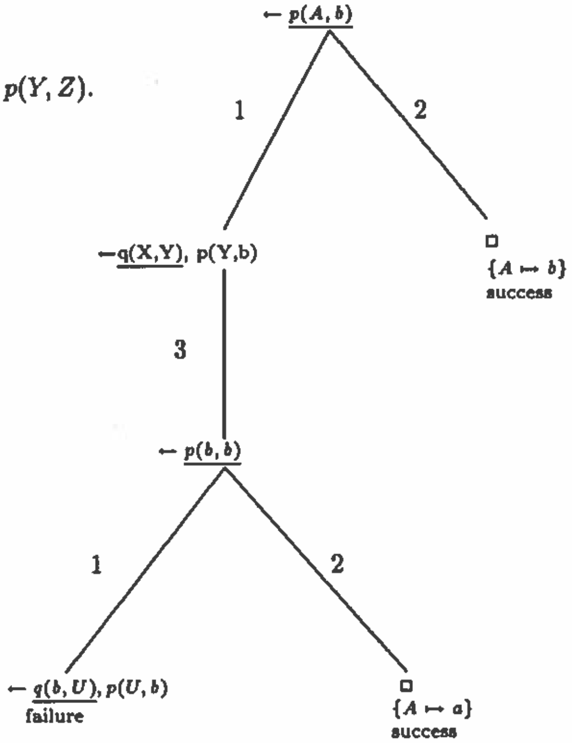

Figure 2.2: SLD-Tree with Leftmost Literal Selection

2.4.2 Use of Parallel Composition

In the operational semantics presented above, in a single step of resolution the current substitution θ is applied to the selected atom a before unifying the atom with the head of a clause. If unification succeeds, the substitution θ is composed with the unifier to give the new current substitution.

This step is reformulated using parallel composition. This simplifies the presentation and proof of correctness of our abstract interpretation. Instead of unifying the head h_j with the goal $a_m\theta$, we unify the head h_j with the goal a_m (*i.e.*, ignoring the current substitution θ) giving a substitution σ. We then obtain the new current substitution θ' as the parallel composition $\sigma \uparrow \theta$.

To prove that our reformulation is correct, we use the following proposition due to Palamidessi [68]. This proposition relates the parallel composition operation to sequential composition.

Proposition 2.1
For all idempotent substitutions σ_1 and σ_2,

$$\sigma_1 \uparrow \sigma_2 = \sigma_1 \circ mgu(Eqn(\sigma_2)\sigma_1)$$

where $Eqn(\sigma_2)\sigma_1 = \{x\sigma_1 = t\sigma_1 \mid x = t \in Eqn(\sigma_2)\}$ □

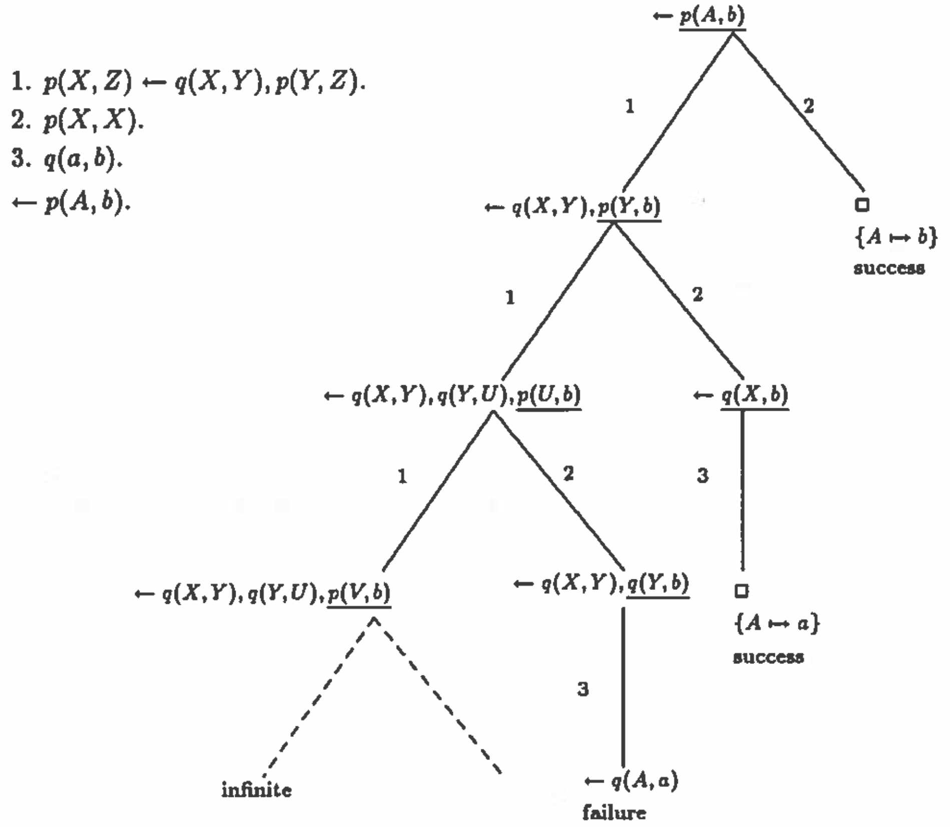

Figure 2.3: SLD-Tree with Rightmost Literal Selection

The proposition says that parallel composition $\sigma_1 \uparrow \sigma_2$ may be derived in terms of sequential composition as follows. The substitutions σ_1 and σ_2 are sets of equality constraints in solved form. Apply the substitution σ_1 to the equations of σ_2. The resulting set of equations $Eqn(\sigma_2)\sigma_1$ may no longer be in solved form, so solve them, *i.e.*, find the mgu of the new set of equations. The sequential composition of σ_1 with the mgu of $Eqn(\sigma_2)\sigma_1$ is equivalent to the parallel composition of σ_1 and σ_2.

Example 2.5

In Example 2.4 we had $\theta = \{X \mapsto g(Y,a), Z \mapsto g(b,Y)\}$ and $\sigma = \{X \mapsto g(a,Q), Z \mapsto g(P,Y)\}$. The parallel composition of θ and σ as per the original definition 2.1 is

$$\begin{aligned}
\theta \uparrow \sigma &= mgu(Eqn(\theta) \cup Eqn(\sigma)) \\
\theta \uparrow \sigma &= mgu(\{X = g(Y,a), Z = g(b,Y), X = g(a,Q), Z = g(P,Y)\}) \\
&= \{\{X \mapsto g(a,a), Z \mapsto g(b,a), Y \mapsto a, P \mapsto b, Q \mapsto a\}\}
\end{aligned}$$

Using Proposition 2.1, let us re-derive the parallel composition of σ and θ.

$$\begin{array}{rcl}\theta \uparrow \sigma & = & \theta \circ mgu(Eqn(\sigma)\theta) \\ & = & \theta \circ mgu(\{X = g(a,Q), Z = g(P,Y)\}\theta) \\ & = & \theta \circ mgu(\{X\theta = g(a,Q)\theta, Z\theta = g(P,Y)\theta\}) \\ & = & \theta \circ mgu(\{g(Y,a) = g(a,Q), g(b,Y) = g(P,Y)\}) \\ & = & \theta \circ \{Y \mapsto a, Q \mapsto a, P \mapsto b\} \\ & = & \{X \mapsto g(a,a), Z \mapsto g(b,a), Y \mapsto a, P \mapsto b, Q \mapsto a\}\end{array}$$

□

We now prove that our reformulation is correct.

Proposition 2.2

Let σ be the current substitution, H be the head of a renamed clause being unified with the goal G and $Eqn(\theta)$ be the solved form of $\{H = G\}$. Then, $\sigma \circ mgu(H, G\sigma) = \sigma \uparrow \theta$. □

Proof

$$\begin{array}{rcll}\sigma \circ mgu(H, G\sigma) & = & \sigma \circ mgu(H\sigma, G\sigma) & \text{due to renaming, } H\sigma = H \\ & = & \sigma \circ mgu(\{H\sigma = G\sigma\}) & \\ & = & \sigma \circ mgu(\{H = G\}\sigma) & \text{by def. } \{H = G\}\sigma = \{H\sigma = G\sigma\} \\ & = & \sigma \circ mgu(Eqn(\theta)\sigma) & Eqn(\theta) \text{ is the solved form of } H = G \\ & = & \sigma \uparrow \theta & \text{by proposition 1}\end{array}$$

□

The Need for Denotational Semantics

The operational semantics relates a goal state to another goal state derived using a single step of execution, namely SLD-resolution. This is similar to structured operational semantics (SOS) [66] in that the emphasis is on individual steps of execution. For the analysis we are interested in, we would like to focus on what is being computed, *i.e.*, given an initial goal and a substitution, what are the answer substitutions of the program? We can ask a similar question at the level of a clause.

A denotational semantics helps in defining the meaning of a program (a clause) as an input-output relationship, *i.e.*, as a function from substitutions to substitutions. The main advantage of a denotational definition is its compositionality. The meaning of a program will be defined in terms of the meanings of its clauses, and the meaning of a clause will be defined in terms of the meanings of its body goals.

In the rest of this chapter, we assume that the body goals are solved in left-to-right order and all the clauses with heads that match the current goal are tried in parallel. The parallel search rule, which corresponds to Or-parallelism, means that we are interested not in the sequence of solutions but in the set of solutions. Our eventual goal is to execute programs using independent AND-parallelism as well, in which goals are not necessarily executed from left to right.

Call, Success, Entry, Exit Substitutions

The current substitution θ is given different names depending on the state of computation. The names are: call, entry, exit and success substitutions. If a goal is about to be unified with the head of a clause, the current substitution θ is known as the *call substitution* of the goal, since a body goal is interpreted operationally as a procedure call. If a goal has just been successfully unified with the head of a clause, the parallel composition of the call substitution θ and the unifier σ, *i.e.*, $\theta \uparrow \sigma$ with its domain restricted to the clause variables is the *entry substitution* for the clause. This corresponds to parameter passing and establishing an environment (activation frame) for the body goals. Since the first goal of a clause is executed in this environment, an entry substitution is also the call substitution for the first literal in the body of the clause. If the last literal in the body has just been solved, the current substitution is known as the *exit substitution*, corresponding to procedure exit. The reconciliation of the exit substitution, the call substitution, and the head–goal unifier when restricted to the variables of the environment of the goal a gives us a *success substitution* of goal a with respect to clause i. The initial substitution for the variables in the goal is denoted as ε.

Variable Renaming

During resolution, the variables of a clause must be renamed apart from the variables used so far. In the denotational semantics about to be given, the renaming of variables is handled by providing an extra argument, called a renaming index, to all the semantic functions. We assume that the variables in a program and the initial goal have been renamed so that a variable does not occur in more than one clause and refer to the set of program variables as *PVar*. We also assume that the set of variables *Var* is partitioned into countably infinite number of subsets Var_s, one corresponding to each renaming index s. Since each clause has only a finite number of variables, it is enough if the size of each partition is greater than or equal to the maximum number of variables in any clause. The index s is used to obtain fresh variables from partition Var_s. One must ensure that each time a clause is used in unification, a different renaming index is used.

The following example illustrates the use of renaming index. The term "reconciliation" is used to denote parallel composition.

```
1.  concat([], L, L).
2.  concat([X|Xs], Y, [X|Zs]) :-
        concat(Xs, Y, Zs).
3.  concat([a,b,], [c,d], T).
```

Note that no variable occurs in more than one clause. The initial renaming index is 1. When concat([a,b],[c,d],T) is unified with clause 2, variables of clause 2 are renamed by variables from the partition Var_1, *i.e.*, clause 2 variables are subscripted by 1. When solving the first subgoal of clause 2, the renaming index is extended by 1 (the literal number of the first subgoal), and hence the current renaming index will be 1.1. The first subgoal concat([b],[c,d],Zs$_1$) will unify with the second clause, and the variables of clause two will be renamed by variables from partition $Var_{1.1}$, *i.e.*, subscripted by 1.1.

We use $Var_{\leq s}$ to mean the set of variables from all partitions r such that r is lexicographically less than a given s. In the concat example, $Var_{\leq 1.2}$ includes all variables subscripted by 1 and 1.1.

All semantic functions use a renaming index $s \in N^*$, a sequence of natural numbers. The initial sequence is denoted by 1 and $s.i$ extends the sequence s by i. The function *rename* (not formally defined) will take a syntactic object such as a clause, and a renaming index, and rename the variables. As mentioned before, renaming can be thought of as simply subscripting the variables with the renaming index.

2.5 Denotational Semantics of Logic Programs

The denotational semantics presented in this section is a variation of Jones and Sondergaard's exposition [47]. The semantic functions are slightly different and we use parallel composition instead of sequential composition. The conditions on domains are simpler.

Recall that a denotational definition of a programming language consists of three parts: the abstract syntax, semantic domains and valuation functions. We first summarize the abstract syntax of a Horn clause program, and then define the semantic domains and valuation functions.

The language has the following abstract syntax. The syntactic categories not defined below are *Pred*, *Func*, and *Var* symbols whose members are p, f, and v respectively. *Pred* is the set of predicates, *func* is the set of function symbols, and *Var* is the set of variables. These are assumed to be countably infinite and mutually disjoint.

A program P is a set of one or more clauses. A clause c_i has a head h_i and a body b_i which may be empty. An empty body is denoted by *nil*. A non-empty body b_i is a conjunction of one or more atoms. If p is a k-ary predicate and $t_1, \ldots, t_k$ are terms, then $p(t_1, \ldots, t_k)$ is an atom. The arity of a predicate may be zero. A term t is either a variable v or of the form $f(t_1, \ldots, t_q)$ where f is a function symbol of arity q, and $t_1, \ldots, t_q$ are terms.

$$\begin{array}{llcll}
P: & \mathit{Prog} & ::= & \{c_1, \ldots, c_n\} & (n \geq 1) \\
c: & \mathit{Clause} & ::= & h_i :- b_i. & (i > 0) \\
b: & \mathit{Body} & ::= & a_1, \ldots, a_p & (p \geq 0) \\
a, h: & \mathit{Atom} & ::= & p(t_1, \ldots, t_o) & (o \geq 0) \\
t: & \mathit{Term} & ::= & v \mid f(t_1, \ldots, t_q) & (q \geq 0)
\end{array}$$

The meaning of a logic program is taken to be the input-output relation computed by the program. We are interested in the substitutions computed by a program for the variables in the goal a_0 and the substitutions that prevailed at various points in the program during an SLD refutation of the goal. For simplicity, we assume that the initial goal consists of a single literal a_0. For a goal b with more than one literal in it, we may introduce a new clause whose body is b and whose head has a new predicate not appearing any where in the program. If b has n distinct variables, the new predicate will be n-ary and its arguments are these n variables. The new goal will be a literal that has the same n-ary predicate as the new clause head.

In the following discussion, references to renaming indices s or their type N^* will be omitted, although the semantic functions will continue use them.

Semantic Domains

The meaning of a program, clause, body, and an atom will be functions, denoted by d_p, d_c, d_b, and d_a respectively. Their corresponding types are D_p, D_c, D_b, and D_a.

Since we are interested in the input-output relationship, it is appropriate to define the denotation d_c of a clause as a function which maps a literal and a set of (call) substitutions to a set of (success) substitutions. Thus, the type of d_c is $D_c = N^* \rightarrow Atom \rightarrow \mathcal{P}(Subst) \rightarrow \mathcal{P}(Subst)$. Note that → associates to the right.

The first argument to the denotation of a clause is the renaming index whose type is N^*, a finite sequence of natural numbers. The second argument is the goal that is to be unified with the head of the clause. The third argument is the set of call substitutions for the goal. The denotation of a clause c maps these three arguments to the set of answer substitutions for the goal arising from clause c.

The denotation of an atom is similar to that of a clause except that we define it to be a mapping from a single (call) substitution to a set of (success) substitutions. Hence, the type of d_a is $D_a = N^* \rightarrow Atom \rightarrow Subst \rightarrow \mathcal{P}(Subst)$. The denotation d_b of a body will be a function that maps a set of (entry) substitutions to a set of (exit) substitutions and hence has type $D_b = N^* \rightarrow \mathcal{P}(Subst) \rightarrow \mathcal{P}(Subst)$.

The denotation d_p of a program is an n-tuple of functions such that the i^{th} component (written $d_p \downarrow i$) is the denotation of the i^{th} clause. Since the meaning function for a clause has type D_c, and since there are n clauses, the denotation d_p of a program has type $D_p = D_c \times \ldots \times D_c = D_c^n$.

We know the types of the meaning of an atom, a clause, a body and a program. Now we define the types of the semantic functions **P**,**C**, **B**, and **A**, corresponding to the syntactic categories program, clause, body, and atom. A semantic function **P** maps a program to its denotation d_p of type D_p. Likewise, the semantic function **C** maps a clause to its denotation d_c. The semantic functions for a body and an atom are similar.

Types of Semantic Functions

Since the denotation of a program has type D_p, the semantic function **P** which maps a program to its denotation has the type $Prog \rightarrow D_p$.

The meaning of a clause is d_c and its type is D_c. The semantic function **C** takes the denotation d_p of the program P and a clause c, and maps them to the clause c's denotation d_c; hence the type of **C** is $Clause \rightarrow D_p \rightarrow D_c$. Likewise, the type of **A** the semantic function for an atom is $Clause \rightarrow D_p \rightarrow D_a$, and the type of **B** the semantic function for a body is $Body \rightarrow D_p \rightarrow D_b$. To summarize, the standard semantics has the following domains and

$\theta :$	$Subst$	Substitutions
$\Phi :$	$\mathcal{P}(Subst)$	Sets of substitutions
$s :$	N^*	Renaming Indices
$d_c :$	$D_c = N^* \rightarrow Atom \rightarrow \mathcal{P}(Subst) \rightarrow \mathcal{P}(Subst)$	Clause Denotations
$d_a :$	$D_a = N^* \rightarrow Atom \rightarrow Subst \rightarrow \mathcal{P}(Subst)$	Atom Denotations
$d_p :$	$D_p = D_c^n$	Program Denotations
$d_b :$	$D_b = N^* \rightarrow \mathcal{P}(Subst) \rightarrow \mathcal{P}(Subst)$	Body denotations

the semantic functions have the following functionality.

$$\begin{aligned} \mathbf{P} &: Prog \rightarrow D_p \\ \mathbf{C} &: Clause \rightarrow D_p \rightarrow D_c \\ \mathbf{A} &: Clause \rightarrow D_p \rightarrow D_a \\ \mathbf{B} &: Body \rightarrow D_p \rightarrow D_b \end{aligned}$$

Let us now consider the definitions of the semantic functions.

Meaning of an Atom: Function **A** defines the meaning of an atom a with respect to a clause c and a call substitution θ. If goal a does not unify with head h of clause c, then the clause does not contribute to the meaning of the atom a. If a unifies with the head of c, then the meaning is simply the set of substitutions obtained by reconciling the call substitution with the unifier of the head and the goal and each of the exit substitutions of the clause corresponding to the entry substitution induced by the call substitution. This may be explained as follows.

The call substitution expresses some equality constraints among the variables of the goal. The unification of the head and the goal introduces additional constraints. Solving all the body goals introduces even more constraints in the form of an exit substitution. All these must hold simultaneously and hence the need for reconciliation.

The entry substitution induced by the call substitution has bindings for both the clause variables and the goal variables. The latter are irrelevant for finding the exit substitutions of a clause (corresponding to its entry substitution). Hence, we restrict the domain of the entry substitution to the clause variables; these are indexed by s. Function *restrict*, used for this purpose, is defined below.

$$\begin{aligned}&\mathit{restrict} : \mathit{Subst} \rightarrow \mathcal{P}(\mathit{Var}) \rightarrow \mathit{Subst}\\ &\mathit{restrict}(\theta, A) = \{v \mapsto t \mid v \mapsto t \in \theta \text{ and } v \in A\}\end{aligned}$$

The exit substitutions of a body b corresponding to an entry substitution are given by the denotation of b (to be discussed). After reconciling the call substitution, the unifying substitution, and the exit substitution, we restrict the domain of the reconciled substitution to the variables $Var_{\leq s}$ at the current renaming level s and below (*i.e.*, variables indexed by some r such that r is lexicographically less than or equal to s). This is because other variables are not observable and are irrelevant.

Meaning of a Clause: Now consider the semantic function **C**. As mentioned earlier, the denotation of a clause is a function that maps an atom and a set of (call) substitutions Φ to a set of (success) substitutions. This is simply the lub of sets of success substitutions corresponding to each call substitution $\theta \in \Phi$.

Meaning of a Body: Function **B** defines the denotation of the body of a clause. Given the denotation d_p of the program, **B** maps a set of (entry) substitutions to a set of (exit) substitutions. In the base case the body is empty, and hence the entry and exit substitutions are the same. In the inductive case, when the body has j literals, let the denotation d_p of the program map the first goal and the call substitutions to a set of success substitutions. Then, using the success substitutions of subgoal a_k as the call substitutions of subgoal a_{k+1} for k ranging from 1 to $j-1$, we find the success substitutions of the last body goal, which are the exit substitutions of the body corresponding to the entry substitutions.

Meaning of a Goal: The answer substitutions of a goal a_0 is just the lub of the sets of answer substitutions of a_0 from each clause in the program. Recall that the meaning of i^{th} clause of a program is given by $(\mathbf{P}[\![P]\!] \downarrow i)$ where $\downarrow i$ denotes the i^{th} component. Thus, the result of the program when run on a goal a_0 is given by $\mathbf{P}[\![P]\!]\, 1\, a_0 \{\varepsilon\} = \bigsqcup_{i=1}^{n} \{(\mathbf{P}[\![P]\!] \downarrow i)\, 1\, [\![a_0]\!]\, \{\varepsilon\}\}$, where 1 is the initial renaming index.

Example 2.6

Consider the following program along with a goal.

```
1. p(X,Z) ← q(X,Y), p(Y,Z).
2. p(X,X).
3. q(a,b).
← p(A,b).
```

The meaning of clause one maps the goal p(A,b) and the initial substitution (which is a singleton set of empty substitution) to a set of substitutions $\{\{A \mapsto a\}\}$, and the meaning of clause two maps the same goal and initial substitution to $\{\{A \mapsto b\}\}$. Therefore, the meaning of the program when applied to the goal p(A,b) and the initial substitution returns the union of $\{\{A \mapsto a\}\}$ and $\{\{A \mapsto b\}\}$ which is $\{\{A \mapsto a\}\{A \mapsto b\}\}$. □

The following definitions formalize our discussion so far.

$$\begin{array}{lll}
\mathbf{P}\llbracket c_1, \ldots, c_n \rrbracket & = & \mathit{fix}\ \lambda d_p.(\mathbf{C}\llbracket c_1 \rrbracket d_p, \ldots, \mathbf{C}\llbracket c_n \rrbracket d_p) \\
\mathbf{C}\llbracket c_i \rrbracket d_p & = & \lambda s. \lambda\llbracket a \rrbracket. \lambda\Phi. \bigsqcup_{\theta \in \Phi} \mathbf{A}\llbracket c_i \rrbracket\, d_p\, s\, \llbracket a \rrbracket\, \theta \\
\mathbf{A}\llbracket h_j :- b_j \rrbracket\, d_p\, s\, \llbracket a \rrbracket\, \theta & = & \\
 & & \begin{array}{ll}
 \mathbf{let} & (h, b) = \mathit{rename}(h_j, b_j, s) \\
 & \theta_{entry} = \mathit{restrict}(\mathit{mgu}(a, h) \uparrow \theta, \mathit{Var}_s) \\
 \mathbf{in} & \\
 & \mathbf{if}\ \theta_{entry} = \mathrm{fail}\ \mathbf{then}\ \emptyset \\
 & \mathbf{else}\ \{\mathit{restrict}(\theta \uparrow \mathit{mgu}(a, h) \uparrow \theta_{exit}, \mathit{Var}_{\leq s}) \mid \\
 & \quad \theta_{exit} \in \mathbf{B}\llbracket b \rrbracket\, d_p\, s\, \{\theta_{entry}\}\} \\
 \mathbf{end} &
 \end{array} \\
\mathbf{B}\llbracket \mathit{nil} \rrbracket\, d_p\, s\, \Phi & = & \Phi \\
\mathbf{B}\llbracket a_i, \ldots, a_j \rrbracket\, d_p\, s\, \Phi & = & \mathbf{B}\llbracket a_{i+1}, \ldots, a_j \rrbracket\, d_p\, s\, \Phi' \text{ where } \Phi' = d_p\, s.i\, \llbracket a_i \rrbracket \Phi
\end{array}$$

2.6 Abstract Interpretation

Abstract interpretation is a semantics-based method for deriving properties of programs. Abstract interpretation can be thought of as performing computations in "non-standard" domains. The domains and the corresponding computations are not arbitrary but bear some relationship to the domains and computations we are interested in. Typically, the non-standard domain is a simplification, or abstraction, of a standard domain.

As a simple example, suppose we want to know the sign of the result of simplifying the expression 984 × −32. We could find the answer by computing in the standard domains: multiply 984 by −32 to get −31488 and notice that the sign is negative. An easier method is to compute the sign of the product in the non-standard domain of signs. The rule of signs tells us that the product of a positive number and a negative number will be negative. We can map the standard domain of computation, integers, onto the set D = {zero, pos, neg} and replace the multiplication operation on the integers with another operation (the rule of signs) $f : \mathrm{D} \times \mathrm{D} \rightarrow \mathrm{D}$. The computation corresponding to 984×-32 is $+ \times -$, and it is apparent that the result will be negative.

It would be nice if an abstract interpreter could be designed that would take as input a program P and decide if P has a certain property or not. Unfortunately, Rice's theorem [25] implies that most of the interesting properties of programs of Turing-equivalent programming languages are undecidable. Rice's theorem says that if f and g are two partial recursive functions in a collection F such that f has property p and g does not, then membership in F is not decidable. Examples of these properties include equivalence of programs, the property of being a recursive function, a primitive recursive function, or a partial recursive function. In the context of logic programs, the undecidable properties include groundness and sharing of variables, determinacy, and others.

We must ensure that all terminating analyses of programs safely approximate the derivation of properties.

Most data flow analyses derive properties that hold in all possible executions of programs. Since we cannot know exactly all executions, the analyses must reason about the set of executions which covers all possible executions. This will include some executions that are not possible. In terms of logic programs, the analysis may derive a superset of correct answer substitutions for a given goal and a program. Thus, a larger set of substitutions A approximates a smaller set B which is a subset of A. If a property holds definitely for A, it holds definitely for B as well.

Suppose we are interested in finding the variables of a goal/clause that will be bound to ground terms during all possible executions starting in an initial goal. Call a variable a V "ground variable" in a substitution θ, if the variable V is bound to a ground term in θ. If a goal variable is ground in all the substitutions in A, it is ground in any subset B of A as well.

The undecidability of groundness causes imprecision in the following manner. A variable may be ground in all the substitutions in B but not in A. Thus a groundness analyzer has to conclude a smaller set of variables will be ground during all possible executions starting with some goal.

The semantics-based approach to program analysis is to define a *core semantics* for a language that leaves some domains and semantic functions unspecified. An interpretation I of the core semantics supplies the missing domains and semantic functions. The word *interpretation* is used to emphasize that the undefined function symbols and domains are *interpreted* to suit our purpose (subject to some conditions). A collecting semantics is then defined to associate with a program point the set of states or environments that may prevail at that point during all executions starting at an initial state. The collecting semantics is thus used as a device to record information at interior points in a program, instead of concentrating only at the input-output behavior.

An interpretation I of the core semantics is an *abstract interpretation* if the domains and semantic functions of I are abstractions of their counterparts in the collecting semantics and if I safely models the collecting semantics. We now discuss the *Galois insertion* approach to modeling a standard semantics by an abstract semantics. The discussion in the following section is based on work by Cousot and Cousot [22] and Debray [30].

2.6.1 The Galois Insertion Approach

In the Galois insertion approach to abstract interpretation, the domain of standard semantics is replaced by a domain of descriptions or properties. Computations are performed symbolically over the description domain. The description domain is generally known as the abstract domain. Naturally, the standard domain operations must have their abstract counterparts. We denote the abstract domain by *Abs* and the standard domain of computation as *Conc*.

Properties of Abstract Domains

In the case of standard semantics, the domains are defined to be pointed cpos, and functions over these domains must be continuous in order to be computable. Although domain theory was originally developed with complete lattices as domains, later developments showed that pointed cpos are sufficient. This is because the successive approximations of a recursively defined function form a chain (an ω-chain, in fact) and the meaning of such a function can be shown to be the lub of the chain. Moreover, cpos can be embedded in complete lattices. However, when computing with "descriptions," we sometimes need to take the lub of arbitrary subsets of the abstract domain and not just chains.

For example, when we define a core semantics later in this chapter, we take the lub of arbitrary subsets of the domain *Asub*. In the case of standard semantics, the lub operation turns out to be set union and hence well-defined. But in the case of an abstract interpretation, for the expression to be well-defined, lub of arbitrary subsets must exist. Hence, the abstract domain must at least be a join semi-lattice, *i.e.*, lubs of arbitrary subsets of *Abs* exist in *Abs*. An abstract domain usually has a least element. A join semi-lattice together with a least element is a complete lattice, *i.e.*, glbs of arbitrary subsets exist. Thus, an abstract domain must be a complete lattice.

Since an abstract domain is a domain of descriptions, we expect each abstract domain element to be well-defined, *i.e.*, it must describe some set of concrete domain elements. Thus there should be a function, known as the concretization function $\gamma : Abs \rightarrow Conc$. Since it is redundant to have two abstract domain elements describing the same set of *Conc* elements, γ must be one-to-one.

We would expect the abstract and concrete domains to be structurally similar, *i.e.*, if $a \sqsubseteq_{Conc} b$ implies that b has more information than a, then the ordering on *Abs* should also have the same meaning. Further, the concretization function must preserve the ordering, *i.e.*, must be monotone.

It is useful (but not essential) if each concrete domain element has a unique and most-precise description in the abstract domain. If each concrete domain element has a unique best description in the abstract domain, one can define an *abstraction* function $\alpha : Conc \rightarrow Abs$. Like the concretization function, the abstraction function is also required to be monotone. Elements of the abstract domain which do not correspond to any concrete domain element may be deleted safely. Hence α must be an onto function.

One usually loses some information abstracting an element x of the concrete domain. However, concretizing it back should safely approximate x from above. If an element x is above another element y in a concrete domain, then x approximates y *in the context of* abstract interpretation. Hence, the following condition should be met.

$$\forall x \in Conc.\ x \sqsubseteq_{Conc} \gamma(\alpha(x)) \qquad (1)$$

Abstracting a concrete domain element entails loss of information. Ideally, the concretization of an abstract element must not lose any information, *i.e.*,

$$\forall x \in Abs.\ x = \alpha(\gamma(x)) \qquad (2)$$

If the abstraction and concretization functions satisfy these conditions, they are said to be *adjoint*.

For each concrete operation $f : Conc \rightarrow Conc$, there should be a corresponding abstract operation $\mathbf{f} : Abs \rightarrow Abs$ that simulates the effect of f. More formally,

$$f(x) \sqsubseteq \gamma(\mathbf{f}(\alpha(x))) \quad \forall x \in Conc$$

In other words, the concretization of the result of a simulated operation over an abstract object must safely approximate the result of the corresponding concrete operation over the corresponding concrete object. This condition is known as the *local correctness condition*. If all concrete operations are simulated locally correctly by the corresponding abstract operators, then we can show by induction that correctness is maintained for a finite number of steps. However, the meanings of non-trivial programs are given by least fixed-points which are reachable in countably infinite number of steps. In such cases, a local correctness condition cannot be extrapolated to global correctness by finite induction. One must resort to transfinite induction or fixed-point induction [72], which implies one must find inclusive or admissible predicates to be used in induction. Cousot and Cousot's [21] well-known theorem established that in the Galois insertion approach, local correctness guarantees global correctness.

The following is a restatement (due to Debray [30]) of Cousots's theorem where *lfp(f)* and *gfp(f)* denote the least fixed-point and the greatest fixed-points of *f*.

Theorem 2.1

> [Cousot and Cousot 1977] Consider an abstract interpretation with concrete domain $\langle Conc, \sqsubseteq \rangle$ and abstract domain $\langle Abs, \leq \rangle$ that are complete lattices, with $\alpha : Conc \rightarrow Abs$ and $\gamma : Abs \rightarrow Conc$ monotone and adjoint. Then, if the abstract operator $\mathbf{f} : Abs \rightarrow Abs$ is locally consistent with the concrete operator $f : Conc \rightarrow Conc$, *i.e.*, $f(x) \sqsubseteq \gamma(\mathbf{f}(\alpha(x)))$ for all $x \in Conc$, then $lfp(f) \sqsubseteq \gamma(lfp(\mathbf{f}))$ and $gfp(f) \sqsubseteq \gamma(gfp(\mathbf{f}))$. □

Termination of the least fixed-point computation in an abstract interpretation can be ensured by having monotone abstract domain operations and abstract domains with no infinite ascending chains.

In the following sections, we develop an abstract interpretation framework for logic programs based on the Galois insertion approach. We develop a core semantics for the language defined in the previous section, show that the standard semantics of Section 2.5 may be seen as an interpretation of the core semantics, and establish correctness conditions. The abstract interpretation framework to be presented is derived from Jones and Sondergaard's abstract interpretation framework [47].

2.7 Abstract Interpretation of Logic Programs

2.7.1 Core Semantics

We refer to the semantics of Section 2.5 as the standard semantics. We generalize the standard semantics and derive a core semantics. Core semantics has domains and semantic functions that are common to all interpretations. Hence some domains and functions are not defined in core semantics. We will show that by defining the missing domains and semantic functions in a certain way, we get an interpretation identical to the standard semantics. We call this interpretation the standard interpretation. Furthermore, by defining the missing domains and semantic functions in a way that abstracts some property of the standard domains and semantic functions, we get an abstraction of the standard interpretation. In short, *core semantics + standard interpretation = standard semantics*, and *core semantics + abstract interpretation = abstract semantics.* The correctness of the interpretations is ensured by using a Galois insertion and by establishing local correctness conditions.

In core semantics, we replace the domain *Subst* of standard semantics with *Subst′* to model substitutions. The domain *Subst′* is left unspecified in core semantics and must be supplied by an interpretation. The domain *Asub* in core semantics performs the role of the domain $\mathcal{P}(Subst)$ in standard semantics. We leave *Asub* unspecified in core semantics and expect an interpretation to supply it. The domains of atom, body, clause and program denotations are changed accordingly.

To summarize, the domains and the types of semantics functions in core semantics are as follows.

Domains

θ :	$Subst'$	a concrete or an abstract substitution
Φ :	$Asub$	set of substitutions or its abstraction
s :	N^*	Renaming Indices
d_c :	$D_c = N^* \rightarrow Atom \rightarrow Asub \rightarrow Asub$	Clause Denotations
d_a :	$D_a = N^* \rightarrow Atom \rightarrow Subst' \rightarrow Asub$	Atom Denotations
d_p :	$D_p = D_c^n$	Program Denotations
d_b :	$D_b = N^* \rightarrow Asub \rightarrow \mathcal{P}(Subst)$	

Valuation Functions

$$\begin{array}{ll}
\mathbf{P}: & Prog \rightarrow D_p \\
\mathbf{C}: & Clause \rightarrow D_p \rightarrow D_c \\
\mathbf{A}: & Clause \rightarrow D_p \rightarrow D_a \\
\mathbf{B}: & Body \rightarrow D_p \rightarrow D_b
\end{array}$$

The valuation function for a body is the same as defined in standard semantics and is given below for the sake of completeness.

$$\begin{array}{lcl}
\mathbf{B}[\![nil]\!]\, d_p\, s\, \Phi & = & \Phi \\
\mathbf{B}[\![a_i, \ldots, a_j]\!]\, d_p\, s\, \Phi & = & \mathbf{B}[\![a_{i+1}, \ldots, a_j]\!]\, d_p\, s\, \Phi' \text{ where } \Phi' = d_p\, s\, [\![a_i]\!]\Phi
\end{array}$$

The semantic function **C** changes slightly in core semantics. We replaced $\mathcal{P}(Subst)$ with *Asub* in core semantics. The structure of *Asub* is left unspecified, to be filled in by an interpretation. Hence, instead of using set membership as in standard semantics, we use $\sqsubseteq_{Asub}$ in the definition of **C**.

$$\mathbf{C}[\![c_i]\!]d_p \quad = \quad \lambda\, s.\, \lambda[\![a]\!].\, \lambda\Phi.\, \bigsqcup_{\theta \sqsubseteq \Phi} \mathbf{A}[\![c_i]\!]\, d_p\, s\, [\![a]\!]\, \theta$$

A set of substitutions is usually abstracted by a single abstract substitution and hence *Asub* and *Subst′* may be defined to be the same in an interpretation. The above definition may then be simplified to the following.

$$\mathbf{C}[\![c_i]\!]d_p \quad = \quad \lambda\, s.\, \lambda[\![a]\!].\, \lambda\Phi.\, \mathbf{A}[\![c_i]\!]\, d_p\, s\, [\![a]\!]\, \Phi$$

The valuation function **A**, which defines the meaning of an atom with respect to a particular clause in the program, does change in core semantics. We abstract away (i) head-goal unification and (ii) the parallel composition of the call substitution with the head–goal unifier and an exit substitution of the clause.

The definition of **A** in standard semantics is reproduced below, followed by the corresponding definition in core semantics to show the differences. In the standard semantics, the clause was renamed and the entry substitution is computed as the parallel composition of the head–goal unifier and the call substitution. In core semantics, we delegate the computation of entry substitution to a function *entry_sub*, to be supplied by an interpretation. Likewise, in standard semantics the exit substitution of clause was reconciled with the head–goal unifier and the call substitution using parallel composition. In core semantics, we abstract this to a function *success_sub*, to be provided by an interpretation.

(standard semantics)

$$\begin{array}{lll}
\mathbf{A}[\![h_j \leftarrow b_j]\!]\, d_p\, s\, [\![a]\!]\, \theta \ = & & \\
& \textbf{let} & (h, b) = rename(h_j, b_j, s) \\
& & \theta_{entry} = restrict(mgu(a, h) \uparrow \theta, \mathrm{Var}_s) \\
& \textbf{in} & \\
& & \text{if } \theta_{entry} = \text{fail} \ \textbf{then} \ \emptyset \\
& & \text{else } \{restrict(\theta \uparrow mgu(a, h) \uparrow \theta_{exit}, \mathrm{Var}_{\leq s}) \mid \\
& & \qquad \theta_{exit} \in \mathbf{B}[\![b]\!]\, d_p\, s\, \{\theta_{entry}\}\} \\
& \textbf{end} &
\end{array}$$

(core semantics)

$$\begin{array}{lll}
\mathbf{A}[\![h_j \leftarrow b_j]\!]\, d_p\, s\, [\![a]\!]\, \theta \ = & & \\
& \textbf{let} & \theta_{entry} = restrict(entry\_sub(a, h_j \leftarrow b_j, \theta, s), \mathrm{Var}_s) \\
& \textbf{in} & \\
& & \text{if } \theta_{entry} = \text{fail} \ \textbf{then} \ Asub_{\perp} \\
& & \text{else } \{restrict(success\_sub(\theta, \theta_{exit}, a, h), \mathrm{Var}_{\leq s}) | \\
& & \qquad \theta_{exit} \in \mathbf{B}[\![b]\!]\, d_p\, s\, \{\theta_{entry}\}\} \\
& \textbf{end} &
\end{array}$$

Interpretations

An interpretation supplies the domains *Subst'* and *Asub* and the semantic functions *entry_sub* and *success_sub* which are required to be continuous in their arguments from the domain *Subst'*. It is not a coincidence that the valuation functions of core semantics are almost the same as those of the standard semantics. We define a standard interpretation as follows:

$$\begin{array}{rcll}
Subst' & = & Subst & \\
Asub & = & \mathcal{P}(Subst) & \\
entry\_sub(a, h \leftarrow b, \theta, s) & = & \textbf{let} & (h', b') = rename(h, b, s) \\
 & & \textbf{in} & mgu(a, h') \uparrow \theta \\
 & & \textbf{end} & \\
success\_sub(\theta, \theta_{exit}, a, h) & = & \theta \uparrow mgu(a, h) \uparrow \theta_{exit} &
\end{array}$$

Note that the standard interpretation coupled with the core semantics is identical to the standard semantics.

The core semantics given above may be enhanced to preserve the association between program points and sets of substitutions that may prevail at those points during execution. This can be achieved by changing the result types of **P**, **C**, **B**, and **A** from *Asub* to *Asub* × *Info* and making other necessary modifications. Alternatively, the type of *Asub* may be modified to include information at all program points. *Info* will be an $n \times k$ array where n is the number of clauses and k is the maximum number of literals in any clause plus one. *Info* stores information about the abstract substitutions encountered at program point i, j, where i is the clause number and j is the literal number in the i^{th} clause. When $j = 0$, $Info_{i,j}$ stores all entry substitutions of clause i and for $i > 0$, $Info_{i,j}$ has the success substitutions of literal j of clause i.

2.8 Summary

This chapter introduced Horn clause logic programming and described an operational semantics based on SLD-resolution. The effect of choosing a different computation rule determines the size and shape of the SLD-tree and hence selection of a good computation rule is important. Next, a denotational semantics of logic programs was given. The denotational definition serves two purposes. One is to focus on what is computed and on the properties of computations. The second is to have a compositional definition, so that static analyses can also be defined compositionally.

We then presented an abstract interpretation framework and discussed some of its properties. The denotational semantics of logic programs is generalized to a core semantics which leaves some domains and functions unspecified. When coupled with interpretations that specify the missing domains and functions, one obtains standard or abstract interpretations.

Chapter 3

An Abstract Interpretation Scheme for Analyzing Groundness, Freeness, and Sharing

This chapter presents our abstract interpretation scheme for deriving groundness, freeness, and sharing of variables in logic programs. Many researchers have proposed sharing analyses that treat sharing as a transitive relation mainly to realize worst-case polynomial time complexity. We claim that giving up precision *a priori* in view of worst-case time complexity is like throwing out the baby with the bath water.

Our proposed sharing analysis has an exponential worst-case time complexity (exponential in the number of variables in a clause). However, for most practical programs, the analyzer works in polynomial time. An adaptive *widening* operation used in conjunction with the abstract domain operations trades precision for time selectively and automatically. Widening may be thought of as skipping fixed-point iterations by making worst-case assumptions. Treating sharing as transitive *a priori* gives up precision for all procedures in all programs, whereas our proposal automatically and selectively loses precision only when certain parameters are met or exceeded during analysis. The widening operation is defined in the next chapter.

The loss of precision due to widening may be local and confined to a procedure or may have its effects propagated to the whole of the program. Thus at its worst, our method is equivalent to treating sharing as transitive. Furthermore, the execution time for our analyzer has worst-case polynomial time complexity for a class of programs that have "stable call substitutions" property. This is discussed in the next chapter.

We end the chapter with a proof of the local consistency of our analysis; the global consistency follows from the Galois insertion approach.

3.1 Interpretation: Sharing, Freeness and Linearity

A set of goals in the body of a clause may be executed in parallel without interfering with each other's computations, if these goals do not share variables. The groundness and sharing properties of terms help us determine when two goals may share and when they do not. Even when two goals share a common variable, it may be possible to execute them in parallel if we know that one of the goals will never bind that variable to a non-variable term, *i.e.,* it will leave the variable free.

Non-transitivity of Sharing

Analysis of sharing of variables is complicated by the fact that the sharing relation is not transitive. Consider the unification goal `X = f(Y, Z)` and assume that X, Y, and Z are unbound before the unification. After the unification, `X` and `Y` share and so do `X` and `Z`. However, this unification does not cause `Y` and `Z` to share and thus sharing is not transitive.

On the other hand, if this unification takes place in an environment in which X is bound to, say, f(Q, Q), then the variables Y and Z will also share after the unification.

Groundness analysis depends on the precision of the sharing analysis. Consider the conjunction X = Y, p(Y). If solving the goal p(Y) leads to binding Y to a ground term, then X also becomes ground. Most of the earlier work [44, 88, 29, 57] on abstract interpretation of logic programs either ignored sharing or considered sharing to be transitive. Both these treatments of sharing lead to imprecise (pessimistic) analyses.

One of the reasons advocated for treating sharing as transitive is to realize computationally efficient analyses. Debray shows that sharing analyses with worst-case polynomial time complexity can be derived only when sharing is treated as transitive [26] and some other conditions are satisfied. Treating sharing as transitive gives up precision apparently for efficiency. That is, the loss of precision is immediate but the computational advantage appears to be realizable only in the worst-case. We elaborate on this point in the next chapter when we discuss the time complexity of our analysis.

In the next section, we start with a sharing domain proposed by Jacobs and Langen [44]. This domain, although good at expressing variable sharing precisely, does not have enough information about sharing subterms. This results in treating sharing as transitive. Although sharing is treated as transitive, resulting in conservative sharing analysis, the efficiency usually claimed for transitive sharing analyses is absent. That is, abstract unification is still exponential in the number of clause variables and the time complexity of sharing analysis is still exponential in the size of the program. A claimed advantage of this domain is that the abstract meaning of a program can be computed bottom-up as the limit of Kleene's iteration sequence, independent of the entry point information. Once this information is computed, the abstract meaning of any goal or entry-point can be computed without further fixed-point computations. To our knowledge, this domain has not been implemented as proposed. Another extension of this domain proposed by Muthukumar and Hermenegildo [61] partly addresses the precision issue. We discuss and compare their work to ours at the end of this chapter.

Our proposed abstract domain is a cartesian product of three domains that express information about sharing, freeness and linearity of terms. Before defining the abstract domains and the new abstract domain operations, we discuss briefly termination and variable renaming in the abstract interpretation.

Termination in the Abstract Interpretation

In an abstract interpretation, we are interested in terminating analyses, *i.e.*, there should be no infinite ascending chains in the abstract domain. For example, a flat, infinite domain such as N has no infinite ascending chains. However, this does not mean that an abstract interpretation will always terminate. One must ensure that the abstract interpreter does not explore the whole space; if it did, the analysis will not terminate.

In our case, we merge information about all renamed versions of a clause and all information is expressed in terms of the program variables, *i.e.*, those that textually appear in the program. Recall that recursive clauses and renaming of variables together introduce infinite number of variables and thus the domain of substitutions will be infinite. In program analyses, we are interested in statements such as "any time a goal unifies with the

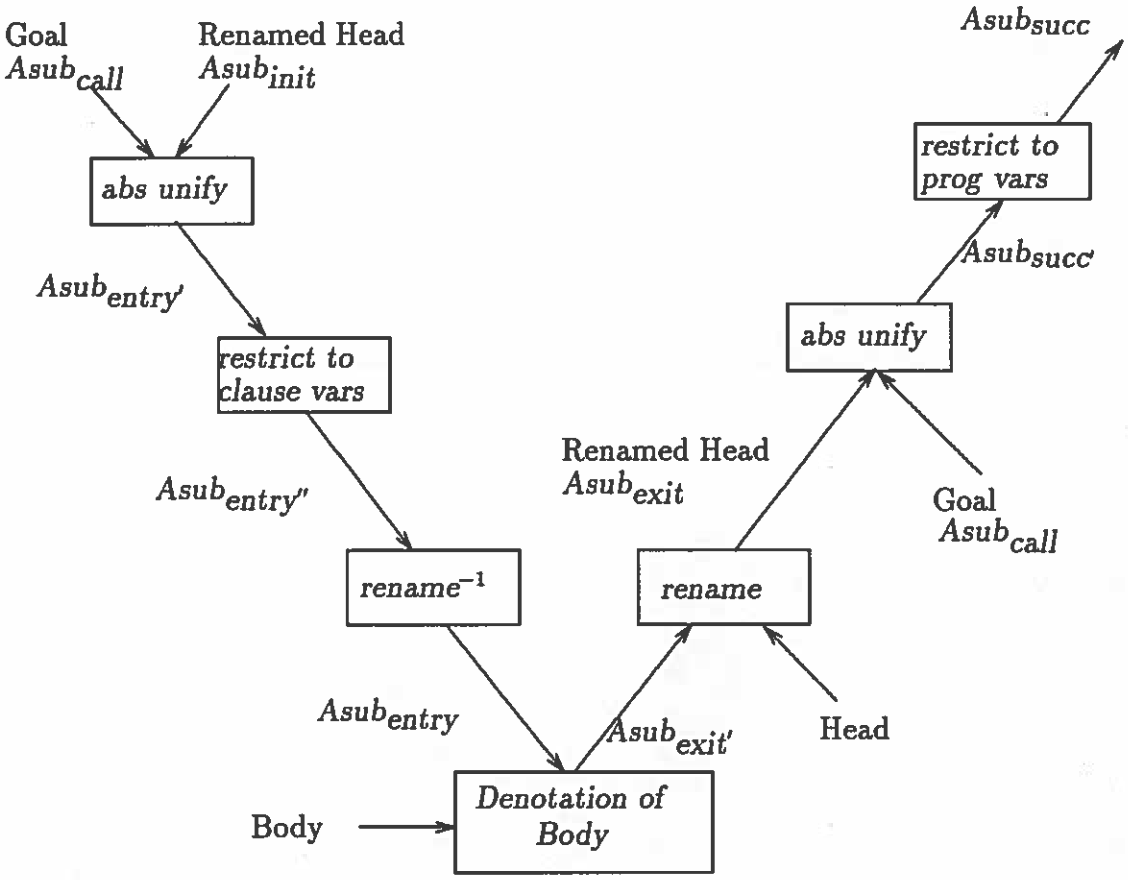

Figure 3.1: Computation of Entry, Exit, and Success Substitutions

head of clause i, the clause variables will have this property." This is mainly because procedures are usually compiled into re-entrant code which operates uniformly in all invocations. This claim holds good even when slightly different, specialized versions of a procedure are compiled because there are only finitely many such versions and each version must handle potentially an infinite number of invocations. Thus we must merge information about all renamed versions of a clause. Since we assume that all clause variables have been renamed so that a variable does not occur in more than one clause, expressing properties in terms of program variables accounts for merging of renamed versions of a variable.

Renaming and Overview of Clause Entry and Exit

Since we merge information about renamed versions anyway, it is better not to create renamed versions in the first place. Therefore, an abstract substitution will always be in terms of the variables of one clause, except during clause entry and exit. During clause entry and exit, information propagates from the goal (calling) environment to the clause (callee) environment and back. Thus, name clashes must be avoided at clause entry and exit. This is accomplished as follows.

During clause entry, the clause variables are consistently renamed away from the goal and the abstract call substitution. After head–goal unification, the abstract entry substitution is restricted to the renamed clause variables first and then an inverse of the renaming substitution is applied to the entry substitution. This recovers the entry substitution in terms of the clause variables which is then used to find the exit substitutions. At clause exit time, we need to pass information from the clause environment back to the goal environment. Once again, we rename the clause variables and the exit substitution consistently away from the goal and its call substitution, compute the success substitution and restrict the result to the variables in the goal environment. Because of this simple renaming scheme, the renaming index is ignored in our abstract interpretation. Instead, we simply subscript clause variables by, say 0, during clause entry and exit to make them different from goal variables.

We do not differentiate among different renamed versions of program variables. Thus the sharing, groundness, and linearity information is maintained only for those variables which appear in a program. The number of program variables is finite, and hence our abstract domain will be finite. Since there are no infinite ascending chains in the domain, fixed-point computations will terminate.

The computation of an entry, exit and a success substitution corresponding to a call substitution of a goal is summarized in Figure 3.1.

3.2 Abstract Domain *Asub*

Our abstract domain *Asub* is a triple $\langle \mathit{Free}, \mathit{Repeat}, \mathit{Sharing} \rangle$. Each component of *Asub* represents a particular property of a set of concrete substitutions Θ:

- *Free* is a set of variables that are *definitely* free (not bound to non-variable terms) in all $\theta \in \Theta$.

- *Repeat* identifies the set of variables that may be bound to non-linear terms, *i.e.*, terms that *may* have the same variable subterm occurring more than once.

- *Sharing* is the same as Jacobs and Langen's domain of the same name. It is a set $\{S_0, \ldots, S_n\}$ where each S_i is a set of variables which *may* be unified to terms that have variables in common by some substitution $\theta \in \Theta$.

Information about which variables may be bound to terms with repeated variable subterms allows us to derive a precise abstract unification algorithm which does not treat sharing as a transitive relation.

We now define the components of the abstract domain *Asub*. Definitions 3.1 and 3.2 are due to Jacobs and Langen [44].

Elements of the abstract domain *Sharing* will contain sets of variables that may have shared subterms in the concrete interpretation:

Definition 3.1 [Abstract Domain *Sharing*]

$\mathit{Sharing}$: $\mathcal{P}(\mathcal{P}(\mathit{Var}))$

$\sqsubseteq_{\mathit{Sharing}} = \subseteq$

$X_1 \sqcup X_2 = X_1 \cup X_2$, for $X_1, X_2 \in \mathit{Sharing}$ □

The partial ordering reflects the fact that *Sharing* is intended to express possible, as opposed to definite, sharing among the variables in any execution.

A substitution θ may have a binding $v \mapsto t$ where t contains a variable u. In this case we say u *occurs through* v. The function $occ(\theta, u)$ is the set of all variables through which u can occur given substitution θ:

Definition 3.2

$occ : Subst \times Var \rightarrow \mathcal{P}(Var)$

$occ(\theta, u) = \{v \in dom(\theta) \mid u \in Vars(v\theta)\}$ □

Note that $occ(\theta, u)$ is a set of variables that have a common variable subterm, namely u.

We can use occ to define a function $shared(\theta)$ that gives us all possible sets of variables that may have shared subterms in substitution θ. A set of variables $\{x_1, \ldots, x_n\}$ is in $shared(\theta)$ when $x_1, \ldots, x_n$ are bound respectively to terms $t_1, \ldots, t_n$ in θ and there is at least one common variable that occurs in the terms $t_1, \ldots, t_n$:

Definition 3.3

$shared : Subst \rightarrow Sharing$

$shared(\theta) = \{occ(\theta, u) \mid u \in Var\}$ □

Example 3.1

Let $\theta = \{W \mapsto f(A, C, A), X \mapsto g(A, C), Y \mapsto B, Z \mapsto h(A, B, C, D)\}$. Then,

$shared(\theta) = \{\emptyset, \{W, X, Z\}, \{Y, Z\}, \{Z\}\}$. □

Variables A and C occur through the bindings of W, X, and Z since $W\theta = f(A, C, A)$, $X\theta = g(A, C)$, and $Z\theta = h(A, B, C, D)$; so, $occ(\theta, A) = \{W, X, Z\}$, and $occ(\theta, C) = \{W, X, Z\}$ as expected. Similarly, B occurs in $Y\theta$ and $Z\theta$ and $occ(\theta, B) = \{Y, Z\}$. Hence, $\{Y, Z\}$ is one of the components of $shared(\theta)$. The variable D in the range of θ occurs through only one variable Z and hence $occ(\theta, D) = \{Z\}$. All other variables in Var occur through no variable in θ, *i.e.,* they occur through the empty set of variables and hence $\emptyset$ is also in $shared(\theta)$.

All variables in the range of an idempotent substitution θ are free in θ. They are placeholders and serve the purpose of signifying sharing, if any, among the variables of the domain of θ. Therefore, we are not interested in the range variables themselves and they do not appear in $shared(\theta)$.

An idempotent substitution may cause sharing only among the variables in its domain. In the above example, θ does not affect the variables A, B, C, and D in its range. The variables W, X, Y, and Z in the domain of θ are made to share by the substitution θ. Thus, the definition of $occ(\theta, D)$, for example, simply says which variables share D. Likewise, $occ(\theta, A)$ says which variables share A and $occ(\theta, C)$ denotes the variables that share C. Note that $occ(\theta, A) = occ(\theta, C) = \{W, X, Z\}$ and since $shared(\theta)$ is a set, $\{W, X, Z\}$ occurs only once in $shared(\theta)$.

If Z becomes ground, all other variables in the domain of θ in this example will become ground. This is represented by the fact that Z occurs in all sharing sets of θ. If we remove all sets in which Z occur when Z gets bound to a ground term, then other variables W, X and Y will also become ground (since they will no longer appear in a sharing set). Likewise, if W, X and Y all become ground, Z may still not become ground. This is because variable

D occurs through only Z. Therefore, the groundness of Z depends not only on W, X and Y but also on Z itself. This is represented by the singleton set $\{Z\}$ in the *shared*(θ). In other words, a part of the value of Z is dependent on the values of W, X and Y and another part of Z is independent of the values of W, X and Y. The function *shared*(θ) not only captures possible sharing but also possible independence of the domain variables of θ.

The abstract domain *Free* is used to derive a set of variables that will *definitely* be free at a given program point in all possible executions. In the *Sharing* domain we were satisfied to know if variables could possibly have shared subterms, so set union was used to define the lub operation in that domain. Here, set intersection is the lub operation.

Definition 3.4 [Abstract domain *Free*]

Free: $\mathcal{P}(Var)$

$\forall x, y \in Free,\ \ x \sqsubseteq_{Free} y \ \textit{iff}\, x \supseteq y$

$\forall x, y \in Free,\ \ x \sqcup y = x \cap y$ □

A variable in the domain of a substitution is free if it is bound to another variable term. The following function *Free*(θ) defines the set of variables that are free in θ:

Definition 3.5

$free : Subst \rightarrow Free$

$free(\theta) = \{x_i \mid x_i \in dom(\theta)\ \wedge\ x_i \mapsto x_j \in \theta\ \wedge\ x_j \in Var\}$ □

Note the fact that substitutions are idempotent guarantees that no variables on the right hand sides of bindings $x_i \mapsto x_j$ can also be on the left hand side of some other binding.

Example 3.2

Let $\theta = \{W \mapsto f(A, C, A), X \mapsto g(A, C), Y \mapsto B, Z \mapsto h(A, B, C, D)\}$. Then,
$free(\theta) = \{Y\}$. □

The domain *Repeat* represents those variables that are bound to terms which may have repeated variable subterms. A larger set of substitutions may induce a larger set of variables which have repeated variable subterms. Hence, set union is the lub operation on *Repeat*.

Definition 3.6 [Abstract domain *Repeat*]

Repeat: $\mathcal{P}(Var)$

$\sqsubseteq_{Repeat} = \subseteq$

$X_1 \sqcup X_2\ =\ X_1 \cup X_2,$ for $X_1, X_2 \in Repeat$ □

Function *Repeat*(θ) returns a set of variables that are bound to terms containing multiple occurrences of at least one variable:

Definition 3.7

$repeat : Subst \rightarrow Repeat$

$repeat(\theta) = \{x_j \mid x_j \mapsto t_j \in \theta\ \wedge\ \exists x_k \in Var \mid x_k \text{ occurs more than once in } t_j\}$ □

Example 3.3

Let $\theta = \{W \mapsto f(A, C, A), X \mapsto g(A, C), Y \mapsto B, Z \mapsto h(A, B, C, D)\}$. Then
$repeat(\theta) = \{W\}$. □

Note that we lose information about the number of occurrences of repeated variables and the positions in which they occur. This may cause some loss of precision in sharing analysis but is not expected to cause any significant loss in most cases. Keeping this information, on the other hand, would make the domains too concrete, the abstract unification more complicated, and the analyses too costly.

The following are the invariant properties of our abstract substitution which will be used in abstract unification.

Proposition 3.1

If $x \in dom(\theta) \ \wedge \ x \notin Vars(shared(\theta))$, then $Vars(x\theta) = \emptyset$.

By extension, if $shared(\theta) = \{\emptyset\}$, then $\forall x \in dom(\theta).\ Vars(x\theta) = \emptyset$. □

Proof

If a variable x in the domain of a substitution θ does not appear in $Vars(shared(\theta))$, then no variable u occurs through x and hence x is bound to a ground term in θ, *i.e.*, $Vars(x\theta) = \emptyset$. By extension, if $\emptyset$ is the only element of $shared(\theta)$, then no variable occurs through any of the variables in $dom(\theta)$ and hence variables in $dom(\theta)$ are bound to ground terms in θ. □

Example 3.4

Consider a substitution $\theta = \{X \mapsto f(A, B), Y \mapsto g(2, A), Z \mapsto a\}$. Notice that the set of sharing sets $shared(\theta) = \{\{X, Y\}, \{X\}\}$ signify that X and Y share and X is also independent of Y. Since no variable occurs through Z (in other words, Z is bound to a ground term), Z does not occur in any sharing set in $shared(\theta)$. If a variable Z in the domain of θ does not occur in a sharing set of θ, it implies that no variable occurs through Z and hence Z must be ground. If Z is not in the domain of θ, nothing can be said about Z. □

Proposition 3.2

$\forall \theta \in Subst.\ free(\theta) \subseteq Vars(shared(\theta))$.

$\forall \theta \in Subst.\ repeat(\theta) \subseteq Vars(shared(\theta))$.

$\forall \theta \in Subst.\ free(\theta) \cap repeat(\theta) = \emptyset$. □

Proof

This follows from the fact that x is free in θ implies that x is bound to some variable u in the range of θ and hence x is in some set in $shared(\theta)$. Likewise, x has a repeating variable subterm u means that u occurs through x and hence x is in some set in $shared(\theta)$. Furthermore, x is free in θ means that x does not have a repeating variable subterm and hence x is not in $repeat(\theta)$. Likewise, x is non-linear in θ implies x is bound to a non-variable term and hence x is not free in θ. □

Having introduced the domains *Free*, *Repeat* and *Sharing*, we now define *Asub* as the product of these three domains. The partial ordering on and lub of *Asub* are derived from the component domains. Domain subscripts are omitted when there can be no confusion.

Definition 3.8

Abstract Domain $Asub = Free \times Repeat \times Sharing$

$Asub_0 \sqsubseteq_{Asub} Asub_1$ *iff*

$Asub_0 \downarrow Free \sqsubseteq_{Free} Asub_1 \downarrow Free$ and

$Asub_0 \downarrow Repeat \sqsubseteq_{Repeat} Asub_1 \downarrow Repeat$ and

$Asub_0 \downarrow Sharing \sqsubseteq_{Sharing} Asub_1 \downarrow Sharing.$ □

Finally, we define the abstraction and concretization functions that are mappings between the concrete and abstract domains. The abstraction function α maps a set of substitutions to an element of *Asub*. We define α in terms of the abstraction functions *free*, *repeat* and *shared*.

Definition 3.9 [Abstraction Function]

$\alpha : \mathcal{P}(Subst) \rightarrow Asub$

$\alpha(\Theta) = \langle \bigsqcup_{\theta\in\Theta} Free(\theta), \bigsqcup_{\theta\in\Theta} Repeat(\theta), \bigsqcup_{\theta\in\Theta} shared(\theta) \rangle$ □

The abstraction of a set of substitutions Θ includes all possible sharing of variables induced by each substitution $\theta \in \Theta$, since set union is the lub operation for *Sharing*. Likewise, a variable is ground in the abstraction of Θ only if it is ground in all the substitutions $\theta \in \Theta$. The *Repeat* component of the abstraction of Θ includes a variable x if x is bound to a term with a repeated variable occurrence in at least one $\theta \in \Theta$. Finally, a variable is free in an abstract substitution Θ only if it is free in all $\theta \in \Theta$.

The concretization function γ is uniquely determined by our abstraction function and thus our definitions of α and γ define a Galois insertion between *Asub* and $\mathcal{P}(Sub)$.

Definition 3.10 [Concretization function]

$\gamma : Asub \rightarrow \mathcal{P}(Subst)$

$\gamma(A) = \bigsqcup\{\Theta \in \mathcal{P}(Subst) \mid \alpha(\Theta) \sqsubseteq A\}$ □

3.3 Operations on the Abstract Domain

In this section, we define the abstract domain operations *entry_sub* and *success_sub*. In abstract interpretation, as in concrete interpretation, the central operation is unification. We will discuss how information in the heads of a called procedure is composed (reconciled) with the current abstract substitution to form the abstract substitution used in the next step. We start with definitions of functions *relevant*, *closure under union* (due to Jacobs and Langen [44]), *restrict*, and *non_linear* that are needed to define abstract unification.

In our abstract domain *Asub*, the position and structure of subterms is not known, *i.e.*, the precise structure of terms is abstracted away. This makes the abstract unification more difficult than its concrete counterpart. We introduce some terminology to refer to the subterms of a term in a position- and structure-independent way, given an abstract substitution. Consider the following concrete substitution and its abstraction:

$$\begin{array}{rcl} \sigma & = & \{W \mapsto f(g(A,C),g(A,C)), X \mapsto g(A,C), Y \mapsto B, Z \mapsto h(A,B,C,D)\} \\ \alpha(\{\sigma\}) & = & \langle \{Y\}, \{W\}, \{\emptyset, \{W,X,Z\}, \{Y,Z\}, \{Z\}\} \rangle \end{array}$$

The set of free variables F is $\{Y\}$, the set of non-linear variables R is $\{W\}$ and the set of sets of sharing variables SS is $\{\emptyset, \{W, X, Z\}, \{Y, Z\}, \{Z\}\}$. We lose information about the position and structure of subterms when we abstract a substitution. However, we can refer to subterms indirectly as follows.

Let a component of a variable X be a non-empty subset of the set of variables in a term t that X is bound to in a substitution σ. The domain *Sharing* represents a variable by its components and collects together in a set all the variables that *share* the same component(s). In the above example, variable Z has the components $\{A, C\}$, $\{B\}$ and $\{D\}$ represented by the sets $\{W, X, Z\}$, $\{Y, Z\}$, $\{Z\}$ in *shared*(σ). Variable Z shares $\{A, C\}$ with W and X, $\{B\}$ with Y, and $\{D\}$ with none.

Each set in a *Sharing* may be viewed in two ways. First, it is a set of variables that *share* the same component(s). For example, the variables in the set $\{W, X, Z\}$ share the component $\{A, C\}$. The emphasis here is on sharing. Second, each set in a *Sharing* represents a particular component of each of the variables in the set. For example, the set $\{W, X, Z\}$ represents some particular component of W, X and Z. We do not know what that component is or where it occurs or how many times it occurs (given that we only have an abstract substitution and not its concretization). We call each set I (in a *Sharing*) a component of each of the variables that occurs in I.

We now define a function *relevant* used for identifying the components of a term or other syntactic objects. Suppose a term t is unified with another term in the environment described by the above abstract substitution. We can abstract the effect of this unification on t by considering its effect on t's sharing, freeness and linearity. Since t's sharing is completely described by t's components in the current abstract substitution, the effect on t's sharing can be abstracted by considering the effect on t's components. Given a syntactic object t and an abstract substitution, the components of t are given by the function *relevant* defined below.

Definition 3.11

$relevant : A \times Sharing \rightarrow Sharing$

$relevant(t, S) = \{X \in S \mid Vars(t) \cap X \neq \emptyset\}$ □

Example 3.5

Let $\sigma = \{W \mapsto f(g(A, E)), Y \mapsto B, Z \mapsto h(A, B, C)\}$.

Then the sharing component of the abstraction of σ is SS = *shared*(σ) = $\{\emptyset, \{W\}, \{W, Z\}, \{Y, Z\}, \{Z\}\}$. The sharing set $\{W\}$ denotes the occurrence of E only through W, the set $\{W, Z\}$ denotes the occurrence of A through both W and Z, $\{Y, Z\}$ denotes B occurring through both Y and Z, and $\{Z\}$ denotes the occurrence of C only through Z. Z has as its components *relevant*(Z,SS) = {{W,Z}, {Y,Z}, {Z} }. □

Suppose Z in the above example now becomes further instantiated as the result of another unification. Such further instantiations may cause some distinct subterms of Z to share; for example, A and B will share if Z gets unified with a term $t = h(Q, Q, R)$. This will cause W and Y to share the variable A with Z. Note that W and Y did not share previously. This can be handled by replacing the sharing sets $\{W, Z\}$ and $\{Y, Z\}$ with their union. On the other hand, Z may be unified with a term $t = h(R, S, T)$ which does not affect the sharing among W, Y and Z and hence the sharing sets $\{W, Z\}$ and $\{Y, Z\}$ remain.

If we do not know anything about t, then we have to account for both the above possibilities and this is accomplished by taking the closure under union of the relevant sets of Z. The closure under union of the relevant sets $\{\{W, Z\}, \{Y, Z\}\}$ is $\{\{W, Z\}, \{Y, Z\}, \{W, Y, Z\}\}$ which represents the possibility that W, Y, and Z may share and also the possibility that W and Y may be independent. Thus, the closure under union of *Sharing* $S \in SS$ approximates further sharing among the variables of S that may be caused.

Definition 3.12 [Closure under union]

The closure under union of a *Sharing* S, denoted S^*, is the smallest superset of S such that $X \in S^* \wedge Y \in S^* \Rightarrow X \cup Y \in S^*$. □

Function *restrict* is the analog of the concrete function with the same name, used to restrict a given abstract substitution to the variables of a clause or to the variables in the goal's environment.

Definition 3.13

restrict : $\mathit{Asub} \times \mathcal{P}(\mathit{Var}) \rightarrow \mathit{Asub}$

restrict$(\langle F, R, S\rangle, V) = \langle F \cap V, R \cap V, \{X' \mid X \in S \;\wedge\; X' = X \cap V\}\rangle$ □

As mentioned before, we use the linearity of terms to derive a more precise abstract unification algorithm. The following definition of *non_linear* determines if a term is non-linear with respect to an abstract substitution. A term t may have repeating variable subterms only if one of the following conditions hold:

- A variable subterm of T occurs in the *Repeat* component of the abstract substitution.

- The same variable occurs in T more than once and the variable is not known to be definitely ground with respect to the given abstract substitution.

- Two distinct variable subterms of T may share in the given abstraction.

These are the only possibilities. This definition safely over-estimates the set of non-linear variables with respect to an abstract substitution. This is because the sharing and repeat components of an abstract substitution represent possible sharing and non-linearity and not definite sharing and non-linearity.

Recall that in an abstract substitution $\langle F, R, S\rangle$, F is the set of Free variables, R is the set of non_linear variables and S is the set of sharing sets. We cannot define non-linearity without referring to an abstract substitution. This is because a term such as $g(X, X)$ may be linear with respect to an abstract substitution Asub_θ which maps X to a ground term, in which case there are no variables at all in $g(X, X)$.

Example 3.6

A term $g(X)$ is non-linear with respect to the set of substitutions whose abstraction is $\langle\{\}, \{X\}, \{\{X\}\}\rangle$. This is because X occurs in the *Repeat* component.

A term $g(X, Y, X)$ is non-linear with respect to the abstract substitution $\langle\{\}, \{\}, \{\{X\}\}\rangle$. This is because X occurs more than once in that term and X is not known to be ground. Recall that if a variable occurs in a sharing set, then it is not known to be definitely ground.

The term $g(X,Y,X)$ is linear with respect to the abstract substitution $\langle \{\}, \{\}, \{\{\}\} \rangle$ because the variables X and Y do not occur in the sharing component of the abstract substitution and hence must be bound to ground terms.

Finally, the term $g(X,Y)$ may be non-linear given the abstract substitution $\langle \{\}, \{\}, \{\{X,Y\}\} \rangle$. This is because the sharing set $\{X,Y\}$ in the abstract substitution implies that X and Y may share a common variable. Hence the term $g(X,Y)$ may have a repeating variable subterm with respect to the given abstract substitution. □

Definition 3.14

non_linear : *Term* × *Asub* → *Bool*
non_linear$(t, Asub_0)$ *iff*
$(\exists X \in Vars(t)$ *s.t.* $X \in Asub_0 \downarrow Repeat) \ \vee$
$(\exists X \in Vars(t)$ *s.t.* X occurs more than once in t $\wedge X \in Vars(Asub_0 \downarrow Sharing)) \vee$
$(\exists X, Y \in Vars(t),\ X \neq Y,\ \exists S \in Asub_0 \downarrow Sharing$ *s.t.* $\{X,Y\} \subseteq S)$ □

We now discuss the main abstract domain operations *abstract_unify*, *entry_sub*, and *success_sub*. Function *entry_sub* abstracts the effect of head unification and is used to propagate information from the abstract environment of a goal to the abstract environment of the clause whose head matched the goal. The abstract operation *success_sub* is used for propagating information in the other direction, *i.e.,* it abstracts the effect of composing the exit substitution of a clause with the call substitution of a goal and restricting the result to the variables of the goal environment.

Function *entry_sub* is an abstraction of the standard domain operation $mgu(a,h) \uparrow \theta$ where θ is the call substitution of goal a and h is the head of a renamed clause. Before we reconcile the constraints $mgu(a,h)$ with abstract call substitution $Asub_{call}$, the abstract call substitution is augmented with information about clause variables. Just before head–goal unification, there are no constraints on the variables of the renamed clause. This means that the clause variables are free, pairwise independent, and linear. Note that extending the abstract environment of the goal with the clause variables does not cause problems precisely because of variable renaming. The effect of $mgu(a,h) \uparrow \theta$ is abstracted by the function *abstract_unify*, given the mgu of a and h and the extended abstract call substitution. As mentioned before, restricting the result of *abstract_unify* to the renamed clause variables and applying the inverse of the renaming function gives us an entry substitution in terms of the original clause variables. As mentioned before, the renaming index is unused in our interpretation.

Definition 3.15

```
$entry\_sub : Atom \times Clause \times Asub \times N \rightarrow Asub$
$entry\_sub(a, h' \leftarrow b', Asub_{call}, i) =$
 let  $h \leftarrow b$          $= rename(h' \leftarrow b')$
      $F$                      $= Asub_{call}\downarrow Free \cup Vars(h \leftarrow b)$
      $R$                      $= Asub_{call}\downarrow Repeat$
      $S$                      $= Asub_{call}\downarrow Sharing \cup \{\{X\} \mid X \in Vars(h \leftarrow b)\}$
      $Asub_{entry'}$          $= abstract\_unify(mgu(a,h), \langle F, R, S\rangle)$
 in
      $rename^{-1}(restrict(Asub_{entry'}, Vars(h \leftarrow b)))$
 end
```

□

In the following example, we use the renamed version of a clause and omit the renaming and inverse renaming steps for simplicity.

Example 3.7

Let $c = h \leftarrow b$ be a clause, a a goal, and $Asub_{call}$ be a's call substitution as defined below.

$$\begin{aligned}
\text{atom } a &= p(X, f(X,X), Y, Z) \\
\text{clause head } h &= p(f(P,Q), f(R,S), g(a,b), g(R,T)) \\
\text{clause body } b &= q(S,T,U) \\
Asub_{call} &= \langle \{X,Z\}, \{Y\}, \{\{Y\}, \{X,Y\}, \{Z, Z1\}\}\rangle
\end{aligned}$$

The entry substitution for clause c is computed as follows:

$$\begin{aligned}
F &= \{X,Z\} \cup \{P,Q,R,S,T,U\} \\
R &= \{Y\} \\
S &= \{\{Y\}, \{X,Y\}, \{Z,Z1\}, \{P\}, \{Q\}, \{R\}, \{S\}, \{T\}, \{U\}\} \\
mgu(a,h) &= \{X \mapsto f(P,Q), R \mapsto f(P,Q), S \mapsto f(P,Q), Y \mapsto g(a,b), \\
&\qquad Z \mapsto g(f(P,Q),T)\} \\
Asub_{entry'} &= abstract\_unify(mgu(a,h), \langle F, R, S\rangle) \\
&= \langle \{T,U\}, \emptyset, \{\{Z, Z1, T\}, \{U\}\}\rangle
\end{aligned}$$

For now, take $abstract\_unify(mgu(a,h), \langle F, R, S\rangle)$ to be $\langle \{T,U\}, \emptyset, \{\{Z, Z1, T\}, \{U\}\}\rangle$. (Function *abstract_unify* will be defined shortly.) Restricting $Asub_{entry'}$ to the clause variables yields the entry substitution $\langle \{T,U\}, \emptyset, \{\{T\}, \{U\}\}\rangle$. □

Function *success_sub* is similar to *entry_sub*. It uses *abstract_unify* to abstract the standard domain operation $\theta \uparrow mgu(a,h) \uparrow \theta_{exit}$ where θ is the call substitution of the goal a. Since $\uparrow$ is associative and commutative, the above expression is equivalent to $mgu(a,h) \uparrow \theta \uparrow \theta_{exit}$ which, using the definition of $\uparrow$ (see Chapter 2.4), is simplified to $mgu(a,h) \uparrow (Eqn(\theta) \cup Eqn(\theta_{exit}))$. The unifier $mgu(a,h)$ can be computed exactly and $(Eqn(\theta) \cup Eqn(\theta_{exit}))$ is obtained as the component-wise union of the corresponding abstract call and exit substitutions. Note that taking component-wise union does not violate the properties of an abstract substitution (Propositions 3.1 and 3.2) because of renaming of the variables of the clause and the exit substitution consistently. Note also that we use *rename′* in *success_sub* instead of *rename* used in *entry_sub* solely because the types are different. Since success substitution is meant to pass information to the goal environment, the result of *abstract_unify* is restricted to program variables.

Definition 3.16

$success\_sub : Atom \times Clause \times Asub \times Asub \rightarrow Asub$

$success\_sub(a, h \leftarrow b, Asub_{call}, Asub_{exit}) =$

let $(h' \leftarrow b', Asub_{exit'}) = rename'(h \leftarrow b, Asub_{exit})$

$\quad Asub_{succ} = abstract\_unify(mgu(a, h'), Asub_{exit'} \cup Asub_{call})$

in

$\quad restrict(Asub_{succ}, PVar)$

end

□

In the following example, the clause c, the goal a, and the call substitution for the goal a are the same as in Example 3.7.

Example 3.8

$$\begin{aligned} \text{atom } a &= p(X, f(X,X), Y, Z) \\ \text{clause head } h &= p(f(P,Q), f(R,S), g(a,b), g(R,T)) \\ \text{clause body } b &= q(S,T,U) \\ Asub_{call} &= \langle \{X,Z\}, \{Y\}, \{\{Y\}, \{X,Y\}, \{Z,Z1\}\} \rangle \end{aligned}$$

□

Assume that solving the only body goal $q(S,T,U)$ results in the exit substitution $Asub_{exit} = \langle \emptyset, \{T\}, \{\{T,U\}, \{U\}\} \rangle$. When computing the entry substitution for clause c, we used the call environment for the goal a augmented with the initial environment for the clause as the environment for solving the equation $a = h$. When computing the success substitution, the exit substitution of the clause is used instead of the initial substitution. The success substitution of goal a with respect to clause c and corresponding to the call substitution $Asub_{call}$ is computed as follows. As in the previous example, we omit the renaming step for simplicity.

$$\begin{aligned} F &= \{X,Z\} \cup \emptyset \\ R &= \{Y\} \cup \{T\} \\ S &= \{\{Y\}, \{X,Y\}, \{Z,Z1\}, \{T\}, \{T,U\}, \{U\}\} \\ mgu(a,h) &= \{X \mapsto f(P,Q), R \mapsto f(P,Q), S \mapsto f(P,Q), Y \mapsto g(a,b), \\ &\quad Z \mapsto g(f(P,Q),T)\} \\ Asub_{succ} &= abstract\_unify(mgu(a,h), \langle F,R,S \rangle) \\ &= \langle \emptyset, \{Z,Z1,T\}, \{\{Z,Z1,T\}, \{Z,Z1,T,U\}\} \rangle \end{aligned}$$

For now, assume $abstract\_unify(mgu(a,h), \langle F,R,S \rangle)$ to be $\langle \emptyset, \{Z,Z1,T\}, \{\{Z,Z1,T\}, \{U\}, \{Z,Z1,T,U\}\} \rangle$. (Function *abstract_unify* will be defined shortly.) Restricting $Asub_{succ}$ to the program variables PVar yields the success substitution $\langle \emptyset, \{Z,Z1\}, \{\{Z,Z1\}\} \rangle$ for the goal a corresponding to the call substitution $\langle \{X,Z\}, \{Y\}, \{\{Y\}, \{X,Y\}, \{Z,Z1\}\} \rangle$.

We now discuss abstract unification.

3.3.1 *Abstract_Unify*

Function *abstract_unify* updates an abstract substitution $Asub_{in}$ to reflect the effects of a set of equations in solved form (*i.e.*, the mgu of a head and a goal). These equations represent the equality constraints to be reconciled with the current environment whose abstraction is $Asub_{in}$. All we have to do is to safely abstract the effect of each of these equations on the abstract substitution. These equations have the form $X = T$ where X is a variable and T is a term (possibly a variable term). Function *abstract_unify* mimics the effects of each of these equations using function *refine*.

Definition 3.17

```
    $abstract\_unify : \mathcal{P}(Eqns) \times Asub \rightarrow Asub$
    $abstract\_unify(Eqns, Asub_0) =$
      if $Eqns = \emptyset$ then $Asub_0$
      else let  $E$ be an equation in $Eqns$
        $Eqns' = Eqns \setminus \{E\}$
        $Asub_1 = refine(Asub_0, E)$
      in
        $abstract\_unify(Eqns', Asub_1)$
      end
```

□

Since *abstract_unify* simply calls *refine* to approximate the effect of solving an equation E in the set of equations of *Eqns*, we will discuss a complete example after defining *refine*.

The main component of abstract unify is *refine*. It takes an equation $v_i = t_i$ in solved form (*i.e.*, v_i is a variables and $v_i \notin Vars(t_i)$) and an abstract substitution $Asub_{cur}$ and refines the abstract substitution by safely approximating the effects of the equation on the abstract substitution.

In the rest of the section, θ is some substitution in the concretization of the current abstract substitution $Asub_{cur}$. In other words, θ is some substitution safely approximated by $Asub_{cur}$. Note that there will be infinite number of substitutions in the concretization of $Asub_{cur}$. We will not assume anything about θ except what is implied by the current abstract substitution $Asub_{cur}$. We will use the properties of an element of the abstract domain, established in propositions 3.1 and 3.2. The effects of solving the equation $v_i\theta = t_i\theta$ on the abstract substitution $Asub_{cur}$ is given by the function *refine* in Definition 3.18. An overview of the the function *refine* follows.

Outline of *refine*: The effect of $v_i\theta = t_i\theta$ on $Asub_{cur}$ is derived by case analysis. We first consider the case when there is no solution to the equation $v_i\theta = t_i\theta$. Next, we consider when $v_i\theta = t_i\theta$ succeeds. In this case, there are two sub-cases to consider. The first and easy case is when at least one of $v_i\theta$ or $t_i\theta$ is a ground term. In the second sub-case when neither v_i nor t_i is a ground term, we consider the effects of the equation $v_i\theta = t_i\theta$ on the sharing, non-linearity, and freeness components of $Asub_{cur}$ in that order. Each of these sub-cases will have sub-cases depending on the structure of the terms v_i and t_i and their properties implied by $Asub_{cur}$. We use examples in each of these cases.

The equation $v_i\theta = t_i\theta$ may or may not have a solution, (*i.e.*, the reconciliation $(\{v_i \mapsto t_i\} \uparrow \theta)$ may succeed or fail). Failure is represented by the empty set of substitutions in the standard domain of computation $\mathcal{P}(Subst)$. Note that *refine* (Definition 3.18) never fails since we do not have enough information in the abstract substitution to capture failure. Thus, the concretization of the abstract substitution returned by *refine* is a non-empty set of substitutions, which is a safe approximation of the empty set of substitutions.

Consider the case where the equation $v_i\theta = t_i\theta$ has a solution σ. Given that a solution σ exists, $(v_i\theta)\sigma \equiv (t_i\theta)\sigma$, where $\equiv$ is syntactic identity on terms, atoms, equations and other syntactic objects. In the following, we will see how to approximate $v_i\theta = t_i\theta$ safely, using invariant properties of the current abstract substitution $Asub_{cur}$.

There are two cases to consider when a solution σ exists for the equation $v_i\theta = t_i\theta$. In the first case, either $v_i\theta$ or $t_i\theta$ is a ground term. In the second case, neither is a ground term.

If $v_i\theta$ is a ground term *i.e.*, $Vars(v_i\theta) = \emptyset$, then $v_i\theta\sigma$ is also ground. *i.e.*, a ground term cannot be further instantiated. Hence, $t_i\theta\sigma$ is also ground term. A variable bound to a ground term is not free, does not share with other variables, and has no repeating variable subterms. Hence, we remove the variables that are relevant to v_i and t_i from the *Free* and *Repeat* components of the abstract substitution $Asub_{cur}$ and remove the relevant sharing sets of v_i and t_i from the *Sharing* component of $Asub_{cur}$. Notice that this maintains the invariance properties 3.1 and 3.2. The case where $t_i\theta$ is a ground term is analogously defined.

Example 3.9

Let $Asub_{cur} = \langle \{X,Y\},\{P,Q\},\{\emptyset,\{A,B,C\},\{P\},\{A,X\},\{B,Y\},\{A,Q\}\}\rangle$. Consider the equation $D = f(X,P,A)$. Since D does not appear in any of the sharing sets in the current abstract substitution, it is definitely ground. Since the term $f(X,P,A)$ is unified with D, the variables X,P and A also become ground terms. Hence, the new abstract substitution $Asub_{new}$ is $\langle\{Y\},\{\},\{\emptyset,\{B,Y\}\}\rangle$. Note that in the abstract substitution $Asub_{cur}$, the groundness of A implies the groundness of B,C,X and Q, since A occurs in all the sharing sets in which B,C,X and Q occur. □

Now consider the case where neither $v_i\theta$ nor $t_i\theta$ is a ground term. This impacts the sharing, linearity, and freeness components of $Asub_{cur}$. We begin with the *Sharing* component.

Sharing Propagation The equation $v_i\theta = t_i\theta$ may cause three types of sharing: some subterm of $v_i\theta$ may share with some subterm of $t_i\theta$; some subterm of $v_i\theta$ may share with some other subterm of $v_i\theta$; likewise, a subterm of $t_i\theta$ may share with another subterm of $t_i\theta$. We need to safely approximate this sharing. We derive safe approximations of the sharing as follows.

The effect of an equation $v_i\theta = t_i\theta$ depends on whether or not $v_i\theta$ (or $t_i\theta$) may be a non-linear term, *i.e.*, may have one or more repeated variable occurrences. This leads to four possibilities, depicted in Figure 3.2. A broken edge from v_i to t_i represents unification of the two terms. A solid edge from v_i to itself indicates that v_i may be non-linear in the current substitution θ; similarly a solid edge from a node t_i to itself means the term t_i may be non-linear. For both nodes v_i and t_i, the variables we are concerned with are those that occur in the term $v_i\theta$ and $t_i\theta$ respectively. In the figure, $X_1,\ldots,X_n$ are all the variables in $v_i\theta$, and $Y_1,\ldots,Y_k$ are the variables in $t_i\theta$ where θ, as mentioned before, is in the concretization of the current abstract substitution.

In case (a), both $v_i\theta$ and $t_i\theta$ are linear, *i.e.*, $v_i \notin Asub \downarrow Repeat$ and $Vars(t_i) \cap Asub \downarrow Repeat = \emptyset$. The only type of sharing that is possible is that between a subterm of v_i and a subterm of t_i. This is because neither $v_i\theta$ nor $t_i\theta$ have a repeating variable subterm.

The syntactic identity $v_i\theta\sigma \equiv t_i\theta\sigma$ implies that $Vars(v_i\theta\sigma) = Vars(t_i\theta\sigma)$. This means for each subterm $v_i\theta\sigma$, there is a corresponding subterm of $t_i\theta\sigma$ such that these two subterms are syntactically identical and the set of variables in these two subterms is the same. In an abstract interpretation, we do not have information about positions of subterms. Therefore, we have to conclude that each variable subterm of $v_i\theta$ may be forced by the unifier to share with some variable subterm(s) of $t_i\theta$. This over-estimates the sharing, *i.e.*, it is a conservative and safe approximation but it is the best we can do in the absence of information about the structure of terms.

We compute the pairwise union of each possible pair of sets of variables from $relevant(v_i, Asub \downarrow Sharing)$ and $relevant(t_i, Asub \downarrow Sharing)$ and add these to $Asub \downarrow Sharing$ (since this information supersedes the old relevant sets, they are removed). Note that $relevant(v_i, Asub \downarrow Sharing)$ represents variable subterm(s) of $v_i\theta$ by the set of variables in the domain of θ with which it shares those variable subterms.

Example 3.10

Let the *Sharing* component of the abstract substitution $Asub_{cur}$ be $Sharing_{cur} = \{\emptyset, \{X, Z\}, \{Y, Z\}, \{Z\}, \{P\}\}$ and the *Repeat* component be $\{X\}$. Consider the equation $P = Z$. The set of sharing sets relevant to Z is $\{\{X, Z\}, \{Y, Z\}, \{Z\}\}$. Likewise, the set of sharing sets relevant to P is $\{\{P\}\}$. The pairwise union of $\{\{X, Z\}, \{Y, Z\}, \{Z\}\}$ and $\{\{P\}\}$ is $\{\{P, X, Z\}, \{P, Y, Z\}, \{P, Z\}\}$. The new *Sharing* is $(\{\emptyset, \{X, Z\}, \{Y, Z\}, \{Z\}, \{P\}\} \setminus \{\{X, Z\}, \{Y, Z\}, \{Z\}, \{P\}\}) \cup \{\{P, X, Z\}, \{P, Y, Z\}, \{P, Z\}\}$ *i.e.*, $\{\emptyset, \{P, X, Z\}, \{P, Y, Z\}, \{P, Z\}\}$. Note that the equation $P = Z$ implies that the groundness of P depends on that of Z and vice versa. This is captured by the fact that P and Z occur together in all sharing sets in the new *Sharing*. Furthermore, since P is linear, the distinct components of Z, namely $\{X, Z\}, \{Y, Z\}$, remain distinct after the unification of P and Z. □

Consider case (b) in Figure 3.2. Some Y_i has a repeated subterm, as indicated by the solid arc. The equation $v_i = t_i$ implies that some of the variables $X_1, \ldots, X_n$ may now depend on each other as the result of the repeated variable somewhere in t_i. To handle this case, we compute the closure under union of $relevant(v_i, Sharing)$ and add the pairwise union of this set and $relevant(t_i, Sharing)$ to *Sharing*.

Case (c) is symmetric to case (b), the only difference being that the repeated variable is one of the X_i and the set to close under union and add to *Sharing* is $relevant(t_i, Sharing)$.

Finally, in case (d) in Figure 3.2, since there may be repeated variable subterms in v_i and t_i, both the sets $relevant(v_i, Sharing)$ and $relevant(t_i, Sharing)$ are closed under union and their pairwise union is added to *Sharing*.

Example 3.11

Let $Asub_{cur} = \langle \{X, Y\}, \{P, Q\}, \{\emptyset, \{A, B\}, \{P\}, \{A, X\}, \{B, Y\}, \{B, Q\}\} \rangle$. Consider the equation $Q = f(X, A)$. Note that Q may be non-linear because it appears in the *Repeat* component of $Asub_{cur}$. Although X and A are linear, the

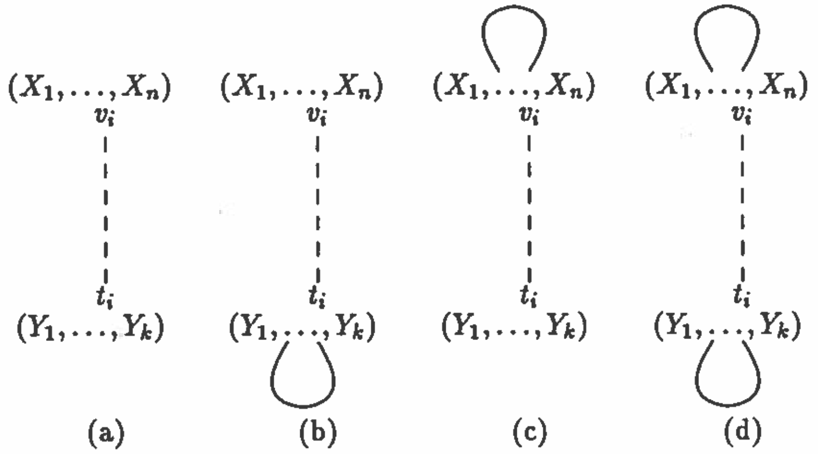

Figure 3.2: Sharing, Freeness and Repeatedness Propagation

term $f(X, A)$ may be non-linear. This is because X and A appear in a sharing set together and hence they may share a variable, which may oocur more than once in the term $f(X, A)$. The definition of non-linearity captures this possibility. The sharing set relevant to Q is: $\{B, Q\}$. Likewise, the sharing sets relevant to $f(X, A)$ are: $\{A, B\}, \{A, X\}$. The closure under union of $\{\{B, Q\}\}$ is itself. The closure under union of $\{\{A, B\}, \{A, X\}\}$ is $\{\{A, B\}, \{A, X\}, \{A, B, X\}\}$. Their pairwise union yields $\{\{A, B, Q\}, \{A, B, Q, X\}\}$. The sharing component of the abstract substitution after solving the equation $Q = f(X, A)$ is $\{\emptyset, \{P\}, \{B, Y\}, \{A, B, Q\}, \{A, B, Q, X\}\}$. Note that if Q becomes ground later on due to another equation, A and X will also become ground. Likewise, if A and B become ground, Q will also become ground. □

Linearity Propagation When updating *Repeat* for an equation $v_i = t_i$, we note that if v_i may be bound to a term which has repeated variable subterms, then t_i now also has repeated variable subterms. Since we do not know which of the variables of v_i repeat, and since we are deriving *potential* non-linearity information, we have to conclude that any subterm of t_i may also have repeated variable subterms. Returning again to the equation $Z = f(T, S)$ and the case where we know Z has repeated variable subterms, we conservatively conclude that T and S may also be bound to terms with repeated variable subterms and add them to *Repeat*.

Even when neither v_i nor t_i has a repeated variable subterm, the equation $v_i = t_i$ may cause two previously distinct variable subterms to become aliased. Consider the equation $P = R$, given substitution $\{P \mapsto (T, U), Q \mapsto (T, U), R \mapsto (U, N)\}$ and the sharing component of its abstraction $\{\{P, Q\}, \{P, R\}, \{R\}\}$. Two previously distinct components T and U in the value of P are now aliased to each other and P, Q, and R will now have a repeated variable occurrence. Such cases can arise only when there is at least one variable (U in this situation) that occurs through both v_i and t_i. We must conclude that v_i, all the variables in t_i, and variables that share with them may be bound to a term with repeated variable subterms.

Freeness Propagation We now consider how freeness of variables is affected due to the equation $v_i\theta = t_i\theta$. If v_i is not free in θ, we have to conclude conservatively that none of the variables in $t_i\theta$ may be free. This is because we are deriving definite freeness.

Consider the equation $X = f(Y, Z)$ and an abstract substitution in which X is not free. For example, θ could be $\{X \mapsto f(2,3), Y \mapsto A, Z \mapsto A', P \mapsto A, Q \mapsto A'\}$. The variables A, A', Y, Z, P and Q are no longer free. Thus if v_i is not free in the current abstract substitution $Asub_{cur}$, then we have to assume that all the variables that share with the term t_i may not be free any more.

On the other hand, if v_i is definitely free in the current substitution θ, then $v_i\theta$ is some unbound variable X and the equation $v_i\theta = t_i\theta$ may at the most create the binding $X \mapsto t_i$, and this does not affect the freeness of variables that occur in t_i. We used the phrase "may at the most create the binding" because t_i may also turn out to be a free variable Y and the binding created may be $Y \mapsto X$ instead of $X \mapsto Y$. The above reasoning holds in either case.

If t_i is not a variable or it is a variable but not free in the current abstract substitution, then v_i and other variables it shares with may no longer be free. These are also removed from the set of variables. On the other hand if t_i is a free variable Y in the current substitution θ, once again the equation $v_i\theta = t_i\theta$ may at the most create the binding $Ymapstov_i$ which does not affect the freeness of v_i.

The following definition of *refine* captures the above case analysis.

Definition 3.18

```
$refine : Asub \times Eqn \rightarrow Asub$
$refine(\langle F, R, S\rangle, v_i = t_i) =$
if $ground(v_i, S) \vee ground(t_i, S)$ then
    let  $A = relevant(v_i, S),\ B = relevant(t_i, S)$
         $S' = S \setminus (A \cup B)$
         $R' = R \cap Vars(S')$
         $F' = F \setminus Vars(A \cup B)$
    in  $\langle F', R', S'\rangle$
    end
else
    let  $A = relevant(v_i, S),\ B = relevant(t_i, S)$
         $B' =$ if $non\_linear(v_i, \langle F, R, S\rangle)$ then $B^*$ else $B$ fi
         $A' =$ if $non\_linear(t_i, \langle F, R, S\rangle)$ then $A^*$ else $A$ fi
         $S' = (S \setminus (A \cup B)) \cup$ pairwise union of $A'$ and $B'$
         $R' = R \cup$ (if $non\_linear(v_i, \langle F, R, S\rangle)$ then $\bigcup B$ else $\emptyset$ fi)
         $R'' = R' \cup$ (if $non\_linear(t_i, \langle F, R, S'\rangle)$then $\bigcup A$ else $\emptyset$ fi)
         $R''' = R'' \cup (\bigcup\{X \cap Y \mid X \in A',\ Y \in B'\})$
         $F' = Free \setminus$ (if $v_i \notin F$ then $\bigcup relevant(t_i, S')$ else $\emptyset$ fi)
         $F'' = F' \setminus$ (if $(t_i \notin Var \vee t_i \notin F')$ then $\bigcup A$ else $\emptyset$ fi)
    in  $\langle F'', R''', S'\rangle$
    end
 fi
```

□

Proposition 3.3

Let θ be any substitution in the concretization of the current abstract substitution, $Asub_{cur}$ and $v_i = t_i$ be an equation in solved form (*i.e.*, $v_i \notin Vars(t_i)$). Then $(\{v_i \mapsto t_i\} \uparrow \theta) \subseteq \gamma(refine(Asub_{cur}, v_i = t_i))$. □

We do not give a rigorous proof of the above proposition but point out that the function *refine* was derived by exhaustive case analysis of the groundness, linearity, and freeness properties of the terms v_i and t_i using the invariant properties of abstract substitutions (Propositions 3.1 and 3.2).

Recall that Examples 3.7 and 3.8 assumed the result of function *abstract_unify* applied to some arguments. We now derive the result of the function call *abstract_unify* in Example 3.7. The result of the call to *abstract_unify* in Example 3.8 may be derived analogously. The relevant details from Example 3.7 are shown below.

Example 3.12

$$\begin{array}{rcl}
F & = & \{X, Z, P, Q, R, S, T, U\} \\
R & = & \{Y\} \\
S & = & \{\{Y\}, \{X, Y\}, \{Z, Z1\}, \{P\}, \{Q\}, \{R\}, \{S\}, \{T\}, \{U\}\} \\
mgu(a, h) & = & \{X \mapsto f(P, Q), R \mapsto f(P, Q), S \mapsto f(P, Q), Y \mapsto g(a, b), \\
 & & \quad Z \mapsto g(f(P, Q), T)\} \\
Asub_{entry'} & = & abstract\_unify(mgu(a, h), \langle F, R, S\rangle)
\end{array}$$

□

There are 5 bindings in the unifier of a and h. We use *refine* to approximate the effects of these bindings, one by one, on the abstract substitution $\langle F, R, S\rangle$.

Consider the binding $X \mapsto f(P, Q)$. Neither X nor $f(P, Q)$ is a ground term in the abstract environment $\langle F, R, S\rangle$. The relevant sharing sets A and B of X and $f(P, Q)$ are computed as follows. $A = relevant(X, S) = \{\{X, Y\}\}$ and $B = relevant(f(P, Q), S) = \{\{P\}, \{Q\}\}$. Since both X and $f(P, Q)$ are linear, there is no need to take the closure under union of A and B. Hence, $A' = A$ and $B' = B$.

$$\begin{array}{rcl}
S_1 & = & S \setminus (A \cup B) \cup pairwise\_union(A, B) \\
 & = & \{\{Y\}, \{Z, Z1\}, \{R\}, \{S\}, \{T\}, \{U\}, \{X, Y, P\}, \{X, Y, Q\}\} \\
R_1 & = & R = \{Y\} \\
F_1 & = & F \setminus (\bigcup A) = \{Z, P, Q, R, S, T, U\}
\end{array}$$

Next consider the binding $R \mapsto f(P, Q)$ in the abstract environment $\langle F_1, R_1, S_1\rangle$. The set A of relevant sharing sets of R is $relevant(R, S_1) = \{\{R\}\}$ and B of $f(P, Q)$ is $relevant(f(P, Q), S_1) = \{\{X, Y, P\}, \{X, Y, Q\}\}$. Both R and $f(P, Q)$ are linear in the abstract environment $\langle F_1, R_1, S_1\rangle$. There is no need to compute closure under union of A and B, and $A' = A$ and $B' = B$.

$$\begin{array}{rcl}
S_2 & = & S_1 \setminus (A \cup B) \cup pairwise\_union(A, B) \\
 & = & \{\{Y\}, \{Z, Z1\}, \{S\}, \{T\}, \{U\}, \{X, Y, P, R\}, \{X, Y, Q, R\}\} \\
R_2 & = & R_1 = \{Y\} \\
F_2 & = & F_1 \setminus (\bigcup A) = \{Z, P, Q, S, T, U\}
\end{array}$$

The effect of the third binding $S \mapsto f(P, Q)$ on $\langle F_2, R_2, S_2\rangle$ is derived as follows. The set A of relevant sharing sets of S is $relevant(S, S_2) = \{\{S\}\}$, and the set B relevant sharing sets of $f(P,Q)$ is $relevant(f(P,Q),S_2) = \{\{X,Y,P,R\},\{X,Y,Q,R\}\}$. Both S and $f(P,Q)$ are linear, and hence $A' = A$ and $B' = B$.

$$\begin{aligned}
S_3 &= S_2 \setminus (A \cup B) \cup pairwise\_union(A,B) \\
&= \{\{Y\},\{Z,Z1\},\{T\},\{U\},\{X,Y,P,R,S\},\{X,Y,Q,R,S\}\} \\
R_3 &= R_2 = \{Y\} \\
F_3 &= F_2 \setminus (\bigcup A) = \{Z,P,Q,T,U\}
\end{aligned}$$

The fourth binding $Y \mapsto g(a,b)$ modifies $\langle F_3, R_3, S_3\rangle$ as follows. The set A of relevant sharing sets of Y is $relevant(Y,S_3) = \{\{Y\},\{X,Y,P,R,S\},\{X,Y,Q,R,S\}\}$, and the set B of relevant sharing sets of $g(a,b)$ is $\emptyset$. Since Y is bound to a ground term, the abstract substitution $\langle F_3, R_3, S_3\rangle$ is refined as follows.

$$\begin{aligned}
S_4 &= S_3 \setminus A = \{\{Z,Z1\},\{T\},\{U\}\} \\
R_4 &= R_3 \setminus \{Y\} = \emptyset \\
F_4 &= F_3 \setminus (\bigcup A) = \{Z,T,U\}
\end{aligned}$$

The final binding $Z \mapsto g(f(P,Q),T)$ affects $\langle F_4, R_4, S_4\rangle$ as follows. The set A of relevant sharing sets of Z is $relevant(Z,S_4) = \{\{Z,Z1\}\}$ and the set B of relevant sharing sets of $g(f(P,Q),T)$ is $relevant(g(f(P,Q),T),S_4) = \{\{T\}\}$. Once again, neither Z nor $g(f(P,Q),T)$ is non-linear, and the abstract substitution is refined as follows.

$$\begin{aligned}
S_5 &= S_4 \setminus (A \cup B) \cup pairwise\_union(A,B) \\
&= \{\{Z,Z1,T\},\{U\}\} \\
R_5 &= R_4 = \emptyset \\
F_5 &= F_4 \setminus (\bigcup A) = \{T,U\}
\end{aligned}$$

The substitution $\langle F_5, R_5, S_5\rangle$ is the result of $abstract\_unify(mgu(a,h),\langle F,R,S\rangle)$.

In this section, we defined an abstract domain for expressing freeness, sharing, and groundness properties of substitutions. We stated and proved the invariant properties of the abstract domain elements. A safe abstract unification algorithm was derived using these invariant properties. We showed that the linearity of terms is useful in treating sharing as a non-transitive relation where it was safe to do so. In the following section, we prove the safety of the abstract unification algorithm.

3.3.2 Correctness

Recall from Chapter 2 that the functions *entry_sub* and *success_sub* are required to be continuous in the arguments from the domain Asub. The following propositions establish their continuity.

Proposition 3.4

Function *refine* is continuous in its argument from the domain Asub. □

Proof

The functions used in the definition of *refine*, such as set union, set subtraction, set intersection are continuous. Function composition and application preserve continuity. Hence, *refine* is continuous. □

Proposition 3.5

> Functions *abstract_unify*, *entry_sub*, and *success_sub* are continuous in their arguments from the domain Asub. □

Proof

> Functions *abstract_unify*, *entry_sub*, and *success_sub* are continuous because they are defined in terms of *refine* and other continuous functions such as set union, and other operations such as product formation, function application, and function composition preserve continuity. □

Simply stated, the functions *refine*, *abstract_unify*, *entry_sub*, and *success_sub* are computable and all computable functions are continuous.

Recall that in the Galois insertion approach (Chapter 2), local correctness guarantees global correctness. Proposition 3.3 states the local correctness of the function *refine*. The correctness of *refine* implies the correctness of *abstract_unify*. Since there are only finite number of equations $v_i = t_i$ in the unifier of a goal and a head, a simple induction on the number of equations will prove the correctness *abstract_unify*.

3.4 Precision of the Analysis

Our abstract domain Asub has Jacobs and Langen's Sharing domain [44] as a component. In addition, our domain expresses the freeness and linearity of variables. This enables us to treat sharing as non-transitive at the cost of increased worst-case time complexity. However, the performance of our analysis is much better both in terms of precision and time taken for the analysis for a number of programs tested, as shown in Table 3.1.

We implemented a domain-independent abstract interpreter and instantiated it with our Asub domain and Jacobs and Langen's Sharing domain [44] in order to compare their precision and efficiency. The results of the analyses are listed in Table 3.1. The first column names the programs tested, and the second column lists the size of the program in number of clauses. The third and fourth list the total number of pairs of shared variables at all program points, using Asub and Sharing respectively. Note that since we derive potential sharing information, the less the number of sharing pairs, the more precise the analysis. The last two columns list the times (in seconds) for the analyses on a HP 735 machine running Sicstus Prolog 2.1.8 in byte code emulation mode.

Several programs were used to test the precision and efficiency of the two analyses. Programs ranged in size from 4 clauses to 178 clauses. Programs such as `splay` (a self-adjusting binary search tree program), `bid` (a program that computes points in a Bridge hand), `deriv` (symbolic derivation program), `read` (a Prolog parser), `rdtok` (A Prolog tokenizer), `peephole` (Stonybrook Prolog peephole optimizer), and `ann` (&-Prolog Clause annotator) are non-trivial.

Since the analyses are aimed at deriving *potentially sharing* variables, an analysis is more precise when it derives a smaller set of such variables. By this metric, our abstract domains and operations are more precise than Jacobs and Langen's. In a few programs, where there is hardly any sharing, both analyses performed equally well. As programs got larger or more complicated (as in the case of a small but complex program `splay`), our analysis consistently derived fewer sharing pairs of variables.

		No of Pairs of Shared Variables		Time in Secs	
Program	Size	Asub	Sharing	Asub	Sharing
Ackermann	4	0	0	0.005	0.004
Iso	6	0	3	0.140	0.140
Serialize	12	42	362	0.689	3.290
Grammar	15	11	11	0.080	0.080
Browse	32	131	218	0.919	1.460
Bid	51	0	11	0.460	0.440
Deriv	61	0	0	1.950	1.849
Read	87	11	26	2.640	2.500
Peephole	134	374	534	4.400	5.609
Boyer	139	1	22	1.219	1.210
Ann	178	2655	2999	9.380	8.900
Semi20	46	1110	1354	6.169	19.369
RdTok	54	54	244	21.140	22.530
Splay	22	1173	1189	20.720	71.500

Table 3.1: Comparison of Asub and Sharing Abstract Domains

Contrary to the popular belief that more precise analyses are necessarily less practical or more time consuming, the improved precision of our analysis actually contributes to faster execution. In the best case, our analysis is faster by three and a half times. This is mainly due to the computation of closure under union during abstract unification. Closure under union of a set of elements may take time exponential in the size of the set in the worst-case. Jacobs and Langen's abstract unification algorithm performs closure under union for all unifications. By maintaining and propagating linearity information, our analysis avoids closure under union most of the time. If we do not perform closure under union, our abstract unification has polynomial time complexity. Furthermore, avoiding closure under union has the added benefit of keeping the abstract substitutions small. The abstract domain and unification algorithm of Jacobs and Langen lose precision in sharing propagation and contribute to slow analysis at the same time. The analysis times for our domain and Jacobs and Langen's show some non-polynomial growth in the case of programs `splay`, `semi20`, and `rdtok`. These anomalous cases are discussed in the next chapter.

3.5 Related Work

There has been considerable amount of work in the field of abstract interpretation of logic programs in the last six or seven years. In this section, we discuss research related to the derivation of sharing, groundness and freeness of variables in a Horn clause logic program.

Jacobs and Langen; Muthukumar and Hermenegildo

Jacobs and Langen [44] proposed an abstract domain called *Sharing* to express the sharing among and independence of variables. As mentioned in the previous chapter, our work starts with their domain *Sharing*. This domain captures variable sharing precisely and thus is good at expressing groundness dependencies among variables. However, it causes the non-transitive sharing relation to be treated as transitive and hence is not good at sharing propagation.

For example, assume that in the concrete substitution at some program point, variable X is bound to f(A,B), Y to g(A,C) and at a subsequent point Z is bound to h(C,D). It is clear that variables X and Y share the subterm A and Y and Z share the subterm C. But there is no sharing between X and Z. The abstract domain *Sharing* and the corresponding abstract unify operation proposed by Jacobs and Langen will have to treat the non-transitive relation *Sharing* as transitive and thus conclude that variables X and Z may also share. Information about linearity of variables allows us to treat sharing as non-transitive and hence our abstract unification is more precise and more complicated, as shown by the examples in the previous section.

Muthukumar and Hermenegildo [61, 62] extended the abstract domains of Jacobs and Langen to account for freeness of variables but suggested different abstract domain operations. Their abstract domain operations are more precise than Jacobs and Langen's because freeness of variables allows one to treat sharing more precisely in some cases. However, the lack of information about the linearity of terms implies that they too have to treat sharing as transitive more often than we do. Specifically, when approximating a unification $t_1 = t_2$, if t_1 is known to be free, then subterms of t_2 will not be brought to share with each other because of this unification. If t_1 is not known to be free, they have to assume that subterms of t_2 may share.

Jones and Sondergaard; Codish, Dams, and Yardeni

Jones and Sondergaard [47] proposed the abstract domain $Asub_{ground}$ to derive groundness of variables and another abstract domain $Asub_{share}$ to express potential sharing among variables. Codish, Dams and Yardeni [17] used the domains of Jones and Sondergaard to formally rederive the abstract domain operations. An element of the abstract domain $Asub_{share}$ is a pair of variables which signifies possible sharing between those two variables. We compare the sharing domain of Jones and Sondergaard with the Sharing domain of Jacobs and Langen [44] using the following example:
$\theta = \{W \mapsto f(C,A), X \mapsto g(A,C), Y \mapsto B, Z \mapsto h(A,B,C)\}$.

In Jones and Sondergaard's approach, the above substitution is approximated as $\{(W,X),(W,Z),(X,Z),(B,Z)\}$ which says that W shares with X and Z, X shares with Z, and B shares with Z. The symmetric elements of the abstract substitution, namely $(X,W),(Z,W),(Z,X),(Z,B)$ are not shown. Notice that the substitution $\theta' = \{W \mapsto f(A,F), X \mapsto g(A,E), Y \mapsto f(B,G), Z \mapsto h(A,B,D)\}$ has the same abstraction. The variables W, X, and Z all share with each other and also have independent subterms in θ', but in θ, W, X, and Z share the same set of variables. This information is lost in the abstract substitution and hence we cannot conclude that binding Z to a ground term will ground W, and X too. Thus, $Asub_{share}$ does not express the *definite* sharing among variables due to unification.

The sharing domain of Jacobs and Langen [44] represents the above substitution as $\{\emptyset, \{W,X,Z\}, \{Y,Z\}\}$, where $\{W,X,Z\}$ represents the variables A, C shared by W, X, and Z, and $\{Y,Z\}$ represents the variable B shared by Y and Z. This not only represents sharing but also independence of terms as discussed in the previous sections. In particular, grounding of Z grounds W, X, and Y in the concrete domain. This is accurately captured by the abstract unification that removes the sets $\{W,X,Z\}$, and $\{Y,Z\}$ from the abstract substitution.

Jones and Sondergaard treat sharing as non-transitive, whereas Jacobs and Langen treat sharing as transitive. Hence, in some cases Jones and Sondergaard's unification is more precise than Jacobs and Langen's and less precise in others.

Our abstract domain is a product of three domains. The sharing component is the same as Jacobs and Langen's but the freeness and linearity components are additions. Since our domain has linearity information, our abstract unification is more complicated than Jacobs and Langen's but also more precise. We do not treat sharing as transitive. In particular, when a X is unified with $f(Y,Z)$, our abstract unification does not treat Y and Z as possibly sharing, unless X is potentially non-linear, *i.e.*, may have a repeating variable sub-term.

Codish, Mulkers, Bruynooghe, de la Banda, and Hermenegildo

In a recent work, Codish *et al* [18] suggest combining the domains of Jones and Sondergaard and of Jacobs and Langen. They use information from one to improve the precision and efficiency of the other. The final outcome of their analysis is comparable to ours in terms of precision and speed. The results of their analysis and our own empirical studies confirm that an increase in the expressiveness of the domains due to combining domains leads to faster analyses and more precise results. But the worst-case time complexity of abstract operations over combined domains is worse than those over their component domains. This apparent paradox is explained by the fact that worst-case scenarios seem to be contrived and are not encountered in practice. The increased precision of the analysis helps to avoid costlier portions of the abstract unification algorithm.

Debray

Predicates in a Horn clause logic program are adirectional, *i.e.,* there is no distinction between "input" and "output" arguments, since they define relations and not functions. However, researchers observed that most of the time procedures are used with a clear direction: some arguments were considered by the programmer to be input (*i.e.,* will be ground terms at the time of procedure call) and others as output (will be variables at call time and bound to ground terms on procedure exit). A compiler can generate more specific unification instructions and perform other optimizations if the input-output modality of predicates are known. This led to the provision of "directives" called mode declarations to the compiler, so that better code can be generated. Debray's proposal [29] for static derivation of modes is one of the earliest in the field, although presented in the traditional data flow analysis framework and not as an abstract interpretation scheme.

The abstract complete lattice of descriptions proposed by Debray [29] consists of five elements: the empty set of terms, set of ground terms, set of free (uninstantiated) variables, set of non-variable terms and the set of all terms. Since the domain cannot express the sharing of variables, groundness and freeness derivation is imprecise (See example 6 in [29]). However, it was not intended for deriving sharing information.

Winsborough

Winsborough [89] sketched a domain for sharing and freeness analysis. His domain keeps track of the structure of terms. A set of substitutions is represented by *generalized term graphs* (GTGs), a form of directed acyclic graphs. A node in the graph can be a program variable, constant, function symbol, predicate symbol, or the descriptions *unknown, exact,* and *ground.* Each node may have one or more incoming arcs and one or more outgoing arcs. The arcs signify variable binding, definite sharing and possible sharing, and groundness. The abstract domain operations have not been fully specified. No complexity analysis of the operations is provided. To the best of our knowledge, the domain and the operations have not been implemented. Thus, the feasibility of the proposal has not been established either analytically or empirically.

Chang; Citrin; Xia and Giloi

Chang [14] models a substitution as a function from variables to descriptions of terms. The abstract domain proposed by Chang consists of three descriptions: G for ground terms, C_i for the i^{th} equivalence class of variables that *may* share a subterm with one another and I for a variable that is not ground and is in singleton equivalence class, *i.e.,* independent. For example, the substitution $\{P \mapsto f(a,b), Q \mapsto g(X), R \mapsto h(X,Y), S \mapsto f(Z)\}$ would be abstracted to $\{P \mapsto G, Q \mapsto C_1, R \mapsto C_1, S \mapsto I\}$. It is clear that this abstract domain does not capture variable sharing information precisely. The variables in an equivalence class are equivalent with respect to possible sharing and not equivalent with respect to groundness propagation. Hence, they cannot conclude that unifying some member of an i^{th} equivalence class with a ground term makes other members of the same class ground. In the absence of precise sharing information, groundness propagation suffers.

Extensions to Chang's abstract domain by Citrin [16] and Xia and Giloi [92] alleviate the problem to some extent by maintaining information about strongly coupled sub-classes of the equivalence classes C_i. A strongly coupled sub-class of C_i is a set of variables such that grounding one variable in that sub-class grounds all others in the same sub-class. This approach still does not allow us to conclude, for example, grounding R grounds Q but not vice-versa. Our abstract domain keeps track of groundness dependencies precisely and will be able to derive that groundness of R implies groundness of Q but not vice-versa.

Marriott and Sondergaard

Marriott and Sondergaard [54] proposed a novel domain *Pos* based on positive Boolean functions to represent groundness dependencies precisely and efficiently in practice. This domain is not intended for sharing analysis.

The domain *Pos* consists of positive Boolean functions of propositional variables *i.e.,* those functions which evaluate to true iff all the variables in the function are assigned true.

The groundness property of a substitution $\theta = \{X \mapsto f(a), Y \mapsto Z\}$ may be represented using boolean functions as follows. Since variable X is ground in θ, the proposition x can represent the groundness of X. Likewise, since any further instantiation of Y which grounds Y will also ground Z and *vice versa*, proposition $y \leftrightarrow z$ can represent this groundness dependency. Thus, the formula $x \wedge (y \leftrightarrow z)$ abstracts the groundness information in θ. It is obvious that whenever two terms t_1 and t_2 are unified, the groundness dependency of the two terms can be abstracted by the formula $\wedge Vars(t_1) \leftrightarrow \wedge Vars(t_2)$.

Each binding $X_1 \mapsto T_1$ in a substitution can be abstracted by an equivalence function $X_1 \leftrightarrow \wedge Vars(T_1)$ and a set of bindings in a substitution is abstracted by the conjunction of the equivalences corresponding to bindings. A set of substitutions is represented by the disjunction of the abstraction of each of the substitutions.

A property of *Pos* that is most appealing is known as *condensation*. The term was introduced by Jacobs and Langen [44]. Marriott and Sondergaard [56] show that *Pos* also has this property. Condensation enables us to represent the abstract meaning of a procedure (a fixed-point of the associated functional) in a closed form. This implies that instead of computing the fixed-point of a procedure for each of possibly many call substitutions, we have to compute the meaning only once in a goal-independent way and specialize it for each call pattern. This meaning can be computed in the usual bottom-up manner. What is not usual is that we do not have to compute the abstract meaning of a clause for all possible entry substitutions. We need to do it only for one entry substitution induced by the call substitution *true* which says nothing is known about the groundness dependencies of the caller's arguments. The specialization of the abstract meaning of a procedure for each call to that procedure is usually very simple and much less expensive than computing fixed-points.

Suppose we want to analyze *concat* for its groundness properties. Assuming nothing about the groundness of the arguments of a call
concat(X, Y, Z), we can derive the formula $(x \wedge y) \leftrightarrow z$ which is a closed form for the abstract meaning of *concat* where x, y, z represent the three arguments of *concat* respectively. It says that X and Y are ground (or will become ground) implies Z is ground (or will become ground). Suppose that in a call *concat*(X_1, Y_1, Z_1), the third argument is ground and nothing is known about the other two. This is abstracted by the proposition z_1 which says that z_1 is ground. The conjunction of the formula z_1 and the formula $(x \wedge y) \leftrightarrow z$ which represents the meaning of *concat* and the formula $(x \leftrightarrow x_1) \wedge (y \leftrightarrow y_1) \wedge (z \leftrightarrow z_1)$ which represents unification of *concat*(X, Y, Z) and *concat*(X_1, Y_1, Z_1), yields after simplification the formula $x \wedge y \wedge z \wedge x_1 \wedge y_1 \wedge z_1$. When restricted to goal variables, we have that the first two arguments X_1 and Y_1 will also be ground after solving the given goal.

3.6 Summary

We discussed the problems of deriving precise sharing, groundness and freeness information and then defined an abstract domain *Asub* that has three components; the sharing component is the same as Jacobs and Langen's; it is augmented with two more components, one for deriving freeness of variables and another for keeping track of the linearity of variables. The linearity information helps to improve the precision and efficiency of our analysis since it allows the relation sharing to be treated as non-transitive.

The freeness component does not add to the precision or efficiency; it is another property we needed to derive. Finally, we presented related research and compared it to our work.

In the next chapter, we discuss the time complexity of our analysis, consider ways to accelerate convergence using widening operation, and present empirical evidence by analyzing a number of programs.

Chapter 4

Efficiency of the Abstract Interpretation Scheme

4.1 Introduction

In this chapter, we analyze the worst-case time complexity of our scheme for deriving groundness, freeness, and sharing of variables. Average case analysis is difficult mainly because characterization of "average programs" is both difficult and open to debate. We review some worst-case time complexity results for data flow analyses of logic programs. These results show that it is highly unlikely that any analysis that treats sharing as non-transitive can have worst-case polynominal time complexity.

An algorithm can have worst-case exponential time complexity in a number of ways. Some of them are preferable to others. One way to identify such preferences is to derive the time complexity in terms of natural parameters of the problem. The time complexity analysis is therefore based on the the number of clauses m in the program, maximum number of literals l in a clause, maximum arity a of a predicate, and maximum number of variables n in a clause.

We show that a naive bottom-up computation of the fixed-point of our analysis has time complexity $\mathcal{O}(m^3 la2^{2^n})$. In general, we need to compute only a small portion of the fixed-point. In an approach called Minimal Function Graph (MFG) Semantics to compute the fixed-points of semantic functions [46], only a subset of the fixed-point of a function is computed. Informally, the subset that is computed is the reflexive, transitive closure of the set of activations (environments or abstract call substitutions) reachable from a given (usually top-level) activation and their corresponding success substitutions. This approach leads to considerable savings in the computation of fixed-points. We show that computing the MFG semantics of our analysis has time complexity $\mathcal{O}(m^3 2^n T)$ where T stands for the cardinality of the set of reachable activations from the top-level goal. Empirical evidence suggests that T is indeed bounded by a polynomial in the number of clauses for most programs, *i.e.*, the number of entry substitutions per clause induced by a top-level goal is bounded. Thus if we assume a bound on the number of clause variables, then we have a pseudo-polynomial time algorithm (for each fixed value of n) for sharing, freeness, and groundness analysis of logic programs. Another encouraging evidence of the feasibility of our analysis is that in general the number of fixed-point iterations is bounded by a small constant for all the programs tested.

We mentioned that the size T of the set of activations reachable from a top-level goal is bounded by a polynomial in the number of clauses for most programs. This claim is made more precise now. We classify programs into two categories, those that induce "stable substitutions" (see Page 72) with respect to an abstract domain and those that do not. For the former class, the number of distinct entry substitutions per clause is bounded by a constant. Hence, the size T of the set of activations reachable from a top-level goal is linear in the number of clauses. If we make an additional assumption that the number of variables in a clause is bounded, then our analysis has pseudo-polynomial worst-case time complexity for the former class. For programs that induce "unstable call substitutions" (see Page 72), our analysis takes time bounded by a polynomial in the number of clauses and an exponential in the number of clause variables. For this class, we propose a widening operation that sacrifices precision for speed. This is done selectively and automatically on a procedure-by-procedure basis.

Cousot and Cousot proposed a *widening* operation [21] for accelerating the convergence of fixed-point computations. Widening is applied to the successive iterates of the fixed-point computation and it sacrifices precision for faster convergence. This is very useful in those cases where the abstract domain has infinite ascending chains or when the height of the abstract domain is finite but very large and the fixed-point computation takes too many iterations to converge. Since the number of fixed-point iterations needed in our analysis is fairly small, Cousot's widening operation applied to fixed-point iterates does not accelerate convergence. We propose a similar operation with nearly identical properties in the spirit of Cousot's original proposal.

To verify the performance of our proposed analysis and the widening operation, we implemented a domain independent analyzer for Horn clause logic programs and instantiated the analyzer with our domains and operations. Analysis of a number of programs show that the number of fixed-point iterations appears to be independent of the height of the abstract domain and is bounded by three in most cases, compared to the worst-case bound which is exponential in the number of clause variables. Our results are supported by similar results for analyses which use different and more or less complex domains and different fixed-point computation algorithms [51, 88].

With this brief outline, we now proceed to an analysis of the time complexity of our abstract interpretation.

4.2 Time Complexity Analysis

Before analyzing the time complexity of our abstract interpretation, we mention some results about the complexity of data flow analysis of logic programs, due to Debray [26]. The following is an informal summary of terms and results from complexity theory [41].

A decision problem (whose solution consists of either yes or no) is in NP iff it can be solved in time bounded by a polynomial in the input size by a non-deterministic Turing machine (NDTM). Likewise, a decision problem is in P iff it can be solved by a deterministic Turing machine (DTM) in polynomial time. A decision problem Π is NP-complete iff Π is in NP and all other decision problems in NP can be transformed to Π by a DTM in polynomial time. Intuitively, NP-complete problems are the the hardest problems in NP, since a deterministic polynomial time solution to any of these can be used to solve all other problems in NP in deterministic polynomial time. Hereafter, we use polynomial time to mean deterministic polynomial time. The class EXPTIME consists of the decision problems that can be solved by a DTM in time bounded by $2^{p(n)}$ when p is a polynomial in the input size n. A decision problem is EXPTIME-complete if it is in EXPTIME and all problems in EXPTIME can be transformed to this problem in polynomial time by a DTM.

The results related to sharing, groundness, and freeness analysis are as follows. Precise groundness and sharing analysis of programs free of function symbols and failure branches is EXPTIME-complete [26]. Debray [26] shows that one may obtain worst-case polynomial time algorithms by sacrificing precision. Specifically, if sharing is treated as transitive (thus introducing spurious sharing during abstract unification), or if dependencies between variables can be safely ignored (once again losing precision), then subject to some additional constraints such as each predicate having at most one abstract call and success substitutions, one can derive a data flow analysis whose worst-case time complexity is bounded by a polynomial in the size of the program. In fact, Debray gives an input-output mode analysis algorithm with these properties [29].

We improve the precision of sharing analysis (proposed by Jacobs and Langen [44]) by treating sharing as non-transitive and by maintaining additional information about linearity and freeness. Hence, the worst-case time complexity of our analysis is unlikely to be better than exponential in the input size. Keeping these results in mind, we now derive the time complexity of our analysis. Our aim is to identify the parts of our analysis which contribute to the exponential complexity. Recognition of the exponential components allows us to perform widening in such a way as to allow graceful degradation in performance. Although the number of fixed-point iterations has an exponential bound, in practice the number of iterations required is always less than a small constant (at least for the programs tested and as reported by other researchers for other complex domains [51]). In practice, the exponentiality arises from the number and size of abstract entry substitutions for programs that do not have the stable call substitutions property. We suggest methods for handling this source of inefficiency.

Outline of Complexity Analysis

Recall that the fixed-point of a function can be computed by computing the successive approximations of the Kleene sequence [76] and stopping as soon as two successive approximations are the same. The time complexity of a fixed-point computation is determined by the number of iterations needed to compute the fixed-point multiplied by the time spent in each iteration and the time for comparing two successive approximations. The number of iterations is bounded by the height of the abstract domain. This is because we start with the least element as the first approximation and in each iteration the approximation moves towards the fixed-point or stays the same when it reaches the fixed-point. In the worst-case, the top element may be the fixed-point. After deriving the worst-case upper bound, we will point out some obvious inefficiencies in the fixed-point computation process and present an approach based on the minimal function graph semantics and discuss some empirical results.

Let the number of clauses in a program be m and the maximum over all clauses of the number of variables in a clause be n. Let l be the maximum number of literals in the body of a clause and a be the maximum arity of a predicate in the program. We first estimate the upper bound on the number of iterations, followed by an analysis of the time complexity for an iteration.

4.2.1 Number of Fixed-Point Iterations

Recall that our abstract domain has three components, Sharing, Repeat and Free. The Sharing component for each clause is the powerset of the powerset of variables in that clause. The size of this component domain is 2^{2^n} and the height is 2^n. This is because the set of all sets of variables in a clause is the top element (represents the worst-possible sharing and independence of clause variables) and the set of singleton sets of variables represents the initial abstract sharing among clause variables. The components Repeat and Free are both powersets of clause variables and hence the height of these domains is n. Thus, the size of the abstract domain for a clause is $2^n 2^n 2^{2^n}$ and the height of our abstract domain for a clause is bounded by $n^2 2^n$, since the size (height) of a product domain is given by the product of the sizes (heights) of the component domains. Thus the fixed-point for each clause is reachable in at most $n^2 2^n$ steps. Since the number of clauses is m, the fixed-point for the whole program can be computed in at most $mn^2 2^n = \mathcal{O}(m2^n)$ iterations.

4.2.2 Time Complexity of an Iteration

We now derive an upper bound for the time complexity of computing each approximation. Each step in the fixed-point computation involves computing a better approximation of the abstract exit substitution corresponding to each entry substitution for each clause. The maximum number of possible entry substitutions for a clause is bounded by the number of elements in the abstract domain for a clause, which is $2^n 2^n 2^{2^n}$.

The time for computing the success substitution of a body literal, given its call substitution, is the number of clauses that unify with the goal times the sum of the time for computing entry and success substitutions for the matching clauses. The time for computing entry and success substitutions is given by the arity of the goal times the time taken for abstract unify. The costliest operation in abstract unify is closure under union, which in the worst-case may take 2^k steps where k is the number of variables in the sets being closed under union. Note that k can be equal to the number of variables in a clause and thus we take k to be n. Since a goal may potentially unify with all the clauses in the program, the time taken for finding the success substitution for one body goal is $ma2^n$. Since there may be l body goals, the time taken for updating one clause's exit substitution is $lma2^n$. There can be $2^n 2^n 2^{2^n}$ different entry substitutions and hence the time for computing one clause's approximation is $lma2^n 2^n 2^{2^{n+1}}$ which is $\mathcal{O}(lma2^{2^n})$.

Since there are m clauses in the program, computing an approximation for all the clauses has time complexity $\mathcal{O}(lm^2 a 2^{2^n})$. If we implement the Sharing, Repeat, and Free components as ordered lists, comparing two elements will take time linear in the size of the elements. The size of an abstract domain element for a clause is at most $n^2 2^n$ (number of non-linear variables times the number of free variables times the number of sharing sets in an abstract substitution). Thus, comparing two successive approximations for all clauses may take $n^2 2^n$ number of steps which is $\mathcal{O}(2^n)$. Therefore, the time taken for one iteration of the fixed-point computation in the worst-case is $\mathcal{O}(2^n) + \mathcal{O}(lm^2 a 2^{2^n})$ which is $\mathcal{O}(lm^2 a 2^{2^n})$.

Time Complexity of the Whole Analysis

Since the number of fixed-point iterations is bounded by $\mathcal{O}(m2^n)$, and each iteration can take $\mathcal{O}(lm^2 a 2^{2^n})$, the time taken for the whole computation is $\mathcal{O}(m^3 la 2^{2^n})$. If we assume that the number of literals l in a clause is smaller than the number of clauses in a program, and the maximum arity a of all predicates does not vary with program size, the above expression simplifies to $\mathcal{O}(m^3 2^{2^n})$, where m is the number of clauses and n is the maximum number of variables in a clause. Note that these two assumptions are reasonable ones. In practice, as the number of procedures increases, the number of procedure calls in each procedure stays relatively constant. Likewise, as the number of procedures increases, the number of arguments per procedure remains constant. Empirical study of Prolog programs by Touati [33] shows that the average arity of a predicate is less than five.

Discussion

From the worst-case complexity analysis, it appears at first glance that the proposed analysis is feasible only for very small programs. Worst-case time complexity analysis which considers computing the abstract meaning of a function at *all* the points in its domain is misleading. We explain this claim now. The meaning of a recursive function is given as the least fixed-point of its associated functional. But it is not necessary to compute the entire fixed-point.

The term 2^{2^n} in the time complexity refers to the number of elements in the domain of the abstract meaning of a clause, *i.e.,* the number of possible entry substitutions for a clause. In a standard semantics, it is obvious that a procedure is always used in a small number of ways (say, with some arguments as input and some output). In fact, one of the reasons for data flow analysis is to detect such regular usage and optimize code generation for such cases. If we know the possible ways in which procedure may be called (*i.e.,* if we are given the abstract descriptions corresponding to these entry points), we can compute the value of the abstract function only at these points instead of computing its value for all points in its domain. This naturally leads to the notion of *Minimal Function Graph* semantics.

4.3 Minimal Function Graph Semantics

We do not compute the meaning of a recursive function such as factorial in its entirety. Just as the meaning of factorial is computed at the desired points on an as-needed basis, the abstract meanings of clauses can be computed as and when needed. This idea was present in Cousot and Cousot's original paper [21] and was later popularized by Mycroft and Jones [46] under the name of *Minimal Function Graph* semantics.

Example 4.1

Consider the familiar concat program which concatenates two lists.

```
1.  concat([], R, R).
2.  concat([X|Y], Z, [X|W]) :-
        concat(Y, Z, W).
:- concat([a], [b,c,d], A).
```

□

The meaning of clause 2 of concat, according our abstract interpretation of Chapter 3, is a function which maps entry substitutions to exit substitutions. There are 4 variables in the clause and hence the size of our abstract domain for the clause has $4^2 \times 2^4 = 256$ elements. In the worst-case, the number of distinct entry substitutions for the clause is also 256. A naive bottom-up computation of fixed-points would compute the exit substitution corresponding to each of these entry substitutions. Clearly, this is wasteful. Suppose we know that concat is called from other parts of the program or from the top-level with a call substitution $Asub_{call}$ in which the first two arguments are always ground and the third argument is an unbound variable. Then, we need to know the exit substitution corresponding to the entry substitution $Asub_{entry}$ in which X, Y, and $\hat{Z}$ are ground, and W is free. Trying to compute the corresponding exit substitution leads to a recursive call to concat in which, not surprisingly, the first two arguments are ground and the third argument is free. We say that the set of entry substitutions (for clause 2) that are reachable from $Asub_{call}$ contains only one abstract substitution, which is $Asub_{entry}$. We say that $Asub_{entry}$ is an activation for concat reachable from $Asub_{call}$.

In other words a call to concat, with the first two arguments ground and the third argument free, gives rise to a bounded number (1) of distinct calls to concat.

This can be generalized and one can talk about the set of distinct call substitutions for each predicate in a program or module arising from a set of entry points and their call substitutions. In essence, one has to consider the reflexive, transitive closure of the activations reachable from top-level activations.

We can do a complexity analysis of an abstract interpretation scheme assuming different cardinalities for the sets of reachable activations. Most programs we tested have the bounded or stable call substitutions property. We will show that it is exactly the class of programs with stable call substitutions that have the polynomial time complexity.

Minimal Function Graph Semantics provides a formal definition of the above description of reachable sets of calls, and their results. This is in contrast to the usual denotational definition which is given independent of the entry points or top-level goal and thus describe the "whole" function. Such a semantics is sometimes referred to as the Total Function Graph (TFG) semantics in the literature although it does not signify a connection to total or partial functions. Given a TFG semantics and entry point information, it is possible to talk about the induced MFG semantics with respect to the entry points.

In logic programming and related areas, a number of algorithms have been proposed for computing the MFG semantics or something very similar conceptually. Most of these arose out of independent work in areas unrelated to data flow analysis or abstract interpretation and hence are known by different names. Notable ones are OLDT-resolution [79], extension tables and memo relations in logic programming [34], abstract AND-OR trees [11], and the magic sets based approach to query answering in deductive databases [6, 67]. Mycroft and Jones [46] developed the MFG semantics based approach to analysis of functional programs and Winsborough [90] provided the first formalization for logic programs.

4.3.1 Modified Complexity Analysis

The set of abstract call descriptions reachable from a top-level abstract description of an entry point is usually very small and its cardinality is usually independent of the size of the abstract domain. In light of the above discussion, assuming the computation of the minimal function graph semantics instead of the total function graph semantics, the worst-case time complexity will change from $\mathcal{O}(m^3 2^{2^n})$ to $\mathcal{O}(m^3 2^n T)$. The term 2^{2^n} disappears because we no longer compute the exit substitutions corresponding to all entry substitutions. The term 2^n (which was previously subsumed by the larger term 2^{2^n}) reappears signifying the complexity of abstract unification and the number of fixed-point iterations needed in the worst-case. The new term T stands for the cardinality of the set of entry substitutions of all clauses that are reachable from the call substitution for the top-level goal. In the worst-case, it could be the entire domain and we are back to where we started. However, the rationale for introducing MFG semantics is supported by empirical evidence of our own and those reported for other large domains by various researchers [88, 77, 51, 18].

We now discuss two issues related to the expected-case performance of our analysis. Consider the worst-case complexity $\mathcal{O}(m^3 2^n T)$ again. In order to guarantee a polynomial time algorithm, we have to ensure that

- the number T of abstract entry substitutions reachable from a top-level goal and its abstract call substitution are independent of the size of the program, in particular independent of the maximum number of variables n of a clause.

- the number of fixed-point iterations and the complexity of the abstract unification algorithm are both independent of the number of variables in a clause.

These two conditions are stronger than necessary. A weaker requirement is to ensure that the number of fixed-point iterations and the complexity of abstract unification both are bounded by a polynomial in the number of variables in a clause.

Meeting these two requirements leads to the notion of widening, discussed next.

4.3.2 Widening

Cousot and Cousot proposed a *widening* operation to accelerate the convergence of an analysis [21]. Widening is used to obtain a sound but probably very imprecise approximation of the program. Informally, a widening operator skips intermediate computations to a point above the least fixed-point of the function of interest.

Suppose that a predicate p is recursively defined and we need to find the success substitution corresponding to the call substitution $Asub_{call} = \{\{X\},\{Y\},\{Z\},\{P\}\{Q\},\{R\}\}$, where P, Q, R, X, Y and Z are the variables that occur in the call to p. We consider only the sharing component of our abstract domain for this example. Suppose the first approximation of the success substitution $Asub_{succ}$ for this call adds the sharing set $\{X, Y\}$ to the set $\{\{X\},\{Y\},\{Z\},\{P\}\{Q\},\{R\}\}$, *i.e.*, the first approximation is $\{\{X\},\{Y\},\{X,Y\},\{Z\},\{P\}\{Q\},\{R\}\}$. We have added one set to the sharing sets in one iteration. In the worst-case, the success substitution will have $2^6 = 64$ sharing sets (the powerset of $\{X, Y, Z, P, Q, R\}$). It may take that many fixed-point iterations to find the success substitution for one call substitution. If we notice the "trend," for a few iterations, we may safely assume all possible sharing. Clearly, this sacrifices precision because we do not know that it will take too many iterations or that the final result will be as bad as we assume.

Assuming the worst possible sharing is a widening operation. Instead of assuming all possible sharing, we may want to use some other widening operation. In order for the assumption to be sound, the widening operation must satisfy some conditions.

Recall from Chapter 1.3 that the least fixed-point of a continuous function $F : L \rightarrow L$ on a cpo $(L, \sqsubseteq)$ is the lub of the Kleene sequence of approximations or *iterates* $F^i(\perp)$, $0 \leq i$, where $F^0(x) = \perp$ and $F^{i+1}(x) = F(F^i(x))$ for all $x \in L$. Note also that $F^i(x) \sqsubseteq F^{i+1}(x)$ for all i because of the monotonicity of F and the Kleene sequence is an increasing sequence which ultimately stabilizes. Although the Kleene sequence converges, it may take an infinite number of iterations to do so. A widening operation "skips" some or all the intermediate iterates and reaches a point in the abstract domain which is above the desired lfp. In doing so, it generally causes imprecision.

A widening operation applied to two successive iterates F^i and F^{i+1} must satisfy some conditions. We now state these conditions due to Cousot and Cousot [21], define a widening operator that satisfies these conditions, and discuss the operator's usefulness.

Definition 4.1

Let L be a cpo $(L, \sqsubseteq)$. A widening operator $\bigtriangledown : L \times L \rightarrow L$ must satisfy the following conditions.

- $\forall x, y \in L.\ x \sqsubseteq x \bigtriangledown y$
- $\forall x, y \in L.\ y \sqsubseteq x \bigtriangledown y$

such that for all increasing chains $x^0 \sqsubseteq x^1 \sqsubseteq \ldots$, the increasing chain defined by $y^0 = x^0, \ldots, y^{i+1} = y^i \bigtriangledown x^{i+1}, \ldots$ is finite. □

Let X be the chain $x_0 \sqsubseteq x_1 \ldots$ and Y be the chain $y_0 \sqsubseteq y_1, \ldots$ such that $y_0 = x_0$ and $y_{i+1} = y_i \bigtriangledown x_{i+1}$ for all i. The first condition implies that $x_i \sqsubseteq y_i$ for all i and hence $\sqcup X \sqsubseteq \sqcup Y$. This in turn implies that if X is the chain of Kleene iterates of a function F, then $lfpF = \sqcup X \sqsubseteq \sqcup Y$. Thus the lub of the chain Y obtained by widening the chain X safely approximates the least fixed-point of F. The condition that Y must be finite ensures that Y is finitely computable. Thus a widening operator can be used to ensure termination in domains with strictly increasing infinite chains.

Proposed Widening operation: We propose to use the lub operation on the abstract domain as a widening operator. In our abstract interpretation, the abstract domain *Asub* is a complete lattice and when restricted to program variables, it is finite. This means all chains in our abstract domain (restricted to program variables) are finite and this trivially satisfies the finite chain condition of a widening operation. The other two conditions, namely $\forall x, y \in L.\ x \sqsubseteq x \bigtriangledown y$ and $\forall x, y \in L.\ y \sqsubseteq x \bigtriangledown y$ are also satisfied by the lub operation because $\forall x, y \in L.\ x \sqsubseteq x \sqcup y$ and $\forall x, y \in L.\ y \sqsubseteq x \sqcup y$ by definition of lub. Therefore, the lub operation on *Asub* qualifies as a widening operator. We now discuss where we need to use the widening operator.

4.3.3 Stable versus Unstable Call Substitutions

As mentioned before, in order to realize a polynomial-time analysis, we must ensure that the number of fixed-point iterations and the complexity of abstract unification are independent of the number of variables n in a clause or at least bounded by a polynomial in n. The same condition must hold for the number of entry substitutions of a clause.

In theory, the number of fixed-point iterations for our analysis may be as large as 2^{2^n} where n is the maximum number of clause variables. The maximum number of clause variables in the programs tested ranges from five to nineteen and 2^{2^5} and $2^{2^{19}}$ are very large numbers. Cousots' widening operation appears to be useful in our analysis, at least in theory. However, in practice, the number of fixed-point iterations turns out to be independent of the maximum number of clause variables and is bounded by four. Hence, the application of widening as proposed by the Cousots is not useful for our analysis.

In order to realize a polynomial-time analysis, we must meet the other two conditions, *i.e.,* establish constant or polynomial bounds on the number of entry substitutions and the complexity of abstract unification. We discuss two classes of programs, one in which these bounds are naturally met by our analysis, and another in which the bounds are not met. We specify a widening operation for the latter class. Note that the proposed widening is not applied to the successive fixed-point iterations. This the novelty of our approach.

Procedures can be classified into two categories. In one category, called "stable call substitutions," the number of calls to a recursive predicate with distinct abstract call substitutions reachable from an initial top-level goal and its call substitution is bounded by a constant. This allows us to amortize/share the cost of fixed-point computations among several calls. Most of the programs that we analyzed exhibit this behavior, which is expected to some extent.

Whether "stable call substitutions" are induced or not depends on the program being analyzed and the granularity of the abstract domain. The finer the abstract domain, the fewer procedures will have the stable call substitutions property. This is because, as abstract domains become more and more fine-grained, each abstract domain element abstracts more and more sets of properties, and the set of substitutions that have these properties becomes smaller and smaller. In the limit, an abstract domain may be as fine-grained and therefore as precise as the concrete domain, and the number of call substitutions for a predicate p induced by an initial call to p may be infinite.

The second category of programs have the "unstable call substitutions" property with respect to a given abstract domain. In these programs, the initial call to a recursive predicate and its call substitution induce two types of inefficient behavior. The first is that the number of reachable calls to the same predicate (with distinct call substitutions) induced by the initial call substitution is not bounded by a constant as in the first category. We call this "exploring the abstract space" because an initial call to a recursive predicate leads to a call to the same predicate with a slightly different call substitution, which in turn induces another slightly different call substitution and so on. Sometimes the induced call substitutions are related by the abstract domain ordering and at other times they are not comparable. The number of iterations to reach a fixed-point is still a small number for each of these call substitutions. It is just that we have to compute the success substitutions corresponding to too many call substitutions for a goal, *i.e.,* the minimal function graph is not small. Since each fixed-point computation takes only a few (less than four) iterations, we cannot expect the widening operation as applied to the successive iterations of a single fixed-point computation to reduce the large number of independent fixed-point computations; indeed it does not. We must find a way to reduce the number of independent fixed-point iterations caused by the large number of call substitutions for a goal.

The second inefficiency is the time taken for abstract unification. The number of sharing sets of variables during abstract unification reaches close to 2^n where n is the number of variables in the head and the call. This is again related to exploring the abstract space. An initial call substitution of a goal g has a few sharing sets and non-linear variables. But recursive calls gradually increase the number of sharing sets and the non-linear variables. For the second category of programs, *i.e.,* those with the unstable call substitutions property, the time complexity of our analysis will not be a polynomial in both the number of clauses and the number of clause variables. We would like to improve the complexity in this case too and so we turn to a novel application of widening.

4.3.4 Proposed Widening Operation

The number of fixed-point iterations needed, to compute the success substitution of a goal corresponding to a call substitution, remains small for both categories of programs. Thus we cannot use widening of the iterates to speed up convergence. But we can reduce the size of reachable sets of call substitutions by widening. This is the approach we take.

Let k be a bound on the number of distinct entry substitutions for a clause. Recall that the abstract domain operations are required to be monotonic in their argument from the domain *Asub*. We can use this property to define a safe widening operation. Let E be the set of entry substitutions that we have seen so far for a clause and let e be a new entry substitution whose exit substitution we need to compute. This may lead to yet another entry substitution which is different from all others in E. If the size of E exceeds the predetermined bound k, take the lub of all the entry substitutions $E \cup \{e\}$ and use it instead of e.

We may even use soft and hard bounds on the number of entry substitutions for a clause and once the hard bound is reached, we can widen all the new entry substitutions e to the top element of the abstract domain and return it as the exit substitution. We may need to use it since the height of the abstract domain 2^n is still too big. This is tantamount to treating sharing as transitive.

This approach has many advantages as compared to treating sharing as transitive *a priori*. Our approach has the graceful degradation property in that the only procedures that lose precision are those that have the unstable substitutions property and those that call them either directly or indirectly. In terms of the call graph for a program, the procedures on the path from root to the unstable procedures will lose precision. Thus the use of widening when the analysis becomes expensive adapts the analyzer automatically to the program without penalizing programs with the stable substitutions property. Even then, not all the procedures in a program are affected by widening as mentioned before. Since the widening operation is similar to treating sharing as transitive, we are not losing precision any more than other approaches that treat sharing as transitive from the outset and lose precision for all programs in the name of efficiency.

The second efficiency measure is equally important. Here we use a compact representation to avoid exponential computations. Let $x_1, \ldots, x_n$ be the set of variables in a clause and let the sharing component of an entry substitution be the set of some subsets of $\{x_1, \ldots, x_n\}$. Notice that when we widen an abstract substitution to the top element, the sharing component represents all possible sharing and independence of the variables $x_1, \ldots, x_n$. This is realized by taking the closure under union of the set of singleton sets $\{\{x_1\}, \ldots, \{x_n\}\}$. The closure under union causes the exponential complexity of the abstract unification algorithm. So, we cannot naively perform this operation. Instead we resort to a compact representation. We simply use the set $\{x_1, \ldots, x_n\}$ together with a tag "maximal" to represent the power set. Any sharing set which is unioned with a subset of a maximal set (which represents the powerset) becomes maximal, and subsets of a maximal set are not retained.

4.3.5 Empirical Measurements and Analysis

In this section, we discuss the results on an empirical study of about thirty programs. Table 4.1 summarizes the number of entry substitutions per clause (# ESPC), the maximum number of entry substitutions (Max ESPC), the number of clauses (# Clauses), and the maximum number of variables in a clause (Max CV). The last column is the time taken for analysis on an HP 9000/735 workstation under Sicstus Prolog 2.1.8 in bytecode emulation mode.

The tested programs include small programs like ackermann's function, two versions of permutation, naive reverse, non-deterministic merge, quicksort (with and without the use of difference lists), mergesort, map coloring, n-queens etc. Other programs like `bestpath` (finds best paths from a single node to all other nodes in a graph, by E. Tick) `peephole` (a peephole optimizer for Stonybrook Prolog), ann (a clause annotator for &-Prolog by Hermenegildo, Warren and Muthukumar), `read` (public domain parser for Prolog by D. H. D Warren and R. A. O'Keefe), `rdtok` (tokenizer for Prolog by D. H. D Warren and R. A. O'Keefe), `press1` (part of a symbolic algebra package), `cs` (a cutting stock program by Dincbas and others [36]), `boyer` (a small theorem prover by E. Tick), `semigroup` (computes the elements in a semigroup, by R. Overbeek), `kalah` (a game playing program from [75]), and `splay` (a self-adjusting binary search tree by V. Saraswat) are non-trivial. These program are written by different people and represent different coding styles, and they all perform symbolic computations.

The maximum number of distinct entry substitutions per clause (reachable from the top-level goal and its call substitution) ranges from one to four in many cases. Notice that the number of all possible entry substitutions is 2^n where n is the maximum number of head variables. In these cases where the number of entry substitutions for all clauses is bound by a constant, there is no need for a widening operation. This is reflected by the fact that the times for analysis shows a polynomial growth in the size of the programs, except for `rdtok, splay, kalah,` and `press1`. The analyzer currently does not process mutually recursive procedures in an efficient way (these form a strongly connected component in the call graph of the program). This affects the analysis times to some extent for `splay, rdtok,` and `press1` which have mutually recursive procedures. In addition, the number of entry substitutions per clause is not bounded by a constant for `splay, kalah` and `press1`. For these programs, the table shows the number of entry substitutions per clause *after* the widening operation has been used. Without the widening operation, the analysis of these programs takes an inordinate amount of time. The widening operation does introduce some spurious sharing (since in effect it treats sharing as transitive) with the benefit of faster convergence.

4.4 Summary

Discovering that some problem is computationally intractable is only the beginning of trying to find an efficient solution. One must still devise ways to solve the problem as efficiently as possible. A combination of complexity analysis and empirical study is used to identify precisely the sources of inefficiency. Once these are identified, a novel use of widening provides a solution that adapts automatically to the type of programs analyzed.

Program	# ESPC	Max ESPC	# Clauses	Max CV	Time (sec)
ackermann	1	1	4	7	0.005
perm	1	1	4	6	0.005
perm1	1	2	4	7	0.005
naiv-rev	1	1	4	5	0.003
ndmerge	1	1	4	6	0.006
quicksort-dl	1.7	4	6	8	0.190
quicksort	1	1	7	8	0.100
mergesort	1	1	9	9	0.110
n-queens	1	1	9	6	0.080
iso	1	1	11	6	0.140
iso1	1	1	11	10	0.390
serialize	1.1	2	12	8	0.680
color	1	1	13	6	0.189
grammar	1	1	15	7	0.090
path	1.1	2	18	5	0.200
pg	1	1	18	12	0.210
plan	1.1	2	29	7	0.330
dcg1	1.2	2	31	8	0.370
browse	1	1	32	13	0.600
bestpath	1.2	2	33	19	1.000
gabriel	1.6	7	45	13	2.040
semigroup	2.3	4	46	12	6.210
bid	1	2	52	8	0.469
deriv	1	1	61	7	1.969
disj	1	1	63	13	0.680
cs	1	3	64	19	0.880
read	1	1	87	14	2.640
kalah	2.6	7	88	13	5.200
peephole	1.1	4	134	9	5.430
ann	1.5	4	178	17	9.420
boyer	1	1	139	9	1.250
press1	3.2	10	158	10	18.920
splay	6	12	22	13	20.600
rdtok	1.6	2	54	8	21.090

Table 4.1: Program Statistics and Analysis Times

In the case of sharing analysis of logic programs, one approach to a tractable analysis is to treat sharing as transitive. This loses precision considerably. Our approach starts with a domain where sharing is treated as transitive. We then add information about linearity and freeness of terms, and derive a non-transitive sharing and freeness propagation algorithm that has polynomial time complexity for a class of programs that induce what we call "stable call substitutions." We point out the sources of combinatorial explosion in the case of programs with unstable substitutions, and specify a widening operation and a compact representation. The result is graceful degradation, automatic and selective loss of precision only where needed, and reasonable running times.

A combination of widening, and a precise domain and operations is a better alternative to choosing coarser domains and imprecise operations. The aim is to balance the concern for worst-case time complexity, and the need for improved precision in the expected case.

Chapter 5

Simultaneous Data flow and Control flow Analysis

5.1 Introduction

In this chapter, we discuss the problem of deriving control flow and data flow simultaneously. The derivation of data flow assuming a given control flow can be done using well-known frameworks. Until now, data flow analyses worked under the assumption that control flow, *i.e.*, the partial order in which body goals are solved, is known. Such an analysis works well with any given control flow. Analogously, control flow analysis can be performed if we know the data flow. The essential idea in performing control flow analysis assuming data flow information is to start with an empty partial order of the subgoals of a clause, and add the goals to the partial order as and when the goals satisfy the given data flow constraints. This is illustrated in detail in the section following the introduction. The data flow information may be given to the control flow analyzer in the form of permissible or prohibited modes for procedures.

A permissible mode for a procedure p places some restrictions on the instantiation state of the arguments of a call to p. Informally, a permissible mode for a procedure requires one or more arguments to be input and zero or more arguments to be output such that it induces a partial ordering of body goals which satisfies the input/output constraints of body goals.

Input and output modes may be defined in a number of ways. We assume the following simple definition of modes for exemplifying our framework for combined data flow and control flow analysis. If an argument i of a procedure p is input in a permissible mode $mode_p$, any call to p which satisfies that mode must have its i^{th} argument bound to a ground term. Likewise, if an argument i of a procedure p is output according to a permissible mode $mode_p$, then the j^{th} argument must be bound to a ground term when a call to p (which satisfies mode $mode_p$) succeeds. In short, input annotation denotes "ground on call" and output annotation denotes "ground on success."

The above definition of input/output modes limit their usefulness. However, one may use a more refined definition of modes and types, such as those suggested by Bruynooghe and Janssens [12], Bronsard, Lakshman, and Reddy [10], and Somogyi [74]. The proposed framework is parameterizable in many ways and these are discussed later. The goal of this chapter is to define a framework for combined data flow and control flow analysis and not to define yet another precise mode analysis assuming control flow. Hence, we choose a simple definition of input/output modes to instantiate the combined data flow and control flow analysis framework. The implementation also uses the same definition of modes, which is again a limitation of the implementation and not of the framework. Relaxing the interpretation of input and output is possible and discussed later in this chapter.

Most of the research in data flow analysis of logic programs has concentrated on relaxing a pre-determined total ordering, such as left-to-right, based on information from a data flow analysis which assumes control flow. Likewise, research on deriving control flow [27] assumed that permissible modes of predicates are known. We propose a framework for deriving data flow and control flow simultaneously. The cyclic dependency between data flow and control flow is broken by first defining the notion of permissible modes in a way that does not depend on a fixed control flow and then deriving control flow based on permissible modes.

The simultaneous derivation of data flow and control flow is accomplished in two stages. The first stage derives permissible modes for all procedures. A permissible mode may induce more than one partial ordering of body goals. The second stage, then, finds a partial ordering of body goals, given the permissible modes of all predicates.

The derivation of permissible modes may be summarized as follows. The permissibility of a mode will be defined recursively. Some initial constraints on permissible modes are defined non-recursively and lead to the notion of partial permissible modes. A partial permissible mode is refined successively until it becomes a permissible mode. The refinement of a partial permissible mode involves incrementally constructing a partial ordering of body goals which respects the body goals's permissible mode constraints. Permissible mode derivation, in the case of mutually recursive procedures, involves fixed-point computations.

The partial ordering of the body goals derived during the first stage may be thought of as a constructive proof of well-modedness of a clause. A clause is well-moded iff there is partial ordering of the body goals which satisfies their input output constraints.

It may seem that the derivation of permissible modes, a form of data flow analysis, precedes control flow, and hence depends on control flow. A closer look will reveal that it is not so. The definition of a permissible mode (given later) posits the existence of a partial ordering, but does not depend on a fixed control flow. Furthermore, during the derivation of permissible modes the only information we need from data flow analysis is of the following form: "If goal g with the call substitution θ succeeds, what will be the success substitution?" It does not matter *how* the goal g is solved; in particular, if $g\theta$ unifies with the head of a clause c, the order in which the body goals of c are solved does not affect the success substitution of g. This property is known as the independence of the computation rule (Chapter 2).

We now address the question of the number of permissible modes derived for a procedure and their completeness. An ideal algorithm would derive a complete set of permissible modes for a procedure in polynomial time. However, there can be exponential number of permissible modes for a procedure. It is inefficient to compute all of them when at most a few may suffice. One way to reduce the set of permissible modes is to reduce the redundancy; the permissibility of some modes will be implied by the permissibility of some other mode. Then it is sufficient to compute the least restrictive or minimal modes. The notion of least restrictiveness or minimality is defined using a partial ordering on the space of modes. Unfortunately, deriving the set of minimal permissible modes is NP-Hard.

Our result on the complexity of the derivation of minimal permissible modes is the first formal result on the tractability of deriving minimal permissible modes. Note that this result holds even though we use a simple definition of modes; more refined definitions based on a system of types will not make the time complexity of permissible mode derivation algorithms any better.

We suggest the following approach to the use of permissible modes in a compiler for a logic programming language. The programmer may omit permissible mode declarations most of the time and let the compiler derive them automatically. If the compiler fails to derive a permissible mode for a procedure or fails to derive the desired permissible mode, then the programmer can add the missing ones.

The empirical results and an analysis of the time complexity of the permissible mode derivation algorithm show this approach to be tractable and useful. Furthermore, the analysis of a number of programs show that the algorithm does derive all minimal permissible modes for many procedures. It fails to derive even one permissible mode for 3 procedures out of a total of about 270 procedures, and these are due to the simple definition of modes we use for experimentation.

We give an approximation algorithm for deriving permissible modes for a clause and an algorithm for combining the permissible modes of clauses into permissible modes for a procedure. The algorithm for deriving permissible modes of a clause has polynomial time complexity. The algorithm for deriving permissible modes for a procedure by combining the modes of clauses has worst-case exponential time complexity, if the number of permissible modes of procedures is unbounded. Since the permissible modes of a clause is not independent of those of other clauses, in practice there is no combinatorial explosion when combining the modes of different clauses. If we assume a bound on the number of permissible modes of a procedure, then our mode intersection algorithm has polynomial time complexity. Analysis of a number of programs support the assumption of bounded permissible modes.

5.1.1 Inter-dependence of Data Flow and Control Flow

Data flow and control flow analyses are mutually dependent. We assumed so far that body goals are to be executed in some fixed order and then analyzed the data flow induced by the given control flow. Whenever two body goals i, $i+1$ are not data dependent on each other, *i.e.*, do not share unbound variables, their executions do not affect each other. Thus we may even execute them in parallel. The idea is the same as in parallelizing sequential Fortran programs [91]. Hermenegildo and Rossi [42] establish correctness of independent-and parallel execution with respect to standard and-sequential execution.

Assuming a fixed control flow for each clause is too rigid. One may want to solve the body goals in different order depending on the bindings for the variables in the head of a clause. We revisit the *perm* example from Chapter 1.

Example 5.1

Consider the following program *perm* which defines the relation permutation between two lists. An empty list `[]` is a permutation of itself. Given a list `[X|Xs]` of terms, its permutation is the list `[Y|Ys]` if there exists a list `Zs` and an element `Y` of `[X|Xs]` such that `Zs` has the same elements as `[X|Xs]` but without `Y` and the list `Ys` is a permutation of `Zs`. This a declarative reading of the relation defined by the predicate *perm*. □

```
1.  perm([], []).
2.  perm([X|Xs], [Y|Ys]) :-
        delete([X|Xs], Y, Zs),
        perm(Zs, Ys).
```

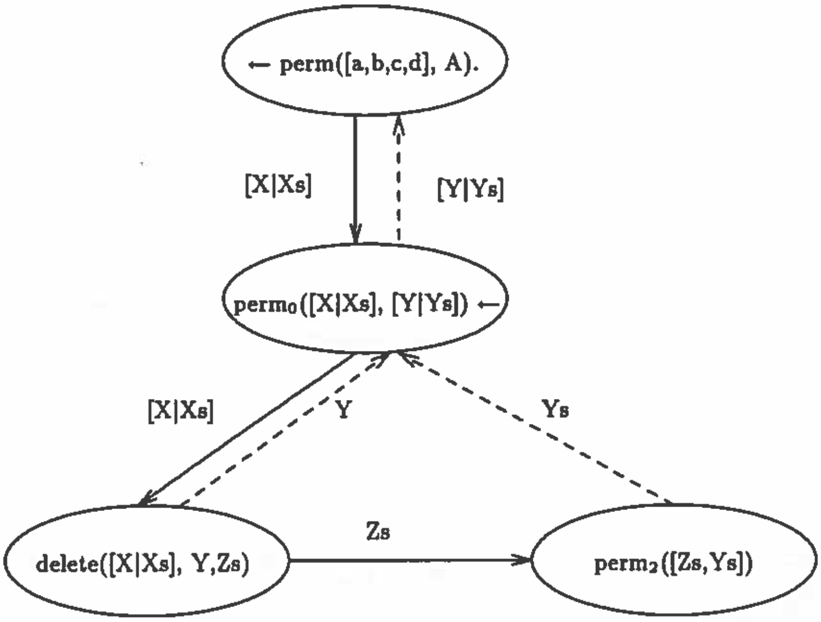

Figure 5.1: Data Flow Graph for Perm Induced by Left-to-Right Control Flow

```
3.  delete([A|Rest], A, Rest).
4.  delete([A,B|Bs], X, [A|Rest]) :-
        delete([B|Bs], X, Rest).
```

Assume that body goals are executed left-to-right (*i.e.,* in the textual order). If *perm* is called with the first argument bound to a list of terms and the second argument a free variable, it will succeed and bind the second argument to a permutation of the first. For example, the goal perm([a,b,c,d],A) will succeed and bind A to [a,c,d,b] as one possible answer substitution. Although *perm* defines a relation, we consider the term [a,b,c,d] as *input* to the procedure and A as the *output*.

The data flow induced by the the left-to-right control flow and the initial goal perm([a,b,c,d],A) is shown in Figure 5.1. We need to define some terminology to explain the data flow graph. Unifying the goal perm([a,b,c,d],A) with the head of the second clause of *perm* binds the head variables X and Xs to the terms a and [b,c,d] respectively. We say the the head literal *generates* or *produces* X and Xs. After head unification, solving the first body goal delete([a,b,c,d],Y,Zs) binds Y to a and Zs to [b,c,d]. We say that *delete* generates Y and Zs; Y is *consumed* by the head literal and Zs by the second body goal *perm*. Finally, solving the second body goal perm([b,c,d],Ys) generates Ys which is consumed by the head literal.

In a data flow graph, each node is a literal and there is directed edge from a node i to another node j labeled with X iff literal i produces a binding for the variable X which is consumed by j. We have two types of edges, solid and dashed, in Figure 5.1. They both denote the producer-consumer relationship mentioned above. Dashed edges signify that the

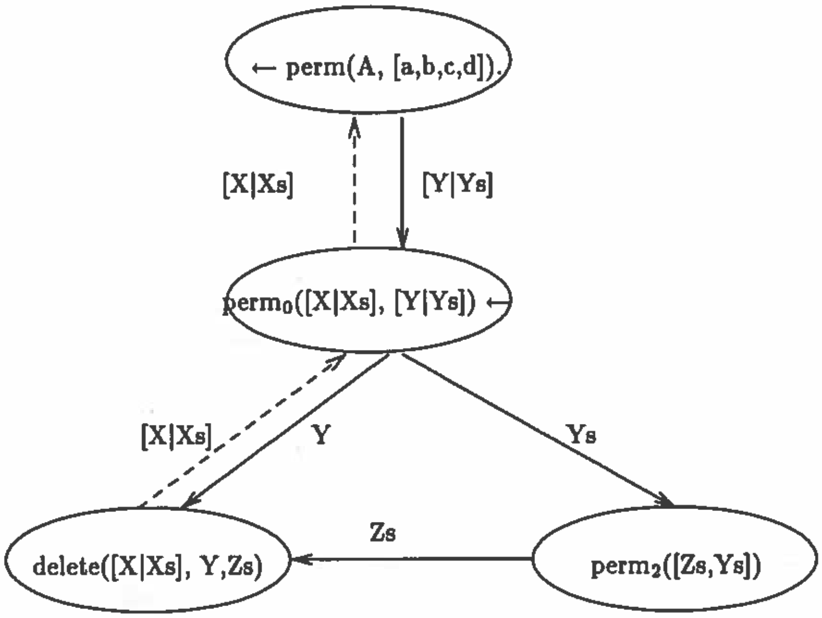

Figure 5.2: Data Flow Graph for Perm Induced by Right-to-Left Control Flow

variable is produced by a body goal and consumed by the head or passed by the head literal to the caller's environment. They provide a visual clue about the directionality of data flow into and out of a clause.

Although no input-output directionality is implied by the declarative reading, from an operational viewpoint we say that the first argument of *perm* has *input mode* and the second argument has *output mode*. Thus the declarative reading is adirectional but the operational reading is directional. The mode *perm*(+,−) indicates that the first argument of *perm* is input to the procedure and the second argument is output. Notice that, with the left-to-right control flow, the mode *perm*(+,−) induces the mode *delete*(+,−,−) for *delete* and *perm*(+,−) for the recursive call in the body of *perm* which is consistent with the mode for the initial call to perm.

Since permutation is a symmetric relation, one may be tempted to call *perm* with the second argument bound to a list of terms and leave the first one free. The expectation is that *perm* will bind the first argument to a permutation of its second argument. Unfortunately, if we assume the same control flow as in the first case, after *perm* produces the first solution, *delete* will go into an infinite loop. This is because solving the body goals of *perm* in left-to-right order causes *delete* to be called with its first and third arguments being unbound (free) variables. Arguments one and three of the recursive call in the body of *delete* are more general[1] than the corresponding head arguments; in fact, they are subterms of the head arguments. Therefore, for *delete* to terminate, either the first or the third argument must be bound to a complete list, *i.e.*, a list whose tail is not a variable.[2] In this case, the

[1] A term t_1 is more general then another term t_2 if there exists a substitution θ such that $t_2 \equiv t_1\theta$.

[2] The length of a complete list remains fixed regardless of further instantiations of it. For example,

desired order for solving the body goals of *perm* is the right-to-left order.

Using the right-to-left control flow, an initial call to *perm* with the mode *perm*(−,+) induces the same mode for the recursive call to *perm* and the mode *delete*(−,+,+) for *delete*. The data flow for the second clause of *perm* induced by the right-to-left control flow is shown in Figure 5.2.

The unsolvability of the halting problem implies that in general there is no algorithm for choosing the right order to guarantee that (logic) programs terminate. Hence, we will not make termination of programs an objective of our control flow analysis. Instead, the objective is to derive a control flow that induces well-modedness for a clause. Recall that a clause is well-moded iff there is partial ordering of the body goals such that for every body goal i, each variable in its input position either occurs in an input position of the head literal or in the output position of one of its predecessors in the partial ordering. How do we know what the input and output positions of a body goal are or should be? We introduce the notion of permissible modes which places some reasonable restrictions on the modality of the arguments of predicates.

Termination proofs of logic programs use the directionality of programs in an essential way [70] and assume that clauses are well-moded with respect to the given partial ordering. Termination proofs show that a well-founded ordering exists on the set of recursive calls to a procedure.[3]

In order to show the existence of a well-founded ordering on the set of recursive calls to a predicate arising from an initial call, one may derive a norm for the literals such that the norm of the head is greater than the norms of the body literals for all clauses in a procedure. A norm, for example, may be the "size" of a term or the sum of the sizes of two or more terms. The idea is to show termination based on the decrease in the norms of recursive calls. It is well known [70] that only a subset of the argument positions, the input positions, is useful for deriving norms. If one considers all argument positions, *i.e.*, including the output positions for a norm, then the norm of recursive calls do not decrease in general.

The above discussion of termination proofs relying on the directionality of predicates is informal. The main point of the discussion is that the well-modedness of clauses is a pre-requisite for termination proofs. Our objective is to derive permissible modes which induce well-modedness. In many cases, the partial ordering used in showing well-modedness, together with the permissible modes used in deriving the partial ordering induce well-foundedness as well and therefore guarantee termination. However, we cannot establish this in the general case.

5.1.2 Outline of Simultaneous Data Flow and Control Flow Analysis Framework

We provide a brief outline the proposed framework for combined data flow and control flow analysis. The framework has the following parameters and an instance of the framework

`[X,Y,Z]` is a complete list whose length is three and any further instantiations of `X`, `Y`, and `Z` do not change the length of the list. On the other hand, a list such as `[X,Y|Z]` is incomplete. Further instantiations may bind `Z` to any term, including lists of different lengths.

[3] A well-founded ordering on a set S is an irreflexive and transitive relation such as the $<$ relation on natural numbers, which implies that there are no infinite descending chains in S.

must define these parameters. These are:

Input, Output Modes The input output modes of a procedure must be a complete lattice. The complete lattice requirement enables us to combine modes of different clauses of a procedure.

Permissible Modes The set of permissible modes for a procedure is a subset of the set of modes for that procedure. The definition of permissible modes may be recursive, but some non-recursive initial constraints on the modes must also be specified. These initial constraints are used to define partial permissible modes and as a starting point for permissible mode derivation.

Given the permissible modes for the clauses of a procedure p, one must be able to derive the permissible modes for p itself. The *mode intersection* operation which combines the modes of different clauses is implicitly specified by the partial ordering on the space of modes.

Data Flow Analysis A data flow analysis which is invariant under different control flows must also be given. The permissible mode derivation algorithm currently uses the sharing and groundness analysis of Chapter 3. One may also a groundness analysis based on the domain *Pos* proposed by Marriott and Sondergaard [54].

The rest of this chapter is organized as follows. First we define modes, partial permissible modes, permissible modes, mode intersection, and minimal modes. Next we prove that the problem of deriving all minimal permissible modes is NP-hard. It is followed by an approximation algorithm for deriving permissible modes. The next section derives permissible modes of some example programs and is followed by proofs of correctness and termination of our algorithm. A time complexity analysis of our algorithm, and an empirical study establish the practicality of the proposal. We then discuss relaxing some parameters of the framework, especially the definition of input, output modes. The second stage of the framework, namely, deriving control flow based on permissible modes is examined. This chapter concludes with a discussion of related research and a summary.

5.2 Definitions

The definitions of modés and well-modedness are non-standard mainly for the following reason. In data flow analyses which assume a fixed control flow, a partial ordering of body goals is constrained to be consistent with the given control flow. For example, when left-to-right execution of body goals is assumed, the partial ordering is constrained to be consistent with the textual ordering. We are deriving control flow and hence do not have such restrictions.

5.2.1 Modes and Well-modedness

Definition 5.1

Let p be an n-ary predicate. A $mode_p$ for p is a function from $\{1, \ldots, n\}$ to the set $\{+, -, \bot, \top\}$. For any i, $1 \leq i \leq n$, if $mode_p(i) = +$, we say that the i^{th}

argument of p is in input mode. Likewise, if $mode_p(i) = -$, we say that the i^{th} argument of p is in output mode. $\bot$ (sometimes denoted ?) stands for an unknown mode and $\top$ denotes conflicting mode requirements. A mode $mode_p$ for a p is a *consistent* mode iff $\forall i$, $1 \leq i \leq n$, $mode_p(i) \neq \top$. A mode $mode_p$ for a p is *inconsistent* otherwise. □

The need for top element $\top$ will become clear shortly. We will write a mode $mode_p$ for an n-ary predicate p explicitly as $p(m_1, \ldots, m_n)$ where each m_i is $+$, $-$, or $\bot$ if $mode_p(i)$ $= +$, $-$ or $\bot$ respectively. For example, a mode $\{1 \mapsto +, 2 \mapsto -\}$ for *perm* will be written as $perm(+, -)$. A predicate may have more than one mode. The mode of an argument position of a predicate is referred to as a mode annotation.

It is useful to order the space of modes for a predicate. Mode annotations of argument positions impose some conditions on the call substitutions for a goal. Intuitively, a $\bot$ mode requirement for an argument position is the weakest, since it imposes no condition at all. An input mode annotation for an argument position is stronger than $\bot$ for that position, since the corresponding goal argument is expected to be a ground term. Likewise, an output mode annotation is a stronger than $\bot$ because the argument is expected to be bound to a ground term on success. The input and output mode annotations for an argument position are neither stronger nor weaker than each other and hence they are not comparable. Finally, we need a top element $\top$ which signifies conflicting mode annotation for an argument position *i.e.*, an argument position is required to be both input and output. Its sole use is to have a well-defined lub operation on the set of all modes for a clause (procedure). In the following sections, we will derive permissible modes for a clause and then "combine" the clause modes to yield permissible modes for a procedure. If a particular combination of permissible modes for clauses yields a mode that maps any of the argument positions to $\top$, that combination will be discarded. For example, if $p(+, -, +)$ is a permissible mode for a clause c of a procedure p and $p(+, +, -)$ is a permissible mode for another clause c' of p, then the combination of these modes (called mode intersection, to be defined shortly), is $p(+, \top, \top)$. The mode $p(+, -, +)$ of clause c requires argument two to be output whereas the mode $p(+, +, -)$ requires the same argument to be input. The conflicting requirements are denoted by $\top$.

By a slight abuse of notation, we use $\top$ to denote both an inconsistent modality for an argument of a predicate, and an inconsistent mode (which maps one or more of the arguments of a predicate to $\top$). Thus the set of all inconsistent modes of a predicate will be denoted by $\{\top\}$.

Since a mode for an n-ary predicate p is a function defined on $1, \ldots, n$, we use the usual ordering on functions.

Definition 5.2

The mode annotations $+, -, \bot, \top$ are partially ordered as follows: $\forall x, y \in \{+, -, \bot, \top\} . x \sqsubseteq y$ *iff* $x = \bot \ \vee \ y = \top$.

Since modes are functions, the partial ordering on the set M of modes of an n-ary predicate p is given by the usual pointwise ordering and the lub operation by the pointwise lub.

Let M be the set of modes of an n-ary predicate p. Then, $\forall mode_x, mode_y \in M$. $mode_x \sqsubseteq mode_y$ iff for $1 \leq i \leq n, mode_x(i) \sqsubseteq mode_y(i)$. □

Example 5.2

Let $p(+,+,-)$, $p(\bot,+,\bot)$ be two modes of a clause c. Then $p(\bot,+,\bot) \sqsubseteq p(+,+,-)$ and $p(\bot,+,\bot) \sqcap p(+,+,-) = p(+,+,-)$. On the other hand, the modes $p(+,+,-)$ and $p(-,\bot,-)$ are not comparable and the lub is the inconsistent mode $p(\top,+,-)$. □

Definition 5.3

For a mode $mode_p$, the set of its input positions $input\_pos(mode_p)$ is defined as $\{i \mid mode_p(i) = +\}$. The set of output positions $output\_pos(mode_p)$ is defined analogously. □

Example 5.3

Let $delete(+,-,-)$ be a mode of procedure *delete* of example 5.1. Then the set of input positions of $delete(+,-,-)$ is $\{1\}$ and its set of output positions is $\{2,3\}$. □

Definition 5.4

A call (or a goal) $p(t_1,\ldots,t_n)$ is *compatible* with or *satisfies* a mode $mode_p$ iff the set of argument positions of $p(t_1,\ldots,t_n)$ that are ground terms is a superset of the input positions of mode $mode_p$. □

Example 5.4

The goal `delete([a,b,c,d],Y,Zs)` is compatible with the mode $delete(+,-,-)$ which requires that the first argument of the call to *delete* be bound to a ground term. □

Given a mode for a predicate p and a call to p, one can identify the variables that occur in the input (output) positions of the call.

Definition 5.5

Let l be a literal and $mode_l$ be a mode of l. We define $invars(l, mode_l)$ to be the set of variables of l that occur in the input positions $input\_pos(mode_l)$. The set of variables appearing in the output positions of l, $outvars(l, mode_l)$ is defined analogously. □

Example 5.5

Let *delete*([a,b,c], Y, Z) be a literal and $mode_{del} = delete(+,-,-)$ be a mode of *delete*. Then $outvars(delete, mode_{del}) = \{Y, Z\}$. □

We can talk about the producer or generator of a variable and its consumer. Whether a literal produces a variable or consumes it depends on whether the variable occurs in an input position or output position and whether the literal is the head literal or a body literal. A variable that occurs in an input position of the head of a clause is produced by the head. If it occurs in the output position of the head, it is consumed by the head. In the case of body literals, the situation is reversed. The reason is as follows. The input arguments of the head literal produce values for the body literals by inheriting them from the caller's environment and the output arguments consume values by passing them back to the caller's environment. For a body literal, its input arguments signify terms that are needed before its execution can start. Therefore, a body literal consumes variables in its input positions. Analogously, a body literal produces values for variables in its output positions.

Modes will be associated with the literals of a clause. Since a predicate p may appear more than once in a clause, different occurrences of p are distinguished by a subscript, usually the literal number in textual order. The head literal will be subscripted with 0.

We now define a well-moded clause. Assume that modes for the head and body literals of a clause are given. One can identify the input positions of the head and the body goals with respect to the given modes. A clause is well-moded iff there is partial ordering of the body goals such that for every body goal i, each variable in its input position either occurs in an input position of the head literal or in the output position of one of its predecessors in the partial ordering. Thus, in a well-moded clause, when all the predecessors of body goal i have been solved, the variables in the input positions of the goal i will be bound to ground terms and goal i is ready for execution. The following definition formalizes the above discussion.

Definition 5.6

Let $c = h \leftarrow b_1, \ldots, b_k$ be a clause and $mode_h$, $mode_{b_1}, \ldots, mode_{b_k}$ be the modes of the head and the body predicates of c respectively. Let $\leq$ be a partial ordering of the body goals $b_1, \ldots, b_k$ and let $pred_{\leq}(i) = \{j \mid j \leq i\}$ denote the set of predecessors of literal i in the partial order $\leq$.

Clause c is *well-moded* with respect to modes $mode_h$ and $mode_{b_1}, \ldots, mode_{b_k}$ (or $mode_h$ and $mode_{b_1}, \ldots, mode_{b_k}$ is a *well-moding* for clause c) iff there exists a partial ordering $\leq$ on the literals $b_1, \ldots, b_k$ such that for $1 \leq i \leq k$,

$$invars(b_i, mode_{b_i}) \setminus invars(h, mode_h) \quad \subseteq \quad \bigcup \{outvars(j, mode_j) \mid j \in pred_{\leq}(i)\}.$$

□

There may be more than one partial ordering of the literals of a clause that induces well-modedness. That is why the predecessor relation *pred* is subscripted with $\leq$ to indicate the partial ordering used in defining predecessors. However, wherever *pred* is used, there will be an implied partial ordering and hence we will omit the subscript on *pred*.

Consider the permutation example again. Given the modes $perm_0(-,+)$, $delete(-,+,+)$, and $perm_2(-,+)$ for the head and body literals of clause 2, the partial ordering $2 \leq 1$ satisfies our requirements, since the input variable `Ys` of literal 2 is covered (generated) by the head literal and the input variables {`Y, Zs`} of literal 1 are covered by the input variables of the head and the output variable `Zs` of literal 2. Note that the partial ordering $2 \leq 1$ is not consistent with the textual ordering 1, 2. In our definitions, we do not assume that a partial ordering of body goals is consistent with their textual order as is usually the case [32, 70]. This is because we are interested in re-ordering body goals to ensure well-modedness. We also do not require that all variables of a clause have a producer and a consumer.

Before defining permissible modes, we recall some concepts about types of procedures. A call graph of a program P is a directed graph $\langle V, E \rangle$ such that the set V of vertices consists of predicates of P, and E is the set of directed edges. There is an edge from p_1 to p_2 if there is a call to p_2 in the body of one of the clauses defining p_1. In other words, p_1 calls p_2. A procedure p is recursive iff there is a cycle involving p in the call graph of program P. Two procedures are mutually recursive iff they are in the same strongly connected component of the call graph of a program. A unit procedure is one that is defined solely by unit clauses. A non-unit procedure p is simply recursive iff it is recursive and none of its clauses contain calls to other procedures. For example, *delete* is simply recursive. A recursive procedure p is directly recursive iff it is neither simply recursive nor mutually recursive with another predicate. For example, quicksort is directly recursive, since it is recursive but neither simply recursive nor mutually recursive with another procedure. A mutually recursive procedure may also be denoted as indirectly recursive.

The framework for simultaneous data flow and control flow analysis must define the notion of permissible modes, and some initial, non-recursive constraints on the modes of procedures. This is accomplished in the following section.

5.2.2 Permissible Modes and Partial Permissible Modes

Permissible modes are used in deriving control flow. A permissible mode for a clause is a mode that satisfies some constraints. The constraints must be reasonable and intuitive so that the programmer will be able to follow them. We now present some criteria which we believe to be reasonable and some that are not. The unreasonable ones indicate the limitations of our current notion of permissible modes, and are the subject of further research. These do not limit the framework for combined data flow and control flow analysis.

We start with permissible mode constraints. These constraints depend on the type of the recursive procedure.

In the *perm* example, although $perm_0$, $perm_2$ and *delete* have two permissible modes each, not all of the eight combinations were used in clause 2 of *perm* when we derived control flow. This is because the choice of a mode for $perm_0$, $perm_2$ or *delete* constrained the others. In particular, the head and body literals $perm_0$ and $perm_2$ have the same mode. This is a mostly reasonable constraint since recursive calls in general solve a similar but smaller problem as the original one. It is not always reasonable to expect recursive calls to have the same mode as the head literal, especially when difference lists are used. The tail of a difference list may have mode ? in one recursive call (essentially a free variable in that call), but may have output mode in another recursive call. This is related to the assumption that input and output modes denote terms ground at call time and at success time respectively.

This constraint in effect collapses n literals of a clause having the same predicate symbol as the head literal into one equivalence class. Thus, instead of considering k permissible modes for each of n literals (leading to k^n choices), we have to consider only k modes when deriving permissible modes for the n literals in an equivalence class.

Permissible Mode Constraint 1

> Body literals (if any) that have the same predicate symbol as the head must have the same modes as the head. □

Note that the above requirement does not mean that all body goals with the same predicate symbol must have the same mode. They must have the same mode only if their predicate symbol is the same as the head literal's predicate symbol. Recall that we use a ranked alphabet of predicate symbols, where the rank of a predicate is its arity, and hence each predicate symbol has associated with it only one arity.

Referring to the *perm* example, *delete* caused an infinite loop when called with variable terms for the argument positions 1 and 3. These are the positions at which the recursive call's arguments are more general[4] than the corresponding head arguments. In such cases, if at least one of these arguments is not sufficiently instantiated (in this case, if not a complete list), infinite derivations will result. Consider a simpler example *concat*.

Example 5.6

The result of concatenating an empty list with L is L itself. Concatenating the list [X|Xs] with Y is equal to the list [X|Zs] if concatenating Xs with Y is equal to Zs.

```
1.  concat([], L, L).
2.  concat([X|Xs], Y, [X|Zs]) :-
        concat(Xs, Y, Zs).
```

□

Note that the first and third arguments Xs and Zs of the recursive call are subterms of the corresponding arguments of the head. It is well-known that *concat* can be used to catenate its first two arguments or to split the third argument into two lists. If the recursive call is indeed intended to solve a smaller problem, then either the first or the third argument must become smaller with each recursive call. This can happen only when either the first or the third is a complete list when *concat* is called. This idea of comparing pairs of arguments to detect structural induction was first suggested by Naish in his dissertation [64].

Permissible Mode Constraint 2

Let $\{i_1, \ldots, i_n\}$ be the set of arguments of the head of a clause in simply recursive procedure p with arity k where $n \leq k$ and the arguments $\{i_1, \ldots, i_n\}$ of a recursive body goal for p are more general than the corresponding head arguments. Then, in a permissible mode at least one of the arguments $\{i_1, \ldots, i_n\}$ must be input. If the set of such arguments is empty, then at least one of the arguments $1, \ldots, k$ must be input in a permissible mode.

For a clause of a recursive but not simply recursive procedure p, at least one of the arguments must be input. □

Example 5.7

It is possible that none of the head arguments of a clause of a simply recursive procedure is less general than the corresponding arguments of a recursive body goal. This may be due to syntactic variations, as illustrated in the following program.

[4] Recall that a term t_1 is more general then another term t_2 if there exists a substitution θ such that $t_2 \equiv t_1\theta$.

```
1.  concat([], L, L).
2.  concat(A, Y, B) :-
        A = [X|Xs],
        B = [X|Zs],
        concat(Xs, Y, Zs).
```

□

The first and third head arguments of the second clause of the usual *concat* procedure are replaced by new variables, and the unifications represented by these arguments appear as two explicit unification goals in the body of the clause. In another variation, one may replace the goal `A = [X|Xs]` with the goal `head_tail(A, X, Xs)` and introduce the following unit clause defining *head_tail*:

```
3.  head_tail([X|Xs], X, Xs).
```

Simple source-to-source transformations can remove these unifications at compile-time. The body goals `A = [X|Xs]`, and `B = [X|Zs]` can be removed and the unifications moved to the head. Likewise, any unit procedure clause can be folded in to the calling clauses. We assume that such simple transformations have been done. In the absence of side-effects, such transformations can always be done. The presence of cuts and other impure features may disallow such transformations. However, such transformations may be made only to determine which arguments need to be constrained to be input. Issues such as separate compilation, and information hiding may preclude such transformations and introduce incompleteness.

Permissible Mode Constraint 3

A permissible mode must be consistent. □

Recall that a mode for a procedure p is consistent if and only if it maps no argument of p to $\top$, *i.e.*, a permissible mode must not require some argument of a procedure to be input and output at the same time. This represents an over-constrained argument whereas the $\perp$ represents an under-constrained argument.

There are no a *priori* constraints on the permissible modes of a unit procedure. The permissible modes of a non-unit, non-recursive procedure are determined by the permissible modes of its body goals and hence, there are no a *priori* constraints on the permissible modes of a non-unit, non-recursive procedures. If a non-unit, non-recursive procedure p calls other recursive procedures, then the permissible modes of the called procedures will induce some constraints on p.

In practice, for example in a data base context, the relations defined by unit procedures (also known as extensional database relations as opposed to intensional database relations defined by recursive predicates) may be very large. It may be reasonable to impose some constraints on permissible modes for unit procedures. Suppose that some efficient access methods or indices exist for some attributes of a large relation. We may want to allow retrievals of the tuples of that relation only when values for one or more of the indexed attributes are known. This avoids retrieving the whole relation and encourages the use of indices to select (in the relational algebraic sense) a subset of the relation more efficiently.

Thus, there are three initial constraints to be satisfied by a permissible mode. We can use a mode that satisfies constraint 2 and 3 as a starting point or an initial approximation for a permissible mode and extend the initial approximation until it satisfies permissible mode definition. The initial approximations of permissible modes are called partial permissible modes.

Definition 5.7

A mode m for a clause c is a *partial permissible* mode, if it satisfies permissible mode constraints 2 and 3.

A mode m_2 for a clause c is an *extension* of a mode m_1 for c iff $m_1 \sqsubseteq m_2$ and m_2 is consistent. □

In the case of clause 2 of concat procedure, the modes $concat(+,?,?)$ and $concat(?,?,+)$ are partial permissible modes, since arguments one and three of the head of the clause are less general than the corresponding goal arguments. Furthermore, these arguments are consistent.

An extension or refinement of a mode for a clause c is a refinement of the modality for some argument from $\perp$ (unknown) to + (input) or − (output). For example, the mode $concat(+,+,?)$ for clause 2 of concat is an extension of the mode $concat(+,?,?)$.

A permissible mode is one that satisfies the initial constraints and, in addition, induces well-modedness for a clause.

Definition 5.8

Let c be a clause $h \leftarrow b_1,\ldots,b_k$ of procedure p. A mode m for clause c is a permissible mode if it satisfies permissible mode constraints 1, 2, and 3 and there exist permissible modes $PM = \{mode_{b_1},\ldots,mode_{b_k}\}$ for the body goals of c such that the clause c is well-moded with respect to mode m and PM. □

Notice that the above definition is recursive, in that the permissibility of a mode for a clause c of procedure p is defined in terms of the minimal constraints and the permissibility of the modes for the body goals which may include calls to p itself, or calls to other procedures which are mutually recursive with p. As is usual with the recursive definitions, we take the definition to denote the least fixed-point of the associated functional. In other words, we take a permissible mode to be the least mode that satisfies the above definition. One can visualize a method which starts with partial permissible modes and progressively converts the unknown modes $\perp$ to + or − to satisfy input and output constraints of the body goals. In fact, the algorithms given later in this chapter follow the above outline.

Example 5.8

Consider the *delete* procedure of the *perm* example again. The mode $delete(-,+,+)$ is a permissible mode for the second clause of *delete* because of the following reasons. It induces the mode $delete_1(-,+,+)$ for the first and only body goal; the induced mode is the same as the mode for the clause and hence satisfies constraint 1; at least one argument is input and hence constraint 2 is satisfied; it is consistent and hence constraint 3 is satisfied; the input variables of the only body goal are produced by the input positions in the head and hence the clause is well-moded. □

Having defined permissible modes for clauses, we require that all clauses of a procedure have the same permissible mode(s).

Definition 5.9

A mode m is a permissible mode for a procedure p iff m is a permissible mode for all clauses defining p. □

We have not specified any additional constraints for mutually recursive procedures. Constraint 2 says that clauses of directly recursive and mutually recursive procedures must have at least one input argument. This, coupled with the definition of permissible modes, is enough to decide whether a given mode for a mutually recursive or directly recursive procedure is a permissible mode.

5.2.3 Mode Intersection

Given permissible modes for clauses of a procedure, we should be able to derive permissible modes for a procedure. Intuitively, a permissible mode for a procedure must satisfy at least one permissible mode for each clause in that procedure. As mentioned before, the mode intersection of the modes m_1 and m_2 of two clauses of a procedure can be obtained simply as $m_1 \sqcup m_2$. This is because each of the modes specify some constraint on the arguments and we want to satisfy both specifications by taking the lub (join) of the modes. Since the set of modes for a procedure is a complete lattice, mode intersection is well-defined.

Definition 5.10

Let M_1 and M_2 be sets of modes of clauses c_1 and c_2 of procedure p. Then the intersection of modes M_1 and M_2, denoted $M_1 \otimes M_2$, is defined as $\{x \sqcup y \mid x \in M_1, y \in M_2\} \setminus \{\top\}$.

More generally, let $M_1, \ldots, M_k$ be sets of modes of clauses $c_1, \ldots, c_k$ respectively of procedure p. Then, the mode intersection $M_1 \otimes \ldots \otimes M_k$ is defined as $\{\sqcup\{x_i \mid x_i \in M_i, 1 \leq i \leq k\}\} \setminus \{\top\}$. □

Example 5.9

Let $\{p(+,\bot,\bot), p(-,+,\bot)\}$, $\{p(+,-,\bot)\}$ be sets of permissible modes for two clauses of p. Their mode intersection consists of the singleton set $\{p(+,-,\bot)\}$. The join of $p(-,+,\bot)$ and $p(+,-,\bot)$ results in an inconsistent mode $p(\top,\top,\bot)$ which is discarded. □

Proposition 5.1

Let $M_{c_1}, \ldots, M_{c_k}$ be sets of permissible modes of clauses $c_1, \ldots, c_k$ of a procedure p. Then, all the modes in the set $M_{c_1} \otimes \ldots \otimes M_{c_k}$ are permissible modes of procedure p.

Proof: Follows from the definitions of permissible modes for a procedure and mode intersection $\otimes$. □

5.2.4 Minimal Modes

We are interested in deriving minimal or least restrictive permissible modes for a procedure. Intuitively, a mode ? is less restrictive than a mode + or − since ? imposes no constraints. Likewise, a mode − is less restrictive than + because the output argument is required to be ground only at the time of success, whereas an input argument must be ground at the time of call.

For example, the mode *delete*$(-,+,+)$ and the mode *delete*$(+,+,+)$ are both permissible modes for the *delete* procedure of permutation example 5.1. The mode *delete*$(-,+,+)$ is intuitively less restrictive than *delete*$(+,+,+)$ because the first argument of a call to *delete* is not required to be input. The notion of minimality can be formalized by considering another partial ordering $\leq$ on the space of permissible modes for a procedure derived from the ordering $\perp \leq - \leq + \leq \top$.

Definition 5.11

Let M_p be the set of permissible modes of an n-ary procedure p. Then for all $m_p, m'_p \in M_p$, $m_p \leq m'_p$ iff $m_p(i) \leq m'_p(i)$, $1 \leq i \leq n$. □

The ordering $\leq_{pm}$ on the set PM of permissible modes of p is a partial ordering since the ordering $\leq$ on $\{\perp, +, -, \top\}$ is a partial ordering.

The above ordering is different from the one used to order the permissible modes of different clauses (Definition 5.2). The need for two different orderings on permissible modes can be explained as follows.

Suppose that $p(+,-,-)$ is a permissible mode for a clause c of p and $p(-,+,-)$ is a permissible mode for another clause c' of p. The mode $p(+,-,-)$ may be permissible for c because there is a recursive call to $p(X,Y,Z)$ in the body of c and $p(X,Y,Z)$ is the sole producer for Y which is consumed by another body goal of c. When the permissible mode $p(+,-,-)$ for clause c and the permissible mode $p(-,+,-)$ for clause c' are combined to form a permissible mode for the procedure p, one cannot change $p(+,-,-)$ to $p(+,+,-)$ because the mode $p(+,+,-)$ is no longer a permissible mode for clause c. Thus, for the purposes of mode intersection, the mode annotations + and − are not comparable. Once we combine the clause modes and obtain the set S of permissible modes for a procedure, we can use a different ordering which treats $\perp$ as less restrictive than − which is less restrictive than + and retains only the minimal permissible modes.

The mode intersection of sets of minimal permissible modes of clauses in p will be permissible modes for p but not necessarily minimal as shown by the following example.

Example 5.10

Consider two sets of minimal permissible modes $\{p(+,?), p(?,+)\}$, $\{p(?,?), p(+,?)\}$ for two clauses of procedure p. The mode intersection of these sets is the set $\{p(+,?), p(+,+), p(?,+)\}$ which has three permissible modes for the procedure. This set is not minimal since $p(+,?) \sqsubseteq_{pm} p(+,+)$. □

To obtain the set of minimal permissible modes for a procedure p from the minimal permissible modes of the clauses of p, we must throw away the non-minimal modes after mode intersection. The next definition shows how minimal permissible modes of a procedure p may be obtained from the minimal permissible modes of the clauses of p.

Definition 5.12

Let M be a set of modes for a procedure p. Then $min(M)$ is defined to be the set of minimal modes of M with respect to the ordering $\leq$, *i.e.*, $min(M) = \{x \mid x \in M \ \wedge \ \nexists y \in M.\ y \leq x\}$. □

Proposition 5.2

Let $M_{c_1}, \ldots, M_{c_k}$ be the sets of minimal permissible modes of the clauses $c_1, \ldots, c_k$ of a procedure p. Then $min(M_{c_1} \otimes \ldots \otimes M_{c_k})$ is the set of minimal permissible modes of procedure p. □

Proof

Follows from the definitions of min, permissible modes for a procedure and mode intersection $\otimes$. □

The mode intersection operator $\otimes$ is defined in terms of the lub operation on the complete lattice of modes. This means that mode intersection inherits some algebraic properties from the lub operation on the complete lattice of modes. This will be useful in reducing the search space when we derive minimal permissible modes for a procedure.

Proposition 5.3

The mode intersection operator $\otimes$ is associative, commutative, and idempotent. □

Proof

Follows from the definition of $\otimes$. □

We defined modes, partial permissible modes and permissible modes. Partial permissible modes satisfy some initial constraints and are used as initial approximations for deriving permissible modes. The type of a recursive procedure is used solely to determine the input arguments.

Mode intersection was also defined for combining permissible modes of different clauses to form permissible modes for a procedure. The notion of minimal permissible modes was introduced to eliminate redundant and overly restrictive permissible modes.

In the next section we prove the intractability of deriving minimal permissible modes.

5.3 Intractability of Deriving Minimal Permissible Modes

Given a set of minimal permissible modes for the subgoals of a clause, is deriving minimal permissible modes for the clause feasible? We show that this problem is computationally hard in general.

The problem of deciding whether a permissible mode is minimal is NP-complete and hence the problem of deriving all minimal permissible modes of a clause is NP-hard. Recall that a decision problem is NP-complete iff it is in NP and all problems in NP can be transformed to it in polynomial time by a deterministic Turing machine [41]. A decision problem is NP-hard if an NP-complete problem can be reduced to it. We now define the set covering problem which is NP-complete and then show that deriving minimal permissible modes has the set covering problem as a sub-problem.

Theorem 5.1 [Minimum Cover]

Given: A collection C of subsets of a set S, a positive integer K.

Question: Does C contain a cover for S of size K or less, that is a subset $C' \subseteq C$ with $|C'| \leq K$ and such that $\bigcup_{c \in C'} c = S$?

Minimum Cover is NP-complete. For proof, see Garey and Johnson [41]. □

Theorem 5.2 [Minimal Permissible Mode]

Given a clause $c = h \leftarrow b_1, \ldots, b_k$ and sets of minimal permissible modes $M_1, \ldots, M_k$ for the goals $b_1, \ldots, b_k$ respectively, finding the set of all minimal permissible modes for c is NP-hard. □

Proof

We will restrict the problem and show that the restriction contains the minimum cover problem as a special case. This proves the theorem.

Consider a clause $p(t_1, \ldots, t_n) \leftarrow q(s_1, \ldots, s_m)$. Suppose that q has only one minimal permissible mode $mode_q$. Given the mode $mode_q$, we have to derive the minimal permissible modes of c. Suppose further that the set $S = invars(q, mode_q)$ of input variables of q is the same as the set of all clause variables, *i.e.*, all variables of q occur in the head. Let C be the collection of sets of variables appearing in terms $t_1, \ldots, t_n$.

It is obvious that with these restrictions, the minimal permissible modes problem contains the minimum cover problem.

Question: Does C contain a cover for S of size K or less, that is a subset $C' \subseteq C$ with $|C'| \leq K$ and such that $\bigcup_{c \in C'} c = S$?

The answer to the above question is yes if there exists a minimal permissible mode for the clause such that the number of input positions in the minimal permissible mode is less than or equal to K and no otherwise. Therefore, the minimal permissible modes problem is at least as hard as the minimum cover problem.

□

Note that in most programs, the arity of predicates tends not to increase with the size of the programs. Hence, if we assumed a bound on the arity of predicates, the above NP-hardness result does not hold.

5.4 An Algorithm for Deriving Permissible Modes

We start with a description of an approximation algorithm for deriving minimal permissible modes for all predicates in a program and use examples to illustrate the method before formally defining the algorithm. The algorithm is approximate since it may find non-minimal permissible modes.

5.4.1 Outline

First construct the call graph of a program, collapse the maximal strongly connected components and derive a reduced graph.[5] Assign numbers to the predicates such that if p calls q, then q has a lower number than p unless they are in the same strongly connected component, in which case they both have the same number. This is known as the "dfs" (depth-first search) number of a predicate [1]. Finally, find permissible modes for the predicates in ascending order of the dfs numbers of procedures, *i.e.,* derive permissible modes for predicates with a smaller dfs number before deriving for those with larger numbers.

Permissible modes of a clause are found in two steps. The first step involves deriving a set of partial permissible modes. The second step is an iterative step which refines the given partial mode and extends a given partial ordering of body goals with one or more body goals using a greedy approach. This step is repeated until all body goals are added to the partial ordering or the mode cannot be extended.

Once permissible modes are found for all clauses in a procedure, the permissible modes for a procedure can be found by mode intersection. We can use the idempotence, associativity, and commutativity of mode intersection to optimize the derivation of permissible modes of a clause. The idea is to maintain a set of "running" permissible modes for a procedure. Initially, it is the singleton set containing the least element of the mode lattice for the procedure, *i.e.,* a mode that does not require an argument to be either input or output. Suppose there are $n > 1$ clauses in a procedure. Order the clauses according to the total number of modes for the body goals and derive the permissible modes for the clauses in ascending order of the total number of permissible modes for body goals. After the first step for each clause which derives the set of partial permissible modes, but before starting the second iterative step, take the mode intersection of the set of running permissible modes for the procedure and the set of partial permissible modes for the current clause and use it as input to the second, iterative, mode-refining step. At the end of the iterative step, the set of permissible modes for the current clause becomes the new set of running permissible modes for the procedure and is used similarly in deriving the permissible modes for the next clause in the procedure.

The steps in the derivation of permissible modes are listed below.

Partial Permissible Modes for a clause :

The set of partial permissible modes for a clause is simply a set of modes, each of which constrains an argument to be input. The number of partial permissible for a recursive procedure depends on whether it is simply recursive or not.

Extend Partial Permissible Modes :

Each partial permissible mode is extended such that the clause is well-moded.

Mode Intersection :

Once permissible modes for all clauses of a procedure are derived, combine them (using mode intersection) into permissible modes for a procedure.

[5] This can be done in time $\mathcal{O}(max(|V|, |E|))$, *i.e.,* linear in the size of the graph [1].

Mutually Recursive Procedures :

For mutually recursive procedures, iterate the previous two steps until permissible modes of a set of mutually recursive procedures stabilize.

5.4.2 Derivation of Partial Permissible Modes

A partial permissible mode constrains a head argument to be input. The type of the recursive procedure, namely simply recursive or otherwise, determines which arguments are so constrained.

Definition 5.13

Let $S = \{l, m, n, \ldots\}$ be the set of arguments of the head of a clause c of a simply recursive procedure p with arity k such that the argument $i \in S$ of a body goal with the same predicate symbol as the head is more general than the corresponding head argument. If the set of such arguments is empty, then let S be $\{1, \ldots, k\}$.

If the procedure p is recursive but not simply recursive, then let $S = \{1, \ldots, k\}$ be the set of arguments of the head of a clause c of p.

Let $input\_mode_p^i$ be a mode of p such that

$$input\_mode_p^i(j) = \begin{cases} + & \text{if } j = i \\ \perp & \text{otherwise} \end{cases}$$

$input\_mode_p^i$ constrains the argument i of p to be input and all others to be $\perp$. Then the set of partial permissible modes PPM of p is $\{input\_mode_p^j \mid j \in S\}$.

□

Proposition 5.4

For all recursive procedures p, the partial permissible modes of p derived according to Definition 5.13 satisfy permissible mode constraint 2 and 3 (Pages 88–89). □

This is because the Definition 5.13 simply enforces the permissible mode constraints 2 and 3 on the partial modes of a simply recursive procedure. Set S in the above definition is derived directly from the definition of permissible mode constraint 2. The set of partial permissible modes PPM is only an initial approximation because we have to ensure the well-modedness of the body of clause c. There may be more than one recursive call in the body of a clause of a simply recursive procedure or there may be calls to built-ins such as arithmetic (`A is X + Y`) or comparison predicates (`X > Y`) whose input requirements must be satisfied. Even when there is only one recursive call and there are no calls to built-in predicates, we have to check the well-modedness of the single literal body. Of the two examples shown below, the first one does not need the well-modedness check. The second one does.

Example 5.11

Consider the procedure *concat*.

```
1.  concat([], L, L).
2.  concat([X|Xs], Y, [X|Zs]) :-
        concat(Xs, Y, Zs).
```

The partial permissible modes for *concat* are *concat*$(+, ?, ?)$, and *concat*$(?, ?, +)$. Arguments 1 and 3 of the recursive call to *concat* in clause 2 are subterms of the respective head arguments. This implies that if argument 1 or 3 is ground in the initial call, all recursive calls will also have this property. Hence, the permissible mode constraint 1 (which says that the head literal and body literals with the same predicate symbol as the head literal must have the same mode) is satisfied. Thus the initial approximation turns out to be final in this case. □

When an argument of a recursive call is not a subterm of the corresponding head argument but is more general, then the initial approximation is not likely to be the final one. The following example illustrates why partial modes need to be extended to permissible modes.

Example 5.12

Informally, the procedure *split* may be used to split a list, (the first argument), evenly among two lists, (the second and third arguments), preserving the order of elements. Or, it can be used to merge the elements of the second and third arguments into the first such that the alternate elements of the first are from the second and third arguments.

```
1.  split([A|B], [A|C], D) :-
        split(B, D, C).
2.  split([], [], []).
```

The first two arguments of `split(B, D, C)` are more general than the corresponding head arguments. Therefore, the initial approximation to the set of permissible modes for *split* is $\{split(+, ?, ?), split(?, +, ?)\}$.

Consider the mode *split*$(+, ?, ?)$ first. By permissible mode constraint 1, the recursive call also must have the mode *split*$(+, ?, ?)$. Since the first argument `B` of the recursive call is a subterm of the first argument of the head, the mode *split*$(+, ?, ?)$ for the head induces the same mode for the recursive call and thus *split*$(+, ?, ?)$ is a permissible mode.

We do not need a complex data flow analysis to verify the permissibility of the mode *split*$(+, ?, ?)$ for two reasons. One, there are no output constraints to be verified and two, the subterm property is enough to verify that an initial call with mode *split*$(+, ?, ?)$ will induce a recursive call which satisfies the same input constraint on argument one.

Now consider the second partial mode *split*$(?, +, ?)$. Given the mode *split*$(?, +, ?)$ for the head, all we know (after head–goal unification) is that `A` and `C` will be ground. Since the recursive call is required to have the same mode, our algorithm finds a producer for the second argument `D` of the recursive call *split* in the body. The only way D can be input to the recursive call is if the third argument D of the head literal is input. Thus, another permissible mode for *split* is *split*$(?,+,+)$. This process may lead to a situation where some input constraint of a body goal cannot be satisfied or we derive an inconsistent mode. In such cases, we discard the initial approximation. □

A partial permissible mode usually has to be extended before it becomes permissible. We will see how to extend a partial mode in the next section.

In the following, we assume that modes for built-ins are known. Likewise, when we are processing the clauses of some procedure p with a depth-first search (dfs) number k, we assume that permissible modes for all procedures with dfs number $< k$ have been derived. As mentioned before, deriving the permissible modes of procedures in ascending order of their dfs numbers will ensure this.

5.4.3 Extension of Partial Modes: Discussion

The strategy for converting partial permissible modes to permissible modes may be summarized as follows. Start with a set of partial modes and try to extend them such that the extension induces well-modedness for the clause. A partial mode may be extended in many ways and so extending a partial mode may result in a set of permissible modes. Or, we may choose some deterministic extension method and return only one permissible mode per partial mode. This choice bounds the number of permissible modes per clause. This is because we start with at most n partial modes per clause, where n is the arity of the head predicate and each partial mode is refined to at most one permissible mode. Thus there can be at most n permissible modes per clause. If there are k clauses in a procedure, there can be at most n^k the permissible modes, due to mode intersection. However, in practice the number of permissible modes per procedure appears to be bounded by the arity n of the procedure.

We associate an empty partial ordering of body goals with each of the partial modes. The set of goals in a partial ordering is referred to as the *scheduled* goals. We also associate a set of variables called *available* variables with each partial mode. Available variables are simply those that have been generated by the goals scheduled so far. Initially, this set contains the input variables of the partial mode. For example, let $p(+,?,?)$ be partial mode of a clause whose head is $p(f(X,Y),g(Z))$. Then, the set of available variables associated with this mode is $\{X,Y\}$. The following is an informal summary of the extension which either adds more goals to the partial ordering or refines one of the head arguments with the modality ? to +. The attempt to extend a partial mode may fail when there is no way to satisfy some input constraints of one or more goals. We apply the following extension repeatedly to a partial mode and its associated partial order until all body goals have been added to the partial order or it is impossible to do so. The extension of a partial mode is sub-divided into two steps, of which the second is not always needed.

Step 1

We extend the partial ordering by adding goals whose input variables are a subset of available variables. The case where no goals can be added is handled by step two. A procedure may have more than one permissible mode and each permissible mode may have a different set of variables in input argument positions. Since we use a greedy approach, we add a goal to the partial ordering as soon one of its sets of input variables is covered by (subset of) the available variables.

We are not committed to any ordering *among the newly added goals*. All we know is that their input constraints are satisfied. The ordering among the newly added goals is an issue that is orthogonal to deriving permissible modes for a clause. This will be considered later. After adding the goals to the partial ordering, we add their output variables to set of the available variables and repeat this step. The output variables of the newly added goals may be found by using abstract interpretation.

Step 2

Given a set of scheduled goals, none of the remaining goals of a clause may be schedulable; for each remaining goal g, for each of its permissible modes $mode_g$, the input variables of g with respect to mode $mode_g$ may not be a subset of available variables. In this case, we have to refine the partial mode of the clause, *i.e.*, convert the modality of some head argument position from $\perp$ to $+$. There are a number of options but they lead to exploring too many combinations. We could, for example, find the smallest set of variables that is needed to schedule at least one of the remaining goals and then find the minimal set of head arguments needed to cover these variables. But this again leads to the minimum cover problem. Alternatively, one could select a literal from among those to be scheduled (using some policy) and try to satisfy its input constraints by converting a head argument from $\perp$ to $+$.

Once again, we use a greedy approach. We find a head argument whose modality is $\perp$ and check if it is *useful* to convert it to input mode. "Usefulness" will be defined shortly. Informally, a head argument is not useful if all the variables in that argument are already generated by some other literal. Likewise, it is not useful if none of the variables in the head argument appear in the input argument positions of unscheduled goals. If we do not find a useful head argument, the partial mode cannot be extended any further and thus is discarded.

If we can extend the mode by converting the modality of some argument from unknown to input, we do so and add the variables in that argument to the set of available variables.

If the modality of some head argument is refined to input and there are recursive calls in the body that have already been scheduled, there is a complication which is discussed next.

5.4.4 Handling Recursive Goals

Recursive goals (those with the same predicate symbol as the head) of a clause pose a special problem. We start with a partial permissible mode for the clause and expect to realize a permissible mode by refining the partial mode as and when needed. This causes two problems.

The first problem is simple to state and solve. When we derive the permissible modes of a procedure p with a dfs number k, the permissible modes of all procedures with a dfs number less than k are known. In particular, a permissible mode for a procedure says which arguments are input and which are output. However, we do not know the permissible modes of p itself. In particular, when we add a recursive call to p to the partial order, we have to add the output variables of p to the set of available variables. But we do not yet know the output variables of p. This is where we use properties of programs that are independent of control flow. Recall that the meaning of a clause is a function from a goal and a call substitution to a set of success substitutions. This meaning does not depend on the order in which body goals are solved. Since we are interested only in the groundness of arguments, we can use a precise groundness analysis based on *Pos* presented in Chapter 3.

Given the input arguments of p, *i.e.*, those arguments that are bound to ground terms on call, we can derive the (output) arguments that will be bound to ground terms on success. We add these output variables to the set of available variables and refine the corresponding head arguments to be output (since the head literal and recursive call must have the same mode). The only concern is that since we are deriving permissible modes, we would call the abstract interpreter more often than in a usual data flow analysis that assumes left-to-right order of execution. This entails fixed-point computations each time the abstract interpreter is called. The benefit of *Pos* domain is that it has the "condensation" property, *i.e.*, we can summarize the abstract meaning of a procedure in a closed form using a fixed-point computation once and for all. This meaning can then be specialized to each call in the time it takes to abstractly unify the head of a clause and a goal, avoiding fixed-point computations.

The second problem caused by recursive calls is related to the non-permissibility of partial modes and the need to revise previous decisions. When a partial mode for a clause is refined successively, the intermediate partial modes may not be permissible. This is illustrated by the following example.

Example 5.13

> Let $p_0(X,Y,Z,P) \leftarrow b_1(X,Y,X1,Y1),\ldots,p_i(X1,Y1,Z1,P1),\ldots,b_j(Z,Z1),$ $\ldots,b_k(\ldots)$. be a clause. Let the current partial mode for p be $p(+,+,\bot,\bot)$ and let the current set of available variables be $\{X,Y,X1,Y1\}$. Assume that b_1's first two arguments must be input. Since the available set of variables $\{X,Y,X1,Y1\}$ covers the first two arguments $\{X,Y\}$ of b_1, b_1 is scheduled. Suppose that b_1 does not produce any variables. Now, schedule p_i since the first two arguments p_i are in the available variables set and p_i appears to satisfy the current approximate permissible mode for p.
>
> Suppose that we try to extend the partial ordering further, by scheduling b_j and b_j must have its first argument Z input. We need to refine the third argument Z of the head predicate to be input. This invalidates the previous step when the recursive call p_i was scheduled with only the first two arguments being input. The previous step may still be valid if, for example, the third argument of p_i is Z instead of $Z1$. □

If the set of variables in the newly added input argument of the head literal is a superset of the variables in the corresponding argument of the recursive call, refining the partial mode does not cause a problem. Likewise, if the set of variables in the argument of the recursive call corresponding to the newly added input argument of the head literal was al-

ready available when the recursive call was scheduled, the previous step is valid. Otherwise, the previous step is invalidated by the refinement.
We have a combination of at least four choices in dealing with this situation.

1. Relax the requirement that recursive calls and the head must have the same mode. This may not be a reasonable choice because we assumed that recursive calls solve similar but smaller problems and hence must have the same mode as the original call.

2. Abortion; recognize the extensions which cause a previous step to be invalidated and abort the attempt to extend the partial mode. This is too severe and appears to compound mistakes.

3. Graceful recovery from mistakes; redo the scheduling from the previous step, but with the new partial mode for the head predicate.

4. Avoid making guesses in the first place. Avoid a decision to schedule a recursive goal and postpone it as much as possible, *i.e.,* do not schedule a recursive goal unless it is the only goal schedulable. We may still end up in a situation where a previous scheduling of a recursive goal is invalidated.

5. A combination of the third and fourth options has the property of making fewer guesses and thus less backtracking. On the other hand, using the fourth option means postponing the scheduling of recursive goals as much as possible, which in turn implies fewer incorrect assumptions about input constraints. Using both the fourth and third options allows backtracking but reduces the frequency of such backtracking.

   We adopt option 5, a combination of third and fourth strategies. One may even add the second option with the proviso that it be used only when number of assumption revisions exceed some limit. The issue of termination is discussed later.

Note that the partial ordering of body goals we choose when refining a partial mode to a permissible mode is important but not crucial. Its main use is to show the existence (in a constructive way) of a partial ordering that induces the well-modedness of the clause with respect to the permissible mode we derived. A permissible mode for a clause may allow more than one control flow, including the one we used to derive the permissible mode. Thus, we are free to revise the control flow later as long as the permissible mode constraints are satisfied.

The above procedure works well for non-mutually recursive procedures. When deriving the permissible modes of a procedure, we assume that the permissible modes its callees are available. This, of course, is not true for a mutually recursive procedures. We now discuss how to handle mutually recursive procedures.

5.4.5 Mutually Recursive Procedures

Consider two mutually recursive procedures p and q. The permissible mode of p depends on the permissible mode of q and vice-versa. The permissible modes of mutually recursive procedures may be thought of as the least fixed-point of the permissible modes definition. This naturally leads to their computation by a sequence of approximations. The successive approximate permissible modes for p and q will be computed using the partial permissible modes of p and q as the initial approximation. The approximate permissible modes derived at stage i are used in stage $i+1$ to compute a better approximation. This procedure will either converge or terminate with failure.[6] When the approximate permissible modes of a set of mutually recursive procedures at stage i are the same as those in stage $i+1$, the procedure terminates successfully.

5.4.6 An Algorithm for Extending Partial Modes

The algorithm for extending partial permissible modes to permissible modes is now formally defined. Figures 5.3 and 5.4 show the algorithm for extending *one* partial mode to a permissible mode. If there are n partial permissible modes, we just need to apply the algorithm to each.

The algorithm in Figures 5.3 and 5.4 is sufficient to extend a partial permissible mode to a permissible mode in the case of non-mutually recursive procedures. For mutually recursive procedures, say p and q, we need a wrapper procedure which starts with the partial permissible modes of p and q as the first approximation for the permissible modes of p and q and uses the algorithm in Figure 5.3 and 5.4 repeatedly until the permissible modes stabilize (*i.e.,*reach a fixed-point) or the algorithm returns $\top$. The algorithm in Figure 5.5 is a formalization of this approach.

Correctness and termination proofs are given later (Page 113). We now proceed to a description of the variables, data structures, and the definitions of the auxiliary functions used by the partial mode extension algorithm.

As mentioned earlier, the extension algorithm starts with a partial mode and refines it at each iteration of the algorithm. The definitions are easier to understand and prove correct when the iterations are numbered and the computation state is indexed by the iteration number. Each iteration starts in state i and returns an updated state. Since the extension algorithm may backtrack, backtracking from an iteration j to a previous iteration i is accomplished simply by using the state with index i (with a small change, which we will discuss shortly) as the starting state of next iteration of the algorithm. We do not reuse iteration numbers when backtracking, *i.e.,* if an iteration j backtracks to some $i < j$, the next iteration is numbered $j+1$.

The iterations $0, 1, \ldots$, of the algorithm are denoted by the variable q. A state of the computation of our greedy extension algorithm is a 7-tuple

$$(\mathit{unsched}_q, \mathit{sched}_q, PO_q, PM^q, \mathit{avail}_q, \mathit{when\_sched}_q, q).$$

A state is characterized by the permissible modes PM, the partial-order of body goals PO, the set of goals *sched* in the partial order PO that have been scheduled, the yet to be scheduled goals[7] *Unsched*, when goals were scheduled *when_sched*, the set of variables *avail* that have been generated, and the iteration number q. The state variables are indexed by q and describe the evolution of the computation in terms of the iterations q.

[6] Termination with failure may mean the non-existence of a fixed-point, or the failure to find one by our algorithm. As discussed in the introduction, in such cases the programmer must supply the permissible mode which will be verified by the system.

[7] The parameter *Unsched* is not really necessary since it is the same as the set of body goals $\{1, \ldots, n\}$ minus the set of scheduled goals *sched*.

```
$extend\_partial\_mode : Mode \times Clause \times PM \rightarrow PM$
$extend\_partial\_mode(mode_h, c, PM) =$
let
    $q = 1$
    $PM^q$ = update $PM$ such that all recursive body goals of $c$ have
        the mode $mode_h$.
    $sched_q = \emptyset$
    $unsched_q$ = all body goals of clause $c$
    $PO_q = \emptyset$
    $avail_q = invars(h, mode_h)$
    for all literals $r$ of clause $c$, set $when\_sched_q(r)$ to $unbound$
in
    $epm(unsched_q, sched_q, PO_q, PM^q, avail_q, when\_sched_q, q)$
end
where
$epm(unsched_q, sched_q, PO_q, PM^q, avail_q, when\_sched_q, q) =$
let
    $result = epm\_iter(unsched_q, sched_q, PO_q, PM^q, avail_q, when\_sched_q)$
in
    if $result = \top$ then  $\top$
    else
        let  $(unsched_{q+1}, sched_{q+1}, PO_{q+1}, PM^{q+1}, avail_{q+1}, when\_sched_{q+1})$
                $= result$
        in
            if $unsched_{q+1} = \emptyset$ then  $PM^{q+1}$
            else  $epm(unsched_{q+1}, sched_{q+1}, PO_{q+1}, PM^{q+1}, avail_{q+1},$
                        $when\_sched_{q+1}, q + 1)$
        end
end
```

Figure 5.3: Functions *extend_partial_mode* and *epm*

```
$epm\_iter(unsched_q, sched_q, PO_q, PM^q, avail_q, when\_sched_q) =$
let  $ready = \{PM^q_{i,k} \mid b_i \in unsched_q \ \wedge\ \exists k.\ invars(b_i, PM^q_{i,k}) \subseteq avail_q\}$
in   if $ready \neq \emptyset$ then
        let  $ready\_literals = \{k \mid PM^q_{i,k} \in ready\}$,
             $sched' = sched_q \cup ready\_literals$
             $avail' = avail_q \cup output\_vars(ready)$
             $unsched' = unsched_q \setminus ready\_literals$
             $PO' = PO_q \cup \{(j,k) \mid j \in sched_q, k \in ready\_literals\}$
             update $when\_sched_q$ to reflect the scheduling of
             $ready\_literals$ in iteration $q$, giving $when\_sched'$
             if $\exists$ a rec. literal $r \in ready\_literals$ s.t. solving $r$ using the cur
             mode $PM^q_r$ as its abst. call subst. causes an $i^{th}$ arg to become
             output, then change the $i^{th}$ arg of the head and all rec. lits
             from $\perp$ to $-$, giving $PM'$; otherwise, $PM'$ is the same as $PM^q$
        in   $(unsched', sched', PO', PM', avail', when\_sched')$  end
     else  let  $new\_input\_arg = useful(h, PM^q_0, PM^q, unsched_q, avail_q)$
        in   if $new\_input\_arg = none$ then  $\top$
             else  let  update the mode of the head and rec. goals in $PM^q$ to
                        reflect the refining of head arg $new\_input\_arg$, giving $PM'$
                        $need\_revising = sched_q \cap recursive\ literals\ of\ c$
                   in   if $need\_revising = \emptyset$ then
                            $(unsched_q, sched_q, PO_q, PM',$
                                  $avail_q \cup invars(h, PM'_0), when\_sched_q)$
                        else   let  $m$ be the least integer such that
                                     $\exists r \in need\_revising$ and $when\_sched_q(r) = m$
                            in
                                 $(unsched_{m-1}, sched_{m-1}, PO_{m-1}, PM',$
                                 $avail_{m-1} \ \cup\ invars(h, PM'_0), when\_sched_{m-1})$
                            end
                   end
        end
end
```

Figure 5.4: Function *epm_iter*

```
$epm\_mr : SCC \times CurApprox \times PM \rightarrow PM$
$epm\_mr(Scc, Curapprox, PM) =$
let
    for all procedures $p \in Scc$,
    update $PM_p$ to $Curapprox_p$ giving $PM'$
in
    $epm\_mr\_iter(Scc, Curapprox, PM')$
end

where

$epm\_mr\_iter(Scc, Curapprox, PM) =$
let
    for all procedures $p \in Scc$, $Nextapprox_p =$

$\{mode\_intersection(\{extend\_partial\_mode(Curapprox_{p,i}, c, PM) \mid$ clause $c \in p\})$
        for each permissible mode $Curapprox_{p,i}$ of $p\}$
in
    if $Nextapprox = Curapprox$ then  $PM$
    else
        let  for all procedures $p \in Scc$, update $PM_p$ to
             $Nextapprox_p$ giving $PM'$
        in
             $epm\_mr\_iter(Scc, Nextapprox, PM')$
        end
end
```

Figure 5.5: Functions *epm_mr* and *epm_mr_iter*

We use PO_q to denote the partial ordering of body goals constructed during an iteration q. $Avail_q$ denotes the set of variables generated by the head literal and the literals scheduled in iteration q. The permissible modes of a clause c in iteration q is denoted by PM^q. The i^{th} element of PM^q is an array of permissible modes for literal i in iteration q. Therefore, $PM^q_{i,k}$ denotes the k^{th} permissible mode of the i^{th} literal in iteration q. We may omit the superscript q on PM when it is used in a context where the iteration q is known. Furthermore, since the head literal (0) and any recursive goal j with the same predicate symbol as the head literal will have only one mode (during the process of refining one partial mode), we will denote their mode by PM^q_0 and PM^q_j, *i.e.*, omit the second subscript.

If the dfs number of the predicate of a body literal k is less than that of the head of a clause, then the permissible modes of k have been derived prior to deriving the permissible modes of the clause. Thus k's permissible modes do not change. On the other hand, the mode of the head literal and the recursive calls, if any, in the body is initially a partial mode and may be refined in an iteration. Thus, we could have used different variables to denote the array of permissible modes of body literals, but it would complicate the definition of our algorithm.

As discussed earlier, refining the modality of a head argument from $\perp$ to $+$ in the i^{th} iteration may force backtracking if we had scheduled a recursive literal at an earlier iteration. There may be more than one recursive literal scheduled earlier on different iterations. We need to backtrack to the earliest iteration which scheduled a recursive literal and resume from there and thus we need to know the iteration to which we must backtrack. The vector *when_sched* maps literal i to the iteration in which it was scheduled. For all literals r of clause c, *when_sched*(r) is unbound at the beginning. When a literal r is scheduled at step q, *when_sched*(r) is bound to q.

When computation backtracks from iteration j to i, the state to be used for the next iteration $j+1$ is almost the same as in iteration i. (If we use the same state as i, we would get into an infinite loop.) The only difference is that the modes of the head literal and recursive literals (if any) have been refined and thus the set of available variables in iteration $j+1$ is the set of available variables in iteration i, union the variables in the newly refined argument of the head literal.

The algorithm *extend_partial_mode* takes as input a partial mode for the head of clause c, clause c, an array PM of permissible modes, and returns either an updated PM or $\top$ indicating failure to extend the partial mode. Function *extend_partial_mode* creates some initial bindings for variables and calls *epm* which performs the iterative refinement. Function *epm* calls *epm_iter* which takes $sched_q$, $Unsched_q$, $avail_q$, PM and PO_q as parameters and returns the updated versions for the next iteration $q+1$. The function *epm* simply tests for termination conditions.

During the extension of a partial mode, there may not be any schedulable goals because their input variables are not covered by the available variables. In such a case, some head argument will be constrained to be input. The notion of *usefulness* was needed to select a head argument. A head argument is j *useful* iff its modality is $\perp$ (*i.e.*, it is unconstrained so far), the set of variables V that occur in it are not a subset of available variables and V covers at least some set V' of input variables of one of the unscheduled goals such that V' is not covered by the available variables. The reason for classifying a head argument as useful is that refining its modality must eventually lead to scheduling of some goal. If some head argument is already classified as input or output, it cannot be constrained further. Likewise, if the set of variables V that occur in it is a subset of available variables, then these are already generated by some other scheduled literal or these already occur in some input argument positions in the head.

The following definition of *useful* is a formalization of the above discussion. There may be more than one such head argument. The function *useful* returns the least such argument and if there is no such argument, it returns *none* which is defined to be the bottom element of $\mathcal{N}$. In the following, $arg(j,h)$ denotes the j^{th} argument of h, and b_k denotes the k^{th} body goal of the clause whose head is h.

Definition 5.14

$useful : Atom \times Mode \times Permissible\_Modes \times \mathcal{P}(Atom) \times \mathcal{P}(Var) \rightarrow \mathcal{N}_\perp$

$useful(h, mode_h, PM^q, unsched, Avail) =$

$$\begin{array}{l} \text{the least } j \text{ s.t } \; mode_h(j) = \perp \;\wedge \\ \qquad Vars(arg(j,h)) \not\subseteq Avail \;\wedge \\ \qquad (\exists i,k.\; k \in unsched \;\wedge \\ \qquad\; Vars(arg(j,h)) \;\cap\; (invars(k, PM^q_{i,k}) \setminus Avail) \neq \emptyset) \\ \text{if such a } j \text{ exists} \\ \text{otherwise } none \end{array}$$

□

Note On Free Variables in Functions

The functions defined in Figures 5.3, 5.4, and 5.5 have the following free variable occurrences: h, c and *output_vars*. The variable h is the head literal of clause c whose permissible mode we are deriving. The variable *output_vars* stands for a function which maps a set of modes such as $\{b_i(+,?,?), b_j(?,+)\}$ to a set of variables of clause c which would be bound to ground terms on successful execution of b_i and b_j. Suppose that using our freeness, groundness, and sharing analysis of Chapter 3 or the groundness analysis based on *Pos*, we conclude that the abstract success pattern corresponding to $b_i(ground\_term, ?, ?)$ is $b_i(ground\_term, ground\_term, ?)$. Then, the set of variables that occur in the second argument of literal b_i of clause c will be bound to ground terms on b_i's success. These are the *output* variables of b_i.

It is possible to remove these free occurrences by passing them as parameters to all functions that need them but we will not do so. We assume that the program P and a sound groundness analysis are given and remain fixed throughout the derivation of permissible modes for all procedures.

5.5 Examples

In this section, we derive permissible modes for some small examples using the algorithm described in the previous section. We start with two different versions of a permutation program.

Example 5.14

```
1.  perm1(□, □).
2.  perm1([X|Xs], [Y|Ys]) :-
        delete1([X|Xs], Y, Zs),
        perm1(Zs, Ys).
3.  delete1([A|Rest], A, Rest).
4.  delete1([A,B|Bs], X, [A|Rest]) :-
        delete1([B|Bs], X, Rest).
```

□

Since *delete1* has a smaller dfs number than *perm1*, we process *delete1* first. The partial permissible modes of clause 2 of *delete1* are *delete1*$(+,\perp,\perp)$ and *delete1*$(\perp,\perp,+)$, since the first and third head arguments are less general than those of the recursive body goal.

The partial mode *delete1*$(+,\perp,\perp)$ implies the set {`A,B,Bs`} of variables are available after head unification and these cover the variables {`B,Bs`} in the input position of the only body goal. Hence *delete1*$(+,\perp,\perp)$ is a permissible mode. Groundness analysis shows that a ground first argument of *delete1* implies the second and third will also be ground, and thus the permissible mode is extended to *delete1*$(+,-,-)$.

Likewise, the partial mode *delete1*$(\perp,\perp,+)$ turns out to be a permissible mode. In this case, however, groundness analysis does not refine the permissible mode *delete1*$(\perp,\perp,+)$. This is because a ground third argument of *delete1* does not the imply the groundness of any other argument.

Now consider *perm1*. Its partial permissible modes are *perm1*$(+,\perp)$ and *perm1*$(\perp,+)$. We extend *perm1*$(+,\perp)$ first. The mode *perm1*$(+,\perp)$ implies the variables {`X,Xs`} are available after head unification. These cover the input variables {`X,Xs`} of one of the permissible modes of *delete1* but not the input variables of the partial permissible mode of the recursive call *perm1*. Hence, *delete1* is the first literal to be scheduled. The output variables {`Y,Zs`} are added to the available variables. Now the recursive call *perm1* becomes schedulable, since the input variable `Zs` of one of its permissible modes is covered by the available set. Thus, the partial permissible mode *perm1*$(+,\perp)$ is also a final permissible mode. Groundness analysis yields permissible mode *perm1*$(+,-)$.

Next, we extend the second permissible mode *perm1*$(\perp,+)$. The available variables after head unification are {`Y,Ys`}. These do not cover the input variables of any of the permissible modes of *delete1* but do cover the input variables of the partial permissible mode of recursive call. Recall that our strategy is to reduce backtracking by postponing the scheduling of a recursive goal unless no other goal can be scheduled. Hence, the goal `perm1` is scheduled. We need to find the output variables of the recursive call, given that its second argument is ground on call. Using groundness analysis, a call to `perm1` with the mode *perm1*$(\perp,+)$ grounds the first argument. The variable `Zs` is added to the set of available variables {`Y,Ys`}. The new set of available variables {`Y,Ys,Zs`} covers the input variable `Zs` of the permissible mode *delete1*$(\perp,\perp,+)$ of the unscheduled goal *delete1*; in fact, it covers the second argument `Y` of *delete1* as well. The output variables {`X,Xs`} of *delete1* are found, once again using abstract interpretation. The set of available variables now is {`Y,Ys,Zs,X,Xs`}. The partial mode *perm1*$(\perp,+)$ is thus a permissible mode and groundness analysis refines the mode to *perm1*$(-,+)$. Notice that there were no wrong "guesses" and hence no backtracking in the example.

Next, we consider another version of permutation.

Example 5.15

```
1.  perm2([], []).
2.  perm2(L, [H|T]) :-
        append(V, [H|U], L),
        perm2(W, T),
        append(V, U, W).
3.  append([],B,B).
4.  append([X|A],B,[X|C]) :-
        append(A,B,C).
```

□

The permissible modes of *append* are: *append*$(+, \perp, \perp)$ and *append*$(-, -, +)$. These are similar to those of *delete* and derived similarly.

The partial permissible modes of *perm2* are *perm2*$(+, \perp)$ and *perm2*$(\perp, +)$ as in the case of *perm1*. Consider the extension of *perm2*$(+, \perp)$. The set of available variables after head unification is {L}. It covers the input variable L of a permissible mode *append*$(-, -, +)$ of the first body goal and hence $append_1$ is scheduled. The output variables of $append_1$, given that its third argument L is ground, are {V,H,U}. Now the set of available variables is {L,V,H,U}. This covers the input variables {V,U} of the third body goal $append_3$ which is scheduled next. The output variable W of $append_3$ is added to the set of available variables. Next, we schedule the recursive goal *perm2* since its input variable W is covered by the available variables. There are no more unscheduled goals.

Similarly, extending the second partial mode *perm2*$(\perp, +)$ results in a permissible mode *perm2*$(-, +)$.

Notice the asymmetry in the permissible mode for *perm2* when the first or the second argument is input. When the first argument is input, the permissible mode is found to be *perm2*$(+, \perp)$, *i.e.,* there is no constraint on the second argument of *perm2*. This is understandable because the second argument T is used only to pass information to the caller and is not used in other non-recursive goals of clause 2 of *perm2*. On the other hand, when the partial mode *perm2*$(\perp, +)$ is extended to the permissible mode *perm2*$(-, +)$, the first argument is required to be output. This can be explained as follows. The first argument W of the recursive call in the body of clause 2 of *perm2* also appears in the input position of a permissible mode (call it $mode_{append}$) of the third goal and appears nowhere else. Thus, in order for the clause to be well-moded, the first argument of *perm2* must be output, if $mode_{append}$ is the permissible mode used to schedule the third goal.

The ordering of body goals induced by the permissible mode *perm2*$(+, \perp)$ is 1, 3, 2. The ordering of body goals induced by the permissible mode *perm2*$(-, +)$ is 2, 3, 1. Notice that assuming a fixed control flow would miss at least one of the two orderings. As in the previous example, the partial mode extension algorithm does not backtrack. Notice also that there are six possible total orderings (three body goals give rise to 3! orderings) but our algorithm is oblivious to them. It does not explore the space of all possible orderings.

The data flow constraints in the above examples forced the sequentialization of body goals. Next, we consider *quicksort*, a typical divide-and-conquer algorithm which has scope for parallelism. This example is a rich source of many points we want to illustrate. One is that the ordering derived by the partial mode extension algorithm is only incidental to the derivation of permissible modes. It is a constructive proof of well-modedness of a clause.

We may not want to use the same ordering for different reasons. This point is exemplified by *quicksort*.

Another point is that the derivation of permissible modes for *quicksort* has room for more optimizations of the partial mode extension algorithm. We will discuss two such optimizations. First, a "running" permissible mode for a procedure can be used to reduce the search for the head arguments that need to be refined from $\perp$ to $+$ or $-$. Second, groundness dependency information from an analysis based on *Pos* can be used to limit the search to a subset of head arguments.

A third optimization is applicable in the case of a non-simply recursive procedure p. We now discuss this optimization although it is not useful for *quicksort*. For a non-simply recursive procedure p, there will be n partial modes to start with, where n is the arity of p, and where the i^{th} mode requires the i^{th} argument to be $+$, and leaves others unconstrained. When extending a partial mode which requires the i^{th} argument to be $+$, there is no need to refine any argument position less than i. It is enough to consider the argument positions from $i+1$ to n. The reason is that when extending any partial mode which maps some $k < i$ to $+$, we would have considered the k^{th} argument if there was a need. This optimization once again reduces the search for permissible modes, without missing any permissible modes. Without this optimization, for each partial mode that requires one argument to be input, we may have to consider all other arguments. Thus for n partial modes, we would potentially consider $n(n-1) = n^2 - n$ possibilities. With this optimization, we would have to consider at the most $\Sigma_{i=1}^{n-1} = (n^2 - n)/2$ possibilities. This does not affect the worst-case complexity but helps in the average case.

Example 5.16

```
1.  qsort([], []).
2.  qsort([X|Xs], Ys) :-
        partition(Xs, X, L, B),
        qsort(L, L1),
        qsort(B, B1),
        append(L1, [X|B1], Ys).
3.  partition([], K, [], []).
4.  partition([X|Xs], Y, [X|Ls], Bs) :-
        X =< Y,
        partition(Xs, Y, Ls, Bs).
5.  partition([X|Xs], Y, Ls, [X|Bs]) :-
        X > Y,
        partition(Xs, Y, Ls, Bs).
6.  append([], X, X).
7.  append([X|Xs], Y, [X|Ans]) :-
        append(Xs, Y, Ans).
```

□

The permissible modes of *append* (derived as before) are: $append(+, \perp, \perp)$ and $append(-, -, +)$. The built-in predicates `=<` and `>` require both their arguments to be ground.

Consider *partition* next. It has three clauses and hence, we initialize the set of permissible modes for *partition* to a singleton set containing the mode $\{partition(\perp, \perp, \perp, \perp)$, which does not constrain any argument of p, *i.e.*, maps them to the modality $\perp$. Clause 3 is a unit clause and hence imposes no constraints. *Partition* is simply recursive and arguments 1 and 3 of clause 4 are less general than the corresponding arguments of the recursive call. Hence, the set of partial modes PM for clause 4 is $\{partition(+,\perp,\perp,\perp), partition(\perp,\perp,+,\perp)\}$. Next, obtain the mode intersection of PM and $\{mode_{p_\perp}\}$ to be used as the partial modes for clause 4. This turns out to be the same as PM. Now we extend each of these modes.

Consider the partial mode $partition(+,\perp,\perp,\perp)$ first. After head unification, the available variables are {X,XS}. The first built-in goal X =< Y cannot be scheduled because Y is not in the set of available variables. The recursive call is the only one schedulable and so schedule it. Find the output variables of the recursive call. Since only the first argument is ground, no other arguments of partition will become ground. Thus, the set of available variables does not change after scheduling the recursive call. Hence, we need to refine a useful head argument so that X =< Y can be scheduled. Only the second argument Y qualifies and thus the partial mode $partition(+,\perp,\perp,\perp)$ is refined to $partition(+,+,\perp,\perp)$. This does not invalidate the previous scheduling of the recursive call since the set of variables {Y} in the newly refined head argument 2 covers the set of variables {Y} in the corresponding argument of the recursive call. With this refinement, we can schedule the built-in goal X =< Y. There are no more goals to be scheduled. Thus, $partition(+,+,\perp,\perp)$ is a permissible mode for clause 4.

As mentioned before, the partial ordering derived for clause 2 is just a constructive proof of the well-modedness of that clause. Notice that there is really no data-dependency between the two goals of clause 2 of partition; the goals may be solved in parallel or in any order. A moment's reflection will show that the call to a built-in goal must be solved before the recursive call because of control dependency. If the built-in predicate X =< Y fails, the whole clause fails and there is no need to execute the recursive call. Even if the built-in is not expected to fail, the granularity is too small to have two threads of execution, one for the built-in and another for the recursive call. Executing built-ins before others and solving deterministic goals before non-deterministic ones are some of the heuristics that may be used in deriving control flow based on permissible modes.

Consider the second partial mode $partition(\perp,\perp,+,\perp)$. Once again, only the recursive call is schedulable; scheduling it does not add more variables to the available set. Trying to extend the partial mode succeeds; the second argument (Y) is refined to input. As in the previous case, this causes no backtracking and another permissible mode for clause 4 is $partition(\perp,+,+,\perp)$

The permissible modes of clause 4, $\{partition(+,+,\perp,\perp), partition(\perp,+,+,\perp)\}$ are also the "running" permissible modes of procedure *partition*. The partial permissible modes of clause 5 are $partition(+,\perp,\perp,\perp)$ and $partition(\perp,\perp,\perp,+)$. We can use the running modes of the procedure to constrain the partial permissible modes of clause 5. As mentioned before, obtain the mode intersection of the set of running modes of the procedure and the set of partial modes of clause 5.

The new partial modes of clause 5 are $\{partition(+,+,\perp,\perp), partition(+,+,\perp,+), partition(+,+,+,\perp), partition(\perp,+,+,+)\}$. We now extend these partial modes, starting with the minimal ones.

The extension of the partial mode $partition(+,+,\perp,\perp)$ is itself and hence it is a permissible mode. The partial modes $partition(+,+,+,\perp)$, $partition(+,+,\perp,+)$ need not be extended nor retained, since these are more defined than the permissible mode *partition*$(+,+,\bot,\bot)$. This leaves the partial mode *partition*$(\bot,+,+,+)$. The extension of *partition*$(\bot,+,+,+)$ is itself and there are no more partial modes left.

Thus the permissible modes of *partition* are: *partition*$(+,+,\bot,\bot)$ and *partition*$(\bot,+,+,+)$. These are also minimal. This simply confirms the possible ways in which *partition* can be used. The first argument can be partitioned about the second argument (the pivot) into a list of numbers (the third argument) not greater than the pivot and a list of numbers (the fourth argument) greater than the pivot. Alternatively, given the pivot, and two lists of numbers greater than and not greater than the pivot, the two lists can be non-deterministically merged and returned as the first argument. The relative order of numbers from the third (fourth) argument will be preserved in the first argument.

Finally, the permissible modes of *qsort* are derived as follows. There are two partial modes, namely, *qsort*$(+,\bot)$ and *qsort*$(\bot,+)$. We extend *qsort*$(\bot,+)$ first.

The set of available variables after head unification is {`Ys`}. This covers the input variables a permissible mode *append*$(-,-,+)$ of the fourth goal. Scheduling it adds the variables {`L1,X,B1`} to the available variables set. Both the second and third recursive calls are the only ones schedulable and hence are scheduled. Using abstract interpretation, the output variables of the second and third goals are found to be {`L,B`}. These are added to the available variables set and the first goal *partition* can now be scheduled. Thus, *qsort*$(\bot,+)$ is extended to a permissible mode *qsort*$(-,+)$ and the partial ordering or body goals used in deriving this permissible mode is $\{4<3, 4<2, 4<1, 3<1, 2<1\}$.

Extending *qsort*$(+,\bot)$ results in a permissible mode *qsort*$(+,-)$ and the familiar ordering of body goals. Note that our method would derive these permissible modes and orderings even if the body goals of *qsort*, and *partition* are permuted and it will take exactly the same time.

We now prove the correctness of our algorithm.

5.6 Proof of Correctness

The first step is to prove that the algorithm terminates. The next step shows that if the algorithm terminates and the mode $mode_p$ derived by the algorithm for a procedure p is not $\top$, then $mode_p$ is a permissible mode.

5.6.1 Termination

The permissible modes algorithm derives partial permissible modes first and then extends each partial permissible mode to a permissible mode. Proposition 5.4 proves the correctness of the derivation of partial permissible modes. Definition 5.13 simply identifies a subset S of the argument positions of the head literal of a clause and create a partial permissible mode for each argument in S. Clearly, this procedure terminates and takes time linear in the number of head arguments (or arity) of the clause.

We have to show that the algorithm for extending a partial mode to a permissible mode for non-mutually recursive procedures, defined in Figures 5.3 and 5.4, terminates. Likewise, the termination of the mutually-recursive procedures's version of the algorithm in Figure 5.5

must also be shown. We start with the termination proof for the algorithm in Figures 5.3 and 5.4.

Theorem 5.3

Let p be procedure with dfs number k and c be a clause in p. Let PM denote an array that contains the permissible modes for all procedures with dfs numbers less than k and also contains the partial permissible $mode_c$ of clause c. Then the call *extend_partial_mode*($mode_c$, PM) terminates. □

Proof

The function *extend_partial_mode* computes the set of input variables of the mode $mode_c$ and binds it to $avail_0$. It creates some other trivial bindings and calls another function *epm* just once. Hence, we have to show that the call to *epm* terminates.

The function *epm* is recursively defined. It calls *epm_iter* and itself recursively. In order to show the termination of *epm*, it is sufficient to show that

- each call to *epm_iter* terminates and
- in successive calls to *epm_iter*, either the set of unscheduled literals becomes smaller or the set of available variables becomes larger.

since there are only finite number of body literals and clause variables.

The termination of function *epm_iter* is easy to establish. It is not a recursive function and all the functions it calls such as *useful*, *update*, set union, set subtraction terminate.

Each call to *epm_iter* either returns $\top$ in which case *epm* terminates or changes the current state and returns the changed state. The possible computations of *epm_iter* are discussed below.

The set of available variables in the current state covers all the input variables of one or more unscheduled literals; in which case, these literals are added to scheduled set and removed from unscheduled set and thus the unscheduled set becomes smaller in the next state.

If the unscheduled set does not become smaller, then the available variables did not cover all the input variables of any of the unscheduled goals. This means that *epm_iter* will either find a useful head argument and refine it to input or return $\top$ signifying failure to extend the partial mode. If *epm_iter* did find a useful argument, then the set of available variables in the next state is larger than the set of available variables in the current state, by definition of usefulness.

□

Note that in the above proof, we did not use the fact that p was non-mutually recursive procedure. This is because the function *extend_partial_mode* will terminate for both mutually recursive and non-mutually recursive cases. In the former case, the result will be just an approximation of a permissible mode. These approximations will have to be successively refined until they stabilize. This is the purpose of the "wrapper" algorithm given in Figure 5.5.

The proof of termination of the algorithm for extending the partial modes of mutually recursive procedures (Figure 5.5) is given below.

Theorem 5.4

Let P be a program and SCC be a maximal set of mutually recursive procedures in P (*i.e.,* SCC is a maximal strongly connected component of the call graph of P). Let *CurApprox* be an array of sets of partial permissible modes of procedures in SCC. Let k be the dfs number of the procedures in SCC and PM denote an array of the permissible modes for all procedures with dfs numbers less than k. Then the call *epm_mr*(SCC, *CurApprox*, PM) terminates. □

Proof

Function *epm_mr* simply adds the partial permissible modes *CurApprox* to PM and calls *epm_mr_iter* just once. Hence, we must prove the termination of *epm_mr_iter*.

Function *epm_mr_iter* calls the function *extend_partial_mode* and itself recursively. The termination of *extend_partial_mode* has already been shown. It remains to be shown that *epm_mr_iter* will call itself only finitely many times.

The current approximation of permissible modes for the procedures in SCC is a two-dimensional finite array of approximate permissible modes. Function *epm_mr_iter* refines each of these finite number of approximate modes (keeping others constant) by calling *extend_partial_mode*. The next approximation is the result of updating once all the current approximate modes.

Function *epm_mr_iter* recursively calls itself only when the next approximation of permissible modes for the the procedures in SCC is not equal to the current approximation. This means at least one approximate mode has changed in the next approximation.

A refinement of a mode can map it to $\top$ in which case it is removed from the set of permissible modes of procedure p by *epm_mr_iter*. An approximation of a permissible mode can be refined/extended to a better approximation only a finite number of times, since an extension of a consistent mode simply changes the modality of one of the arguments from $\perp$ to $+$ or $-$ and the arity of procedures is finite. □

5.6.2 Soundness

According to proposition 5.1, the mode intersection of the sets of permissible modes of the clauses of a procedure results in permissible modes for the procedure. Hence, to prove the soundness of the algorithm for deriving permissible modes, we just need to establish the soundness of the algorithm for extending a partial mode of a clause to a permissible mode. The definition of well-modedness and permissible mode are repeated below for convenience.

Definition 5.15

Let $c = h \leftarrow b_1, \ldots, b_k$ be a clause and $mode_h$, $mode_{b_1}, \ldots, mode_{b_k}$ be the modes of the head and the body predicates of c respectively. Let $\leq$ be a partial ordering of the body goals $b_1, \ldots, b_k$ and let $pred_{\leq}(i) = \{j \mid j \leq i\}$ denote the set of predecessors of literal i in the partial order $\leq$.

Clause c is *well-moded* with respect to modes $mode_h$ and $mode_{b_1}, \ldots, mode_{b_k}$ (or $mode_h$ and $mode_{b_1}, \ldots, mode_{b_k}$ is a *well-moding* for clause c) iff there exists a partial ordering $\leq$ on the literals $b_1, \ldots, b_k$ such that for $1 \leq i \leq k$,

$$invars(b_i, mode_{b_i}) \setminus invars(h, mode_h) \quad \subseteq \quad \bigcup \{outvars(j, mode_j) \mid j \in pred_{\leq}(i)\}.$$

□

Definition 5.16

Let c be a clause $h \leftarrow b_1, \ldots, b_k$. A mode m for clause c is a permissible mode, if it satisfies permissible mode constraints 1, 2 and 3 and there exist permissible modes $PM = \{mode_{b_1}, \ldots, mode_{b_k}\}$ for the body goals of c such that the clause c is well-moded with respect to mode m and PM. □

Proposition 5.5

Any extension of a partial mode using the algorithm in Figures 5.5 or 5.3 and 5.4 satisfies permissible mode constraints 1, 2 and 3. □

Proof

By definition, partial permissible modes satisfy permissible mode constraints 2 and 3. We will show that extension of a partial mode satisfies permissible mode constraint 1, which requires that the head literal and recursive body literals, if any, must have the same mode.

The function *extend_partial_mode* ensures that all recursive literals (if any) in the body of a clause have the same mode as the head literal before starting the iterative mode extension process. We will show that the function *epm_iter* maintains this invariance and thus satisfies permissible mode constraint 1. Then a simple induction on the number of states in the computation that extends a partial mode to a permissible mode completes the proof.

The only place where a mode is extended is in the function *epm_iter* and there are two possible refinements of a mode. When the current set of available variables does not cover the input variables of any of the unscheduled goals, the modality of a useful head argument is refined from $\perp$ to $+$. At the same time, the modes of recursive body literals, if any, are also refined from $\perp$ to $+$. Thus, the invariance is maintained.

The second possible refinement of a mode also occurs in *epm_iter*. Whenever a recursive literal is scheduled, it may generate some argument positions. For example, when a recursive literal p with an approximate permissible mode $p(+, +, \perp, \perp)$ is scheduled, we need to know the variables produced by p. Suppose that a groundness analysis shows that the groundness of the first two arguments of p implies the groundness of the fourth argument, then we can refine the mode $p(+,+,\perp,\perp)$ to $p(+,+,\perp,-)$. We do so and immediately update the modes of the head literal and all other recursive literals. Once again, the invariance that the head and recursive literals must have the same mode is maintained. □

Theorem 5.5

Let p be a non-mutually recursive procedure and $c = h \leftarrow b_1, \ldots, b_n$ be a clause of p. Let $mode_c$ be a partial permissible mode of c and PM be the array of permissible modes of procedure with a dfs number smaller than the dfs number of p. Then, *extend_partial_mode*$(mode_c, PM)$ returns either $\top$ or a permissible mode $mode_{c'}$ for c. □

Proof

We need to show that extending a partial mode $mode_c$ of c using the function *extend_partial_mode* results in a permissible mode for c. Proposition 5.5 shows that the permissible mode constraints hold for $mode_{c'}$. In particular, proposition 5.5 shows that all recursive literals (if any) and the head have the same mode when *epm* terminates successfully. Thus, all we need to show is that there exist permissible modes $PM_b = \{mode_{b1}, \ldots, mode_{bn}\}$ for the body goals with smaller dfs numbers than the dfs number of p such that PM_b and the $mode_{c'}$ together induce well-modedness for c.

For body literals with a dfs number less than that of procedure p, the array PM has the permissible modes. These are the ones used by the function *epm_iter* in scheduling goals. Therefore, existence of permissible modes for these goals is established. Thus, what remains to be shown is that there is a partial ordering of the body goals with induces well-modedness.

The function *extend_partial_mode* constructs the needed partial ordering of body goals via the functions *epm* and *epm_iter*. Let $state^q$ be the 7-tuple state $(unsched_q, sched_q, PO_q, PM^q, avail_q, when\_sched_q, q)$ of the q^{th} iteration of the function *epm*. The function *epm* maintains the following invariances at each state $q = 1, \ldots, q'$ where q' is the last state of computation.

- The set of available variables at iteration q is equal to the set of input variables of the head literal (according to its mode PM_0^q) union the output variables of the scheduled literals (if any) $sched^q$.
- If a literal j is scheduled at iteration $i+1$ (*i.e.*, $when\_sched_q(j) = i+1$), then the input variables of literal j are a subset of available variables of iteration i.
- If $(j,k) \in PO_q$ then $when\_sched_q(j) < when\_sched_q(k)$, *i.e.*, if j precedes k in the partial order at iteration q, then j was scheduled before k.

It is easy to check that these invariances are maintained at the initial state $q = 1$, by inspecting the function *extend_partial_mode*. Each subsequent state is created by the function *epm_iter* either by scheduling one or more literals, or by backtracking to a previous state. Once again, these invariances are maintained whether new goals were scheduled in a state or not. A simple induction on q establishes these invariances. This shows the existence of the partial ordering of body goals. The partial ordering together with the permissible modes that were actually used in scheduling body goals and the clause mode $mode_{c'}$ show the well-modedness of clause c. □

Theorem 5.6

Let P be a program and SCC be a maximal set of mutually recursive procedures and *CurApprox* be the partial permissible modes of the procedures in SCC. Let $PM_{p,i}$ denote a permissible mode of a procedure p where $1 \leq i \leq$ *arity of p* if the dfs number of p is less than that of a procedure in SCC or undefined if p is in SCC.

Then, $epm\_mr(SCC, CurApprox, PM)$ returns PM' where for each p in SCC PM'_p is either empty or a non-empty set of permissible modes for procedure p. □

Proof

Function *epm_mr* just updates PM with the initial approximate modes (partial permissible modes) of procedures in SCC and then calls function *epm_mr_iter*. The initial partial permissible modes of all procedures p in SCC are consistent by Proposition 5.4. The correctness of *epm_mr_iter* is proved by induction on the number of iterations.

Inductive Hypothesis: Assume that the current approximation of the permissible modes $CurApprox_p$ of each procedure p in SCC is consistent. Each permissible mode $CurApprox_{p,i}$ in the current approximation of permissible modes of p is refined for each clause c in the procedure p and the mode intersection of the refinements of all clauses in p is taken. Recall that the definition of mode intersection removes inconsistent modes from the intersection. Since *extend_partial_mode* and mode intersection are sound, then by induction on the number of clauses in procedure p, the operation
$mode\_intersection(\{extend\_partial\_mode(Curapprox_{p,i}, c, PM) \mid \text{clause } c \in p\})$
returns a consistent set of approximate permissible modes, which may be empty.

We have thus shown that each iteration of *epm_mr_iter* refines a consistent set of approximate permissible modes for each procedure p in SCC and maintains the consistency. The set of permissible modes refined at an iteration may be empty or non-empty; if empty, the theorem is proved. Consider the non-empty case. We have already shown termination. Since the function *epm_mr_iter* terminates only when the previous approximation and current approximation of permissible modes for all procedures p in SCC are equal, the scheduling of any call to a procedure p in SCC occurring in any of the clauses of p must have used a permissible mode. This, combined with the fact that PM initially contains permissible modes for all procedures with a lower dfs number than a procedure p in SCC proves the permissibility of the modes PM_p of p.

□

5.7 Complexity Analysis

The worst-case complexity of the partial mode extension algorithm *extend_partial_mode* (Figure 5.3) is bounded by the time taken for the groundness analysis and a polynomial in following parameters: n is the number of variables of a clause, l is the number of literals of a clause, a is the arity of a clause, and m is the number of permissible modes of a body goal. Under reasonable assumptions, the worst-case time complexity of the mode intersection operation and that of the permissible modes derivation algorithm for mutually recursive procedures are also polynomial in the following parameters, in addition to the ones mentioned above: c is the number of clauses in a procedure, and p is the number of procedures in a maximal set of mutually recursive procedures.

The partial mode extension algorithm *extend_partial_mode* is parametrizable by an abstract interpretation scheme for groundness analysis. This is because the algorithm needs to know the set of output variables of a body goal when scheduled. One can use the groundness analysis proposed by Debray [29] (as part of his sequential mode analysis scheme) or a more precise analysis based on *Pos*. Debray's groundness analysis has polynomial time complexity if the maximum arity of predicates is bounded. Groundness analysis of Marriott and Sondergaard [56] using *Pos* likewise has polynomial time complexity if the maximum number of clause variables is bounded. Both analyses are efficient in practice, but *Pos* is more precise than the other. In the following complexity analysis of our partial mode extension algorithm, we assume a polynomial time groundness analysis. The time taken for groundness analysis of a program and a given goal is denoted by T_g.

5.7.1 Analysis of *extend_partial_mode*

The partial mode extension algorithm *extend_partial_mode* has two clearly identifiable parts: function *epm_iter* which either schedules a goal, or refines the modality of a head argument from $\perp$ to $+$, and function *epm* which calls *epm_iter* repeatedly until all goals are scheduled or *epm_iter* returns $\top$ signifying failure to extend a partial mode. The time complexity of *extend_partial_mode* is determined by the number of times *epm_iter* is invoked multiplied by the time taken by *epm_iter*.

Function *epm_iter* has three clearly identifiable parts. The first part schedules goals, whenever possible. The second part identifies a head argument (if any) that needs to be refined from $\perp$ to $+$. The third part backtracks as and when needed.

The first part schedules a goal, if its input variables (according to one of its permissible modes) is covered by the set of available variables. The scheduling of a literal involves set union, set subtraction, updating an array of scheduled literals, and adding the output variables of a scheduled literal to the set of available variables. We represent sets as ordered lists, and set union and intersection can be done in time linear in the sizes of the sets. Depending on the set, this translates to a worst-case time complexity of $\mathcal{O}(l)$ or $\mathcal{O}(n)$ where l is the number of body literals and n is the number of clause variables. The time taken for finding the output variables of a goal is, as mentioned before, T_g. Updating the array of scheduled literals can take $\mathcal{O}(l)$ time. Thus the time for scheduling a goal is $\mathcal{O}(max(T_g, l, n))$.

The second part of *epm_iter* finds a useful head argument (if it exists) whose modality is changed from $\perp$ to $+$. The definition of *useful* involves computing a least j such that the j^{th} head argument is unconstrained and that argument has variables that are needed to schedule one or more goals and these variables are not already available. Finding an argument with modality $\perp$ can be done in time $\mathcal{O}(a)$ where a is the maximum arity of a predicate. Each body goal may have m permissible modes and hence the potential input variables of each body goal can be computed in time $\mathcal{O}(ma)$. This need be computed just once before *epm_iter* is ever called, and looked-up in *epm_iter*. Hence, we will not include it in the complexity of *epm_iter*. The size of the set of input variables of a goal is bounded by the number n of clause variables, and the intersection of the set of variables in head argument and the sets of input variables of a body goal can be done in time $\mathcal{O}(a+n)$. Since all body goals may have to be considered for finding a useful head argument, the time complexity of *useful* is $\mathcal{O}(l(a+n)) = \mathcal{O}(la+ln)$

The third part of *epm_iter* checks if backtracking is necessary. This involves taking the intersection of the set of scheduled literals, and the set of recursive literals whose modes have now been revised. Clearly, this can be done in time $\mathcal{O}(l)$ where l is the number of body goals.

The time complexity of an invocation of *epm_iter* is the maximum of the time complexities of its three parts, which is $\mathcal{O}(max(T_g, la+ln))$ where T_g is the time for groundness analysis, l is the maximum number of body goals, n is the maximum number of clause variables, and a is the maximum arity of a predicate.

The next step is to derive the number of times *epm_iter* may be invoked by *epm*. If there is no backtracking at all, then each call to *epm_iter* must schedule one literal and there are only l such literals. But backtracking complicates the picture by unscheduling scheduled literals. Note that only the scheduling of a recursive literal may cause backtracking. Consider a recursive literal r scheduled first (*i.e.,* before all other goals) whose right place is actually last in the ordering of body goals. This may not be discovered until $l-2$ more literals have been scheduled and the last one is about to be scheduled. Thus, the wrong scheduling of r as the first goal, may cause $l-2$ more calls to *epm_iter*. When backtracking occurs, the set of available variables becomes larger due to refining a useful head argument. If r is still schedulable as the first goal even after backtracking, then we would not have backtracked to r in the first place. In other words, if the set of available variables in the backtracked state covered the input variables of r according to the extended partial mode, the refinement did not invalidate the original scheduling. Thus each backtracking causes the set of available variables to grow and moves a wrongly scheduled literal at least one step towards its right place in the ordering of body goals. Wrong scheduling as the first goal may cause $l-1$ calls to *epm_iter* and similarly wrong scheduling as the i^{th} goals causes $l-i$ calls to *epm_iter*. There can be at most $l(l-1)/2$ such calls and thus the maximum number of calls to *epm_iter* will be $\mathcal{O}(l^2)$.

Notice that when backtracking occurs, we backtrack to the earliest wrongly scheduled goal and the position of the earliest wrongly scheduled goal can vary from one to $l-1$ and thus the wasted iterations can at most be $\mathcal{O}(l^2)$.

Thus, the worst-case time complexity of partial modes extension algorithm is $\mathcal{O}(l^2) \times \mathcal{O}(max(T_g, la+ln))$ which is $\mathcal{O}(max(l^2T_g, l^3a+l^3n))$. The groundness analysis of Debray [29] has polynomial-time worst-case complexity assuming a bound on the maximum arity of predicates, which is reasonable. Debray's abstract domain does not have the condensation property discussed earlier in this chapter. Hence if it is used with our the partial mode extension algorithm, we will have to compute fixed-points for each goal and each call substitution. On the other hand, if we use Marriott and Sondergaard's *Pos* domain which has the condensation property, then the fixed-point representing the abstract meaning of a program need be computed just once. The worst-case time complexity for Debray's analysis is exponential in the arity of a predicate but polynomial if we assume bounded arity. Likewise, groundness analysis using *Pos* domain has worst-case complexity exponential in the number of clause variables but assuming bounded number of clause variables results in polynomial-time complexity.

The main point of the above discussion is to show that our algorithm for extending a partial mode for a clause has polynomial-time worst-case complexity for the explicitly specified computation, and when coupled with a practical groundness analysis method, results in an efficient algorithm.

5.7.2 Complexity of Mode Intersection

Once the permissible modes for all clauses of a procedure are computed, they are combined to yield the permissible modes of a procedure. In theory, the mode intersection of the permissible modes of the clauses of a procedure may result in exponential number of permissible modes for a procedure, *i.e.*, mode intersection of m permissible modes each of s clauses of a procedure may yield m^s permissible modes for a procedure. The worst-case scenario assumes that the set of permissible modes of a clause is independent of that of another clause of the same procedure, and their intersection could result in all possible combinations of the two sets of modes. In practice, however, the modes of the clauses are not independent of each other, and the number of permissible modes for a procedure is less than the arity of the procedure.

There are two reasonable assumptions we can make. One is that the maximum arity of a predicate is bounded, which implies that the maximum number of permissible modes is also bounded (but very large). Another assumption is that the number of minimal permissible modes of procedure is bounded by the arity of that procedure. Under these assumptions, the worst-case time complexity of mode intersection is bounded by ca^2, where c is the maximum number of clauses in a procedure and a is the maximum arity of a procedure.

5.7.3 Complexity of Function *epm_mr*

The time complexity of the function *epm_mr* which extends the partial permissible modes of mutually recursive procedures to permissible modes is determined by the time complexity of mode intersection and that of the function *extend_partial_mode*.

Function *epm_mr* calls *epm_mr_iter* just once, and *epm_mr_iter* calls *extend_partial_mode* recursively. Each call to *extend_partial_mode* refines the mode of at least one argument of one clause in the set of clauses of mutually recursively predicates. Let the maximum number of clauses in a procedures be c and the maximum number of procedures in a maximal strongly connected component be p. Then, there can be at most $pcma$ calls to *extend_partial_mode*, where a and m are the same as before, *i.e.*, a is the maximum arity of a predicate, and m the maximum number of permissible modes for a predicate.

To find the number of times mode intersection is computed by *epm_mr_iter*, we must know the number of times *epm_mr_iter* calls itself. Each call to *epm_mr_iter* must update at least one permissible mode of at least one predicate in the strongly connected component. Let the maximum number of predicates in a strongly connected component be p. Then the maximum number of calls to *epm_mr_iter*, and the maximum number of mode intersections computed by function *epm_mr_iter* is pa. Since the worst-case time complexity of mode intersection is exponential in the number of clauses, and that of *extend_partial_mode* is polynomial in the number of literals, clause variables, the arity of the head predicate, and the maximum number of permissible modes for body goals, the time complexity of *epm_mr* is dominated by that of mode intersection. However, if we assume that the number of permissible modes per procedure is bounded by the arity of the procedure, then mode intersection takes ca^2, ane hence the worst-case complexity of *epm_mr_iter* is $(pa \times ca^2) + (pcma \times (\mathcal{O}(l^2) + T_g)) = \mathcal{O}(pca^3 + pcma(l^2 + T_g))$ where T_g is the complexity of groundness analysis.

Analysis of a number of programs show that the number of permissible modes for a procedure is less than the arity of the procedure and is usually less than four. The arity of predicates do not increase with the size of the program and is bounded by a small constant. Likewise, the number of iterations needed to reach convergence in the case of mutually recursive procedures is also four or less. Since the assumption of bounded arity predicates and bounded permissible modes per procedure appear to hold, the analyzer has polynomial time complexity in practice.

5.8 Empirical Measurements

The algorithm for deriving permissible modes has been implemented, using the groundness analysis of Chapter 3. In this section, we discuss the results of an empirical study.

Table 5.1 lists some statistics about the analysis of 22 programs. The statistics are: the total number of permissible modes derived by our algorithm for the whole program, the number of procedures, the maximum number of permissible modes for a procedure derived by our algorithm, the number of clauses, the time for permissible modes derivation, the part of the time spent in the groundness analysis component of the analyzer, and finally the number of calls to the groundness analyzer when deriving permissible modes.

Execution times are in seconds, as measured on an HP 735 workstation using Sicstus Prolog 2.1.8 in bytecode emulation mode.

Many of the programs are the ones used to measure the precision and efficiency of the sharing and groundness analyzer of Chapter 3. The benchmarks include small programs such as `quicksort`, `perm`, `perm1`, `ackermann`, `merge`, `mergesort`, and `n-queens` and medium-size programs such as `Press1` (a symbolic algebra package), `Boyer`, and `Peep1` (peephole optimizer for Stonybrook Prolog).

The maximum number of permissible modes per procedure derived by our method ranges from one to four for the programs tested. The number of permissible modes is less than the maximum arity of a procedure in all cases. Thus, the mode intersection of the modes of clauses of a procedure does not result in an exponential number of permissible modes for a procedure. The algorithm fails to derive permissible modes in some cases; these are due to our definition of input and output modes as denoting fully ground terms. The

Program	Tot PM	No of Procs	Max PM per proc	CL	Calls to Gr Anl	Total Time	Gr Anl Time
Ackermann	1	1	1	4	9	0.52	0.37
Perm	4	2	2	4	5	0.15	0.07
Perm1	4	2	2	4	5	0.18	0.12
Naiv-rev	4	2	2	4	5	0.16	0.07
Ndmerge	2	1	2	4	4	0.23	0.18
Quicksort	6	3	2	7	13	0.51	0.38
Mergesort	6	3	2	9	10	0.47	0.29
N-queens	8	5	2	9	13	0.34	0.15
Iso	8	6	2	11	14	0.65	0.34
Serialize	7	6	3	12	18	1.42	1.18
Grammar	6	6	1	15	15	0.31	0.18
Path	3	2	2	18	5	0.35	0.20
Plan	19	16	2	29	35	2.48	1.94
Dcg1	19	13	2	31	46	2.11	1.35
Browse	17	16	1	32	68	5.03	4.16
Gabriel	28	20	3	45	66	4.84	4.01
Bid	24	21	2	52	50	2.37	1.49
Disj	36	34	2	63	69	4.59	3.67
Boyer	32	25	2	139	92	18.52	16.22
Semigroup	23	20	2	46	67	34.02	32.82
Peep1	20	18	2	227	69	51.69	42.69
Press1	75	48	4	158	164	61.51	57.91

Table 5.1: Minimal Modes Derivation

limitation and proposed extensions are discussed in the last chapter.

For the programs tested, the modes derived by our algorithm are minimal. The permissible modes algorithm derives at most n modes per clause, where n is the arity of the head predicate of the clause. A natural question to ask is: What fraction of the set of permissible modes is derived by the algorithm for the programs tested? For small programs such as `quicksort`, `perm`, `mergesort`, `path`, `n-queens`, and `iso`, the algorithm derives all minimal modes. The textual ordering of body goals does not affect either the number of minimal modes derived or the time taken for the analysis.

For larger programs, it is hard to say how many permissible modes were missed by our method. This because it too expensive to compute all minimal permissible modes, and then check how many of these were missed by our method. Our method did not find any permissible mode for one clause in programs `serialize`, `browse`, and `semi20`, for two clauses in `peep1`, for three clauses in `boyer2`, and for nine clauses in `press1`. These misses are either due to an imprecision in the implementation groundness propagation or due to the standard Prolog practice of using partially instantiated structures. When compared to the total number of modes (352) derived by the algorithm for all the programs tested, the number of misses is not unreasonable. However, only three of these misses cause the analyzer to abort and these happen to be the top-level goals. In our implementation, if the analyzer fails to derive at least one permissible mode for a clause c but is able to derive permissible modes for one or more clauses in the same procedure, it warns the user about missing modes for clause c but proceeds with the rest of the analysis using modes from other clauses. Only when it fails to derive at least one permissible mode for a procedure p, and a permissible mode for p is needed in some clause c (because a call to p appears in clause c), does the analyzer abort. The handling of partially instantiated data structures is an extension of our current proposal and is discussed in the last chapter.

In general, our algorithm may derive non-minimal modes where an analysis using a fixed control flow may find minimal modes, and vice-versa. This does not imply that our permissible mode derivation algorithm and a sequential mode analysis are comparable in their performance and usefulness. We define the notion of permissible modes and use it to derive data flow and control flow simultaneously. A sequential mode analysis algorithm, on the other hand, simply describes what modes may arise in a sequential computation; in such an analysis, there is no notion of permissible modes to ensure desirable data flow.

We now discuss the efficiency of the implementation. The execution times of the analyzer are more or less proportional to the size of the program analyzed, except for the programs `semi20`, `peep1`, and `press1`. The variation in the times for programs other than `semi20` and `press1` are due to the presence or absence of built-in evaluable predicates such as arithmetic and relational predicates. These impose strong directionality on their arguments, and their permissible modes are input to the system and do not involve fixed-point computations.

The higher analysis times for `semi20`, `press1` and `peep1` have two main reasons. First is the use of the sharing and groundness domain of Chapter 3 which does not have the condensation property. When extending a partial mode, the scheduling of a literal leads to computing the output variables of that literal. Finding the output variables of a literal entails fixed-point computations, if a non-condensing domain is used for groundness analysis. These fixed-point computations may be shared, if they involve computing the abstract success substitution of a goal for the same abstract call substitution more than once. However, the abstract call substitutions of a predicate p that arise in the course of extending a partial mode are not the same most of the time.

The time spent by our algorithm in deriving permissible modes for a procedure is dominated by the time spent in the abstract interpreter for groundness analysis. The time spent in groundness analysis as a percentage of the total time for deriving permissible modes ranges from 44% for small programs to 96% for `semi20` and 94% for `press1`. For other programs such as `peep1`, `disj`, and `boyer`, the time spent in groundness analysis is more than 80% of the total time for permissible mode derivation. If we consider the time for deriving permissible modes ignoring the time for groundness analysis, the execution times are almost linear in the size of the program. The number of calls to the groundness analyzer (each of which entails a fixed-point computation) is two to four times the number of procedures in a program. With a condensing domain for groundness analysis such as *Pos*, there will be exactly one fixed-point computation per procedure. Hence, it appears that a condensing domain such as *Pos* for groundness analysis should be very useful since it avoids repeated fixed-point computations.

The second reason for the high analysis times for `press1` and `peep1` is that these programs have many sets of mutually recursive predicates. Recall that the time complexity for deriving permissible modes for mutually recursive procedures is $\mathcal{O}(pca^3 + pcma(l^2 + T_g))$ compared to $\mathcal{O}(ca^2 + (l^2 + T_g))$ for a non-mutually recursive procedure, where p is the number of procedures in a strongly connected component, c is the number of clauses in a procedure and m is the number of permissible modes for p.

The permissible mode analysis presented in the previous sections is actually a framework with some parameters in place. The parameters are now discussed, which leads to the main limitation of our definitions of input and output modes.

5.9 Parameters of the Permissible Mode Analysis

The parameters of the framework are as follows.

- In the definition of input and output modes, we equated input and output modalities with ground terms; input argument must be ground on call and output argument must be ground on success. This is done to illustrate the derivation of permissible modes and to have a concrete implementation whose performance can be evaluated. We realize that it is too restrictive. The example below illustrates the limitation of the current analysis in deriving permissible modes.

  The framework itself does not depend on the exact definitions of input and output arguments. A combination of modes and regular types have been proposed for Prolog programs by Bruynooghe and Janssens [12], Bronsard, Lakshman, and Reddy [10], and Somogyi [74]. Polymorphic type systems have been proposed Zobel [93], Lakshman and Reddy [49], among others. It is possible to redefine permissible modes and partial modes to allow finer specification of input-output modes in combination with types, using one of the above systems with necessary modifications. This is an extension we intend to pursue.

- The second parameter to the permissible mode derivation algorithm is the groundness analysis. As mentioned before, any abstract interpretation scheme for groundness analysis may be used as part of the permissible modes derivation algorithm. It would be better to use a condensing groundness analysis in order to minimize the number of fixed-point computations. When input-output modes are redefined as suggested above, the analyzer will need derive types and modes instead of groundness of variables.

- The partial modes extension algorithm deterministically chooses a useful head argument for refinement. The current implementation makes no special effort in choosing a head argument for refinement; the head arguments are maintained as an ordered list, the list is examined sequentially. Changing the search order does not seem to have much effect when the modes are coarse and for the programs tested. However, it would be worth exploring various options when the input-output modes are redefined to include types.

Example 5.17

The procedure *pairs_list* takes a list of terms as the first argument and constructs a list of pairs of terms, such that the first component of each pair is a term from the first list and the second component of each pair is a free variable. For example, `pairs_list([a,b,c], A)` will bind `A` to `[pair(a,X),pair(b,Y),pair(c,Z)]`.

```
1.  pairs_list([], []).
2.  pairs_list([A|As], [pair(A,A1)|Rest]) :-
        pairs_list(As, Rest).
3.  p([]).
4.  p([pair(X,X1)|Rest]) :-
        q(X, X1),
        p(Rest).
5.  r(A) :-
        pairs_list(A, B),
        p(B).
```

□

Our algorithm will derive the permissible modes *pairs_list*(+,?) and *pairs_list*(-,+) for *pairs_list*. Notice that in the former mode, the second argument of *pairs_list* does not become a fully ground term. It will, however, be a finite list of pairs. Suppose another procedure *p* is intended to take the list of partly instantiated pairs and produce bindings for the uninstantiated variables (via another procedure *q*). The permissible mode for *p* is *p*(+). Since *pairs_list*(+,?) does not produce ground bindings for its second argument, our method will fail to derive permissible modes for r. The problem here is that in order for *pairs_list* and *p* to terminate, it is enough if the first argument is a complete list, *i.e.*, the tail is not a variable. It does not matter if the elements of the lists are uninstantiated or partly instantiated.

This limitation is due to our definition of input and output. Instead of requiring the input and output arguments to be fully ground terms, we need finer characterization, based on a system of types and modes.

5.10 Derivation of Control Flow using Permissible Modes

The mutual dependency of data flow and control flow analyses was illustrated in the introduction to this chapter. The notion of permissible modes is used to characterize certain data flow as desirable and disallow certain others, and to break the cyclic dependency between data flow and control flow. The permissible modes derivation algorithm constructs a partial ordering of the body goals to show well-modedness. However, such a partial ordering is only one of many that are possible.

Recall procedure *partition* of the *quicksort* example 5.16 (Page 111). The built-in goal `X =< Y` and the recursive goal `partition(Xs, Y, Ls, Bs)` can be executed in parallel or in any order. Since the execution of built-ins such as `X =< Y` may fail, it is better to solve the built-in before the recursive call.

Furthermore, it is possible to detect, via static analysis, deterministic computations in logic programs using the method proposed by Debray [31]. A call to *partition* conforming to the mode $partition(+,+,-,-)$ can be shown to be deterministic. Thus, permissible modes which induce to deterministic computations are to be preferred to those that do not.

The clauses of a procedure may have a unique control flow associated with a mode, even though the procedure itself may be non-deterministic with respect to that mode.

Example 5.18

Consider the permutation program, reproduced here for convenience.

```
1.  perm([], []).
2.  perm([X|Xs], [Y|Ys]) :-
        delete([X|Xs], Y, Zs),
        perm(Zs, Ys).
3.  delete([A|Rest], A, Rest).
4.  delete([A,B|Bs], X, [A|Rest]) :-
        delete([B|Bs], X, Rest).
```

□

The procedures *perm* and *delete* are both non-deterministic in the modes $perm(+,-)$ and $delete(+,-,-)$, because they can produce multiple solutions for a query such as `perm([1,2,3,4], A)` which satisfies the mode $perm(+,-)$. These solutions may also be produced in any order in a parallel implementation. However, when we derive a minimal permissible mode $delete(+,-,-)$ for *delete*, it induces only one minimal permissible mode $perm(+,-)$ which in turn induces a unique control flow for *perm*; the recursive call of *perm* must be executed after the call to *delete*, for the mode $perm(+,-)$. Likewise, the permissible mode for *delete* induces a unique control flow.

Thus non-determinism and unique control flow are orthogonal attributes of a procedure. When more than one control flow is possible for a clause and a permissible mode, we need a mechanism to evaluate the alternative control flows. The possibility of multiple control flows for the same mode signifies search-type programs. One may have to use heuristics such as scheduling deterministic calls before non-deterministic ones, and preferring goals which generate more variables than those that do not.

Consider two goals $p(X,Y)$ and $q(X,Y)$. Assume that both can be scheduled, both will terminate, and variable X is bound to a ground term, variable Y is free, and p and q may create m bindings for Y. If m is smaller than n, it may be better to solve p first. Likewise, if n is smaller than m, it may be better to solve q first. The idea is to constrain the search space for solutions. The solutions and the number of solutions remain the same regardless of the order in which p and q are solved. In the presence of infinite derivations, the system may not find all the solutions. The programmer may have to specify the control flow for search-type programs. Even in search-type programs, not all procedures may involve search. Thus we expect minimal programmer intervention.

The current implementation derives permissible modes and a partial ordering of body literals for each permissible mode. As discussed above, this suffices for procedures that have unique data flow per call pattern. Non-search-type procedures appear to have a unique control flow associated with a permissible mode, even though the procedure may be nondeterministic. For search-type programs, the partial ordering constructed by our method for each mode is not likely to be unique. When the partial ordering is not unique for a call pattern the current implementation does not derive any more orderings. It is one of the planned extensions.

5.11 Related Research

Our research is complementary to the recent work in automating termination proofs of logic programs and is inspired by the work of Naish [64] on negation and control in Prolog. We therefore start the section with a brief discussion of termination proof procedures and their connection to our research, and then discuss research directly related to ours.

5.11.1 Termination Proofs

Frameworks for termination proof procedures (whether automated or not) for logic programs rely on the programs meeting some pre-conditions. Our research aims, in part, to meet the pre-conditions of termination proof procedures by deriving data flow and control flow simultaneously.

The earliest work in improving the termination behavior of logic programs is due to Naish [64]. Many approaches for proving termination of logic programs have been proposed by Apt, Pedreschi, Bol, and Klop [3, 4], Plumr [70], Baudinet [8], Bezem [9], Janssens and Bruynooghe [45], Vasak and Potter [86], Verschaetse and De Schreye [87], and Ullman and Van Gelder [85]. A common theme underlying most of these approaches to proving termination is to assume the well-modedness of the clauses of a program with respect to some given control flow and then constructing a well-founded ordering on the sequence of resolvents.[8] A well-founded ordering is usually shown to exist by means of a norm. Termination is proved by showing that any sequence of resolvents of an initial goal conforming to a given mode, results in a corresponding well-founded sequence of norms.

Example 5.19

Consider the familiar *concat* program.

[8] A well-founded ordering $<$ on a set T is an irreflexive and transitive ordering such that no infinite descending chains of the form $x_i > x_{i+1} > x_{i+2} > \ldots$ exist in T, where $>$ is the inverse of $<$.

```
1.  concat([], L, L).
2.  concat([X|Xs], Y, [X|Zs]) :-
        concat(Xs, Y, Zs).
```

If we use list size as the norm and *concat*$(+,-)$ as the mode for *concat*, then any goal which conforms to the above mode will generate a sequence of resolvents whose norms form a decreasing sequence of positive numbers. Since the set $\mathcal{N}$ has no infinite descending chains, the decreasing sequence of norms of the resolvents of *concat* must reach zero and terminate. □

In the presence of recursive goals, it is clear that one or more arguments of a recursive goal must be non-variable. For example, if *concat* is invoked as `concat(X,Y,Z)` with all the three arguments being unbound variables, then the recursive call `concat(X',Y',Z')` (whose arguments are also unbound variables) will not be smaller than the original call using any norm. In fact, there is an infinite sequence of resolvents of the form `concat(X,Y,Z)`, `concat(X',Y',Z')`, `concat(X'',Y'',Z'')`, $\cdots$. This is the main reason for one of our permissible mode constraints for recursive procedures, which says that at least one of the arguments of a recursive procedure must be input. (Note that the permissible mode constraint does not say that one of the arguments must be ground.) This is also the reason for comparing pairs of arguments of the head and a recursive goal of a clause of a simply recursive procedure. We already mentioned that the current definition of an input mode is solely for experimentation. A more expressive domain for types and modes such as the ones used by Ueda and Morita [82], Janssens and Bruynooghe [12], Bronsard, Lakshman, and Reddy [10], and Somogyi [74] may be used to specify that the first or the third argument must be a complete list.

It is no surprise that proofs of termination, an operational concept, use another operational concept, namely directionality of procedures. Consider the following program for reversing lists.

```
1.  reverse([], L, L).
2.  reverse([X|Xs], L1, L2) :-
        reverse(Xs, [X|L1], L2).
```

Suppose that *reverse* is called with all three arguments being complete lists, possibly containing unbound variables. The initial call to *reverse* may either succeed or fail, but there will not be an infinite derivation. This is because the first argument size decreases with each call. However, if we use list size as the norm for the arguments *and* consider the sum of the list sizes of all the arguments of `reverse` as the norm for `reverse`, then termination cannot be shown, since the sum of the list sizes of the three arguments of *reverse* does not decrease; it stays the same. When constructing a norm for a predicate, only a subset of the arguments must be considered so that the norm of these arguments (such as the list length of the first argument of *reverse*) decreases with each recursive call; these are the "input" arguments. We once again emphasize that "input" does not mean ground terms.

With a more complicated procedure such as *quicksort*, in order to drive termination proofs, one must have a well-moded clauses for a procedure and the norms must consider inter-argument relationships. The goals of a well-moded clause can be partially ordered such that when the predecessors of a goal g are solved, g's input constraints are satisfied. Thus, the concepts of well-modedness and directionality are fundamental to proving termination of logic programs.

As shown in the introductory section of this chapter, assuming a fixed control flow will render many clauses and procedures ill-moded for some calling patterns and well-moded for others. Our research attempts to fill this gap by providing a framework for deriving control flow and data flow simultaneously. Our algorithm finds permissible modes which induce well-moded clauses. The fact it does not always succeed does not detract from its usefulness. Its intended use is in a compiler which reconstructs the permissible modes omitted by the programmer. Whenever, it fails to derive a desired permissible mode the programmer may add the permissible mode declarations which are then verified by the system.

The current algorithm sacrifices completeness for tractable complexity and is expected to be useful in a system where the responsibility of permissible mode inferencing and reconstruction is shared between the system and the programmer.

Naish

Naish's research [64] on handling control in Prolog is the work most relevant to ours. He discusses problems of non-termination in Prolog and suggest using *wait* declarations which delay a goal if some of the arguments are not sufficiently instantiated. For example, a call to the *concat* procedure must be delayed in the first and the third argument are both variables. He gives an algorithm which generates *wait* declarations automatically. The essence of the algorithm is in comparing pairs of arguments of the head and a recursive goal of a clause; whenever a head argument is less general than the corresponding goal argument, a wait declaration is created. This procedure is applied to all recursive procedures. The algorithm fails when all the arguments of a head are as general as the arguments of a recursive call. The wait declarations are tested at run-time for each call to a procedure that has wait declarations. Note that wait declarations are generated for each procedure without considering the wait declarations of other procedures. This may appear as intra-procedural analysis. However, it is not so because no analysis of the body goals of a clause is done to ensure that input output requirements are met. In fact, there is no notion of input output requirements.

We use the same idea of comparing head and goal arguments, but only for simply recursive procedures, as a starting point. We define different rules for partial permissible modes of non-simply-recursive procedures, define the notion of permissible modes, and give a sound and terminating algorithm for deriving permissible modes. We use a control flow independent abstract interpretation scheme to derive the output variables of literals. The partial order (if unique) induced by a permissible mode is used as the control flow within a clause.

The wait declarations of Naish disallow unbound variables in procedure calls at certain argument positions, *i.e.*, does not require them to be fully ground terms. Our current mode definition expects the input arguments to be fully ground. However, as mentioned before, the framework does not depend on the definitions of input and output modes; the mode definitions can be refined.

Sagiv, Ullman, Van Gelder

The research of Sagiv and Ullman [71], and Ullman and Van Gelder [85] is concerned mainly with proving termination of top-down (or goal-directed) evaluation of logic programs in the context of deductive data bases. The language is usually restricted to have no function symbols. Bottom-up evaluation of function-free programs will always terminate because the Herbrand universe (the set of first order ground terms) is finite; since the domain is finite, fixed-point computations of monotonic functions will terminate. However, top-down evaluations may not terminate even if there are no function symbols.

Example 5.20

> Consider a transitive relation *path* defined on a directed graph by the following program. Assume that the edges of the graph are represented by a set of facts of the form `edge(A,B)` (not shown here), where `A` and `B` are vertices in the graph. There is a path from `X` to `Y` if there is an edge from `X` to `Y`. In the transitive case, there is a path from `X` to `Y` if there is an edge from `X` to `Z` and there is a path from `Z` to `Y`.
>
> ```
> 1.  path(X,Y) :- edge(X,Y).
> 2.  path(X,Y) :- edge(X,Z), path(Z,Y).
> ```
>
> The above program does not have function symbols. A bottom-up evaluation will terminate, whether the graph has cycles or not. A naive top-down evaluation will have infinite derivations, if there are cycles in the graph.[9] □

However, a top-down or goal-directed computation can be more efficient than a bottom-up computation since the former only computes what is necessary, unlike naive bottom-up methods which compute facts after facts whether they are needed are not.[10] Hence, researchers have concentrated on using top-down methods for query evaluation in deductive databases [83, 84]. The problem with top-down is that naive evaluation may not terminate. Termination is essential in deductive database context, especially since an ordinary user who types in a query in a high level query language is not expected to worry about termination. "Capture rules" are used to guide top-down evaluation in deductive databases. A capture rule is similar to a mode in logic programs; it says when a goal is sufficiently instantiated for evaluation. As mentioned before, the research of Ullman and Van Gelder [85] is aimed at proving termination of capture rules. Sagiv and Ullman studied the complexity of one such capture rule and show that it has better worst-case complexity than Naish's [64] method.

Our research is complementary to the above research and other research in proving termination. Once we have a permissible mode for all predicates, the termination proof procedures can be applied to show termination. The derivation of permissible modes which ensures that the data dependencies of body goals are satisfied (assuming termination) is akin to showing partial correctness of programs. Partial correctness, in program verification

[9] Experienced Prolog programmers would write the above relation with an extra parameter `Visited` which keeps track of the nodes visited and thus avoid loops.

[10] This is the same argument made about naive bottom-up computation of fixed-points and the top-down directed computation of Minimal Function Graph fixed-point semantics in Chapter 4

context, means proving statements of the following form. "If some property P holds at the start of a program and the execution reaches point A, then the property Q holds for the program state at point A." Point A may be the end of a program, in which case, the statement becomes: "If a property P holds at the start of a program, and if the program terminates, then property Q holds." If one proves termination in addition to proving the above, then one has proved total correctness.

Debray

Debray is one of earliest researchers in the field of data flow analysis of logic programs. Among his contributions are data flow analyses for deriving modes for sequential logic programs [29], and a framework for analysis of parallel logic programs [28].

The last mentioned work on analysis of parallel logic programs appears relevant to our work, since it does not assume a sequential control flow. However, it assumes that the control flow is given, either by the programmer or derived automatically by some other method. This is a departure from the usual assumption that body goals are executed left-to-right. However, the the problem of deriving control flow without assuming data flow and vice versa was not addressed.

In a technical report [27], Debray considers the problem of deriving a control flow, given "needs assertions" for procedures. "Needs assertions" are similar in spirit to our permissible modes, except that he does not define "needs assertions" and does not provide an algorithm for deriving them. Assuming that needs assertions are provided either by the programmer or by the some other method, he shows that the problem of deriving a partial ordering of body goals, in order to maximize parallelism, contains an NP-Complete problem of computing maximal independent subsets as a sub-problem. He then gives a greedy approximation algorithm for deriving a partial ordering to maximize parallelism.

We did not consider the problem of deriving a partial ordering for the body goals, given permissible modes, in detail; in many cases (even for non-deterministic programs), a permissible mode for a procedure induces a unique control flow for the clauses of a procedure. When more than one control flow is possible, Debray's algorithm for deriving a partial ordering may be used.

Ueda and Morita

Ueda and Morita [82] proposed a system of types and modes for Guarded Horn Clause programs. In their system, the types of terms are represented by rational trees, and each path in the tree is assigned a unique mode. They propose a set of mode inference rules or constraints for deriving the modes of all paths in a program. A constraint propagation system is used for deriving missing modes and for signaling mode violations, if any. Tick and others [81] provide an algorithm based on cyclic unification for deriving modes according to Ueda's mode inference rules. An FGHC program is well-moded if and only if every path in the program has a unique mode.

At a high level of abstraction, the notion of modes as constraints and mode derivation as constraint propagation point to the similarities between Ueda's and our approach to mode analysis. In fact, this is a common theme in all mode analyses. We also use a notion of well-modedness in defining permissible modes. The constraint rule that recursive calls in the body of a clause must have the same mode as the head is similar to one of Ueda's mode inference rules. However, there are some significant differences.

The first and foremost difference is that FGHC programs have an implicit notion of inputs and outputs. Head–goal matching in FGHC programs is not allowed to bind variables in the goal. If an actual parameter is less instantiated than the corresponding formal parameter of a clause, execution suspends. Thus structures and constants (non-variable terms) appearing in the head of a clause therefore signify input modality. In Horn Clause Logic (HCL) programs, there is no such restriction, *i.e.,* structures in the head of a clause do not restrict the actual parameter to be input. This makes mode inferencing harder in the case of Horn Clause Logic programs. Similar to head–goal matching, guards in a clause may be used for inferring modes for an FGHC procedure. There are no guards in HCL programs.

The second difference is that HCL procedures may have more than one permissible mode. Moded GHC programs restrict each procedure to one mode only. Another difference is that every path in an FGHC program must have a mode. For HCL programs, some arguments of a procedure may not have a mode.

Currently, our definition of modes is restrictive: "input" mode denotes a fully ground term at call time. Ueda's notion of modes is more fine-grained and similar in spirit to other proposals by Bruynooghe and Janssens [12], Bronsard, Lakshman, and Reddy [10], and Somogyi [74].

In our framework, mode conflicts or failure to derive permissible modes can be localized to clauses. Permissible mode derivation continues as long as at least one permissible mode is derived for a clause in a procedure. In Ueda's [82] and Tick's [81] approach, mode conflicts are not always localized to clauses. Attribution of errors to specific clauses is difficult because of the manner in which constraints are propagated.

Conery

Conery [20] considered the problem of ordering body literals of a clause and discusses a number of heuristics in the context of both sequential and AND-parallel executions. The literal ordering strategies considered in sequential execution are: select literals in ascending order of the number of solutions; when the number of solutions of literals are not known in advance, select in ascending order of number of unbound variables in goals. This is known as the "ground term is better" assumption. At least in a database context, more ground arguments imply that the selection condition is stronger, and hence the number of tuples which satisfy the selection criteria is expected to be equal to or smaller than the case when fewer arguments are bound to ground terms.

In the parallel execution context, Conery's literal ordering algorithm has static and dynamic components. The algorithm works with or without mode annotations, either provided by the programmer, or derived by data flow analysis. The algorithm maintains two sets of variables, those that have been assigned generators (G) and those whose producers are unknown (U), and the set of literals yet to be ordered. The static component uses mode information, if any, and initializes U and G accordingly. It also marks literals as generators or non-generators of variables, depending on whether the variables occur in an output or input argument. At run-time, mode violations are detected; variables are assigned generators based on some heuristics. For example, a *connection rule* heuristic schedules an unscheduled literal g if g has at least one variable X that is already generated and one variable Y yet to be generated such that g is not marked as a non-generator of Y. Another heuristic used is the *leftmost rule*, which designates the leftmost unscheduled literal g as the generator of all the variables in U for which g is not marked as a non-generator.

Our approach is fully static in detecting lack of permissible modes, which is similar in spirit to detecting mode violations. Like all static methods, ours will be less flexible than a dynamic method in scheduling goals, simply because there is more information at run-time. For example, the connection rule in Conery's approach requires that at least one variable in a literal g must have been generated before g can be scheduled. Clearly, the generated variable can differ from call to call. In our scheme, the requirement that at least one argument to a clause be input is similar in spirit. The argument required to be input is chosen based on both the partial permissibility conditions and the input requirements of body goals.

Others

All the other work in data flow analysis of logic programs such as those by Bruynooghe [13, 11, 12], Jones and Sondergaard [47], Winsborough [89], Marriott and Sondergaard [55], Mellish [57], assume a fixed control flow. Constraint Logic Programming languages do not have a sequential control flow; the execution of the body goals may be interleaved. However, the analysis of constraint logic programs is not related to our work mainly because of the need to safely approximate the interleaving and the implied modality of predicates. The unification of a head and a goal is not allowed to bind goal variables, *i.e.*, non-variable terms appearing in a head argument imply that the corresponding goal argument is bound to the same structure; if not, the head–goal matching either fails (if the goal argument is bound to a structure with a different functor) or suspends until the goal argument is sufficiently instantiated by another process.

5.12 Summary

Data flow analyses to date have assumed control flow and vice-versa. We studied the problem of simultaneous data flow and control flow analysis and provided a framework. The mutual dependency of data flow and control flow is handled by introducing the notion of permissible modes. Permissible modes try to capture desirable data flow. We showed that the problem of deriving all minimal permissible modes of a procedure is intractable. We then defined an approximation algorithm, which has polynomial time complexity under reasonable assumptions. The efficacy of the analysis is supported by our implementation of the permissible modes algorithm and the analysis of a number of non-trivial programs. Groundness analysis is shown to be a bottleneck in permissible mode derivation. We suggested using a condensing domain such as *Pos* for groundness analysis to avoid repeated fixed-point computations.

Well-modedness and partial ordering of literals of a clause are pre-conditions for termination proof procedures. These pre-conditions capture what is intuitively desirable data flow. We accomplish this by defining the notion of permissible modes and providing a practical procedure for deriving permissible modes without assuming control flow *a priori.*

Using the permissible modes, we derived a partial ordering of body goals such that the clause is well-moded. It is true that our procedure for deriving permissible modes is not complete. When the analyzer fails to find a desired permissible mode, the user may declare one which is then verified by the analyzer. The suggested use of our algorithm as part of a compiler is to produce the permissible modes and one or more orderings for the clauses of a procedure, from which the programmer may choose the ordering.

In an approach similar in spirit to static type inferencing and type reconstruction [23], a compiler can share the responsibility of permissible mode specification/reconstruction with the programmer. Once permissible modes are specified and reconstructed, the compiler can generate the orderings and handle most of the details of choosing among different orderings itself; when it cannot decide on an ordering based on its rules, the programmer may interact with the system and choose among the competing orderings. When combined with a condensing domain for data flow analysis, our method will be efficient enough to be used in practice. We believe that the proposed simultaneous data flow and control flow analysis will relieve the programmer of many mundane details and catch far more errors in data flow, than an analysis which assumes a fixed control flow and propagates errors in data flow.

As mentioned in the introduction, the current definitions of input and output modes limit the usefulness of the system. However, this is not a limitation of the framework. It is just the current state of the implementation. The framework can be instantiated with a more expressive mode system.

Chapter 6

Summary, Evaluation, and Future Work

In this chapter, we summarize and evaluate our contributions, discuss limitations, and present future research directions.

6.1 Summary and Evaluation

We studied two important issues in static analysis of logic programs. The first issue deals with precise derivation of data dependencies, specifically information about sharing, freeness, and groundness of variables. The second issue is the derivation of combined data flow and control flow analyses.

One of the main uses of sharing, groundness, and freeness analysis is to parallelize independent subgoals of a clause. There are other uses such as reducing the strength of unification instructions, and compile-time detection of reusable memory structures. Data flow analysis for deriving sharing information is an actively researched area. The main problems in sharing analysis pertain to intractability of precise analyses, and the imprecision of tractable analyses. Predictably, there are two broad categories of sharing analyses: one strives for tractable analysis at the expense of precision and the other aims to derive as precise an analysis as possible.

Our work takes the middle ground. We start with a sharing domain proposed by Jacobs and Langen [44]. This domain captures sharing information in a substitution precisely, but its lack of expressivity results in treating sharing as a transitive relation. This leads to two practical problems. The analysis is not very precise in deriving sharing information, and the abstract unification has an exponential component which is exercised often in practice. We define a new domain Asub of which the sharing domain of Jacobs and Langen is a component, maintain information about the linearity and freeness of terms, in addition to sharing information, and define an abstract unification algorithm that treats sharing as non-transitive whenever it is safe to do so. We analyze the time complexity of our abstract interpretation scheme and identify exponential components. Worst-case analysis does not shed light on the practicality of an algorithm when the worst-case complexity is exponential in some parameter. We introduce the notion of programs with and without "stable call substitutions" property and use this characterization to show that our scheme has polynomial time complexity for programs with stable call substitutions property. For programs without this property, a widening operation mitigates the problem of exponential complexity to a some extent.

A combination of widening, and a precise domain and operations appears to be a better alternative to choosing coarser domains and imprecise operations. The aim is to balance the concern for worst-case time complexity, and the need for improved precision in the expected case. In a practical setting, an implementation must provide both an imprecise but worst-case polynomial time analysis, and a precise and mostly practical but worst-case exponential time analysis.

A precise sharing analysis is feasible under some conditions. Our implementation of the abstract interpretation scheme and analysis of a number of programs show the feasibility of the proposed domain and operations. Another contribution is the novel characterization of programs with and without stable call substitutions property. We believe that this technique can be effectively used to shed some light on the complexity of analysis for different classes of programs. Instead of claiming that an exponential time analysis seems to be mostly practical, we can say that the analysis has polynomial time complexity for a certain class of programs and not for others. A limitation of this characterization is that while it can be used to justify why an analysis was efficient, it cannot always be used to predict whether the analysis will be efficient. This is because the characterization does not appear to be a syntactic property of programs. It is determined by the programs and the abstract domain.

The second issue studied in this dissertation is the combined derivation of data flow and control flow, without assuming one or the other. All approaches to static analysis so far assumed that either data flow information is known or control flow is given. A fixed control flow is inflexible and there is a cyclic dependency between data flow and control flow. The cyclic dependency is broken by introducing permissible modes. We show the intractability of deriving minimal permissible modes and define a polynomial time approximation algorithm for deriving minimal permissible modes. Analysis of a number of programs shows the usefulness and practicality of deriving permissible modes. For many programs, a permissible mode for a procedure induces unique control flow for all its clauses. For search-type programs, there may be more than one control flow associated with a given permissible mode. We suggest heuristics but do not explore this further in this dissertation.

Our current implementation (not the framework) has a number of limitations. Foremost among them is the definition of input and output modes. Input mode denotes a ground argument at call time and output mode denotes a ground argument when a goal succeeds. In logic programs, partially instantiated data structures are used often and we cannot reason about them using our current definition of modes. However, there have been many proposals for combined type and mode analysis (assuming control flow) for logic programs. The next line of research is to explore how to translate these ideas into our framework.

By opting for tractable permissible modes derivation algorithm, we sacrificed completeness and minimality. Thus our analyzer may reject some programs that are well-moded. It is easier to verify if a given mode is permissible than to derive one. Hence, a promising avenue is to combine permissible mode specification by the programmer with automatic derivation by the system. The programmer's intervention is needed only when the analyzer fails to derive expected permissible modes. This is analogous to combining type checking and type inferencing in the same system. Permissible mode derivation and checking can be combined effectively. The permissible modes derivation algorithm will be more useful when extended to deal with partially instantiated structures, and when combined with occasional help from the programmer.

6.2 Future Work

Defining a combined type and mode analysis is the first and foremost extension of our system. There has been active research in this area for sequential languages and concurrent logic languages. Since groundness analysis accounted for 40% to 96% of the total time spent in permissible mode derivation, defining a condensing abstract domain of modes and types is important. Recall that a condensing domain needs fixed-point computations only once per procedure to represent the abstract meaning of the procedure in closed form. This implies that the success substitution corresponding to a call substitution can be computed efficiently without fixed-point computations.

A condensing domain becomes more important in the context of modular programs and separate compilation. One cannot expect to do a global analysis (the source code may not be available, for example) every time some part of a system is changed. Thus defining condensing domains for mode and type analysis, and deriving incremental data flow and control flow analysis seem to be promising.

In a joint work [78], we developed compile-time memory reuse scheme for concurrent logic programs. A limitation of that approach is that it has to consider all possible interleavings allowed by the semantics. Deriving all permissible modes for concurrent logic procedures using input-output modality constraints imposed by the language, and using that information to focus on a few orderings that satisfy mode constraints is likely to increase opportunities for memory reuse.

A longer term goal is to study the applicability of our analysis to higher-order logic programming language such as Lambda Prolog [63] which is cleaner and more powerful than Prolog.

Bibliography

[1] Aho, A., Hopcroft, J., and Ullman, J. *The Design and Analysis of Computer Algorithms*, first ed. Addison-Wesley Publishing Company, 1974.

[2] Aho, A., Sethi, R., and Ullman, J. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading MA, 1985.

[3] Apt, K. R., Bol, R. N., and Klop, J. W. On the Safe Termination of Prolog Programs. In *International Conference on Logic Programming* (1989), pp. 353–368.

[4] Apt, K. R., and Pedreschi, D. Studies in Pure Prolog: Termination. Tech. Rep. CS-R9048, Centrum voor Wiskunde en Informatica, Amsterdam, 1990.

[5] Apt, K. R., and van Emden, M. H. Contributions to the Theory of Logic Programming. *Journal of the ACM 29*, 3 (July 1982), 841–863.

[6] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Symposium on Principles of Database Systems* (1986), pp. 1–15.

[7] Barbuti, R., Bellia, M., Levi, G., and Martelli, M. On the Integration of Logic Programming and Functional Programming. In *International Symposium on Logic Programming* (February 1984), IEEE Computer Society Press, pp. 160–168.

[8] Baudinet, M. Proving Termination Properties of Prolog Programs: A Semantic Approach. In *Annual Symposium on Logic in Computer Science* (1988), IEEE Computer Society Press, pp. 336–347.

[9] Bezem, M. Characterizing Termination of Logic Programs with Level Mappings. In *North American Conference on Logic Programming* (1989), MIT Press, pp. 69–80.

[10] Bronsard, F., Lakshman, T. K., and Reddy, U. S. A Framework of Directionality for Proving Termination of Logic Programs. In *Joint International Conference and Symposium on Logic Programming* (1992), pp. 321–335.

[11] Bruynooghe, M. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming 10*, 2 (February 1991), 91–124.

[12] Bruynooghe, M., and Janssens, G. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *International Conference and Symposium on Logic Programming* (August 1988), MIT Press, pp. 669–683.

[13] Bruynooghe, M., Janssens, G., Callebaut, A., and Demoen, B. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Symposium on Logic Programming* (1987), IEEE Computer Society Press, pp. 192–204.

[14] Chang, J. H. *High Performance Execution of Prolog Programs based on a Static Data Dependency Analysis.* PhD thesis, Dept. of Electrical Engineering and Computer Science, University of California at Berkeley, 1985.

[15] Chen, W., Kifer, M., and Warren, D. S. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In *North American Conference on Logic Programming* (1989), pp. 1090–1114.

[16] Citrin, W. V. *Parallel Unification Scheduling in Prolog.* PhD thesis, Dept. of Electrical Engineering and Computer Science, University of California at Berkeley, 1988.

[17] Codish, M., Dams, D., and Yardeni, E. Derivation and Safety of an Abstract Unification Algorithm for Groundness and Aliasing Analysis. In *International Conference on Logic Programming* (1991), MIT Press, pp. 79–96.

[18] Codish, M., Mulkers, A., Bruynooghe, M., de la Banda, M. G., and Hermenegildo, M. Improving Abstract Interpretation by Combining Domains. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (1993), ACM Press.

[19] Cohen, J. Constraint Logic Programming Languages. *Communications of the ACM 28*, 4 (1990).

[20] Conery, J. S. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Boston, MA, 1987.

[21] Cousot, P., and Cousot, R. Abstract Interpretation: A Unified Lattice-Theoretic Model for Static Analysis of Programs by Construction of Approximation of Fixpoints. In *Symposium on Principles of Programming Languages* (1977), ACM Press, pp. 238–252.

[22] Cousot, P., and Cousot, R. Systematic Design of Program Analysis Frameworks. In *Symposium on Principles of Programming Languages* (1979), ACM Press, pp. 269–282.

[23] Damas, L., and Milner, R. Principal Type-Schemes for Functional Programs. In *Symposium on Principles of Programming Languages* (1982), ACM Press, pp. 207–212.

[24] Darlington, J., Field, A. J., and Pull, H. The Unification of Functional and Logic Languages. In *Logic Programming: Relations, Functions, and Equations*, D. DeGroot and G. Lindstrom, Eds. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[25] Davis, M. D., and Weyuker, E. J. *Computability, Complexity and Languages.* Academic Press, 1983.

[26] Debray, S. On the Complexity of Dataflow Analysis of Logic Programs. In *International Colloquium on Automata, Languages and Programming* (1992), no. 623 in Lecture Notes in Computer Science, Springer-Verlag, pp. 509–520.

[27] Debray, S. K. Synthesizing Control Strategies for AND-Parallel Logic Programs. Tech. Rep. TR-87-12, University of Arizona, Tucson, Arizona, AZ-85721, 1987.

[28] Debray, S. K. Static Analysis of Parallel Logic Programs. In *International Conference and Symposium on Logic Programming* (1988), MIT Press, pp. 711–733.

[29] Debray, S. K. Static Inference of Modes and Data Dependencies. *ACM Transactions of Programming Languages and Systems 11*, 3 (July 1989), 418–450.

[30] Debray, S. K. Formal Bases for Dataflow Analysis of Logic Programs. Unpublished, 1993.

[31] Debray, S. K., and Warren, D. S. Functional Computations in Logic Programs. *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), 451–481.

[32] Dembinski, P., and Maluszynski, J. AND-Parallelism with Intelligent Backtracking for Annotated Logic Programs. In *Symposium on Logic Programming* (1985), IEEE Computer Society Press, pp. 29–38.

[33] Despain, A., and Touati, H. An Empirical Study of the Warren Abstract Machine. In *International Symposium on Logic Programming* (1987), IEEE Computer Society Press, pp. 114–124.

[34] Dietrich, S. W. Extension Tables: Memo Relations in Logic Programming. In *International Symposium on Logic Programming* (1987), IEEE Computer Society Press, pp. 264–272.

[35] Dincbas, M. Constraints, Logic Programming and Deductive Databases. In *France-Japan Artificial Intelligence and Computer Science Symposium* (1986), pp. 1–27.

[36] Dincbas, M., Simonis, H., and Van Hentenryck, P. Solving a Cutting-Stock Problem in Constraint Logic Programming. In *International Conference and Symposium on Logic Programming* (1988), MIT Press, pp. 42–58.

[37] Dincbas, M., and Van Hentenryck, P. Extended Unification Algorithms for the Integration of Functional Programming into Logic Programming. *Journal of Logic Programming 4* (September 1987), 199–227.

[38] Emden, M. H. V., and Kowalski, R. A. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM 23* (1976), 733–742.

[39] Englebert, V., Le Charlier, B., Roland, D., and Van Hentenryck, P. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and Their Experimental Evaluation. In *Programming Language Implementation and Logic Programming* (1992), vol. 631 of *Lecture Notes in Computer Science*, pp. 311–325.

[40] Gallaire, H., Minker, J., and Nicolas, J.-M. Logic and Databases: a Deductive Approach. *Computing Surveys 16*, 2 (June 1984), 153–185.

[41] Garey, M., and Johnson, D. *Computers and Intractability.* W.H. Freeman and Company, New York, 1979.

[42] Hermenegildo, M. V., and Rossi, F. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *North American Conference on Logic Programming* (1989), MIT Press, pp. 369–390.

[43] Huet, G. *Constrained Resolution : A Complete Method for Higher Order Logic.* PhD thesis, Case Western Reserve University, August 1972.

[44] Jacobs, D., and Langen, A. Accurate and efficient approximation of variable aliasing in logic programs. In *North American Conference on Logic Programming* (1989), MIT Press, pp. 154–165.

[45] Janssen, G., and Bruynooghe, M. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming 13*, 1, 2, 3 and 4 (1992), 205–258.

[46] Jones, N., and Mycroft, A. Data Flow Analysis of Applicative Programs Using Minimal Function Graphs. In *Symposium on Principles of Programming Languages* (1986), ACM Press, pp. 296–306.

[47] Jones, N., and Sondergaard, H. A Semantics Based Framework for the Abstract Interpretation of Prolog. In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds. Ellis Horwood Limited, 1987, ch. 6, pp. 123–142.

[48] Kowalski, R. A. Predicate Logic as a Programming Language. In *Proceedings IFIPS* (1974), Stockholm, pp. 569–574.

[49] Lakshman, T. K., and Reddy, U. S. Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System. In *International Symposium on Logic Programming* (1991), MIT Press, pp. 202–220.

[50] Lassez, J.-L., Maher, M., and Marriott, K. Unification Revisited. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann Publishers Inc., 1988, pp. 587–626.

[51] Le Charlier, B., and Van Hentenryck, P. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. In *International Conference on Computer Languages* (1992), IEEE Computer Society Press.

[52] Lloyd, J. W. *Foundations of Logic Programming*, second ed. Springer-Verlag, Berlin, 1987.

[53] Manna, Z. *Mathematical Theory of Computation.* Computer Science Series. McGraw Hill Book Company, 1974.

[54] Marriott, K., and Sondergaard, H. Abstract Interpretation of Logic Programs: the Denotational Approach. To appear in ACM Transactions on Programming Languages and Systems.

[55] Marriott, K., and Sondergaard, H. Bottom-up Dataflow Analysis of Normal Logic Programs. In *International Conference and Symposium on Logic Programming* (1988), MIT Press, pp. 733–748.

[56] Marriott, K., and Sondergaard, H. Notes for a Tutorial on Abstract Interpretation of Logic Programs. North American Conference on Logic Programming, 1989.

[57] Mellish, C. S. Abstract Interpretation of Prolog Programs. In *International Conference on Logic Programming* (1986), pp. 463–475.

[58] Miller, D. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation 1*, 4 (1991), 497 – 536.

[59] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, 1990.

[60] Moss, C., and Sterling, L. Prolog Thousand Database. Obtainable by anonymous ftp from src.doc.ic.ac.uk, Aug 1993.

[61] Muthukumar, K., and Hermenegildo, M. Determination of Variable Dependence Information Through Abstract Interpretation. In *North American Conference on Logic Programming* (1989), MIT Press, pp. 166–185.

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

[63] Nadathur, G., and Miller, D. Higher-Order Horn Clauses. *Journal of the ACM 37*, 4 (October 1990), 777 – 814.

[64] Naish, L. *Negation and Control in Prolog.* No. 238 in Lecture Notes in Computer Science. Springer-Verlag, 1986.

[65] Nerode, A., Marek, W., and Subrahmanian, V., Eds. *Logic Programming and Non-Monotonic Reasoning* (1991), MIT Press.

[66] Nielson, H. R., and Nielson, F. *Semantics with Applications.* John Wiley and Sons, 1992.

[67] Nilsson, U. Abstract Interpretation: A Kind of Magic. In *Programming Language Implementation and Logic Programming* (1991), vol. 528 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 299–309.

[68] Palamidessi, C. Algebraic Properties of Idempotent Substitutions. In *International Colloquium on Automata, Languages and Programming* (1990), vol. 443 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 386–399.

[69] Plotkin, G. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, CS Department, University of Aarhus, 1981.

[70] Plumer, L. *Termination Proofs for Logic Programs.* No. 446 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1990.

[71] Sagiv, Y., and Ullman, J. D. Complexity of a Top-Down Capture Rule. Tech. Rep. STAN-CS-84-1009, Stanford University, Stanford, CA 94305, 1984.

[72] Schmidt, D. *Denotational Semantics*, second ed. Wm.C.Brown Publishers, 1988.

[73] Shapiro, E. Y. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys 21*, 3 (1989), 413–510.

[74] Somogyi, Z. A System of Precise Modes for Logic Programs. In *International Conference on Logic Programming* (1987), MIT Press, pp. 769–787.

[75] Sterling, L., and Shapiro, E. Y. *The Art of Prolog.* MIT Press, Cambridge MA., 1986.

[76] Stoy, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, first ed. MIT Press, Cambridge MA., 1977.

[77] Sundararajan, R., and Conery, J. An Abstract Interpretation Scheme for Groundness, Freeness and Sharing Analysis of Logic Programs. In 12^{th} *Conference on Foundations of Software Technology and Theoretical Computer Science* (1992), no. 652 in Lecture Notes in Computer Science, Springer-Verlag, pp. 203–216.

[78] Sundararajan, R., Sastry, A. V. S., and Tick, E. Variable Threadedness Analysis for Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming* (1992), MIT Press, pp. 493–508.

[79] Tamaki, H., and Sato, T. OLD-Resolution with Tabulation. In *International Conference on Logic Programming* (1986), pp. 84–98.

[80] Tennent, D. The Denotational Semantics of Programming Languages. *Communications of the ACM 19*, 8 (1976), 437–453.

[81] Tick, E., Massey, B. C., Rakoczi, F., and Tulayathun, P. Concurrent Logic Programs *a la Mode.* In *Implementations of Logic Programming Systems*, E. Tick and G. Succi, Eds. Kluwer Academic Publishers, 1994.

[82] Ueda, K., and Morita, M. A New Implementation Technique for Flat GHC. In *International Conference on Logic Programming* (1990), MIT Press, pp. 3–17.

[83] Ullman, J. D. *Principles of Database and Knowledge-base Systems*, vol. I. Computer Science Press, 1988.

[84] Ullman, J. D. *Principles of Database and Knowledge-base Systems*, vol. II. Computer Science Press, 1988.

[85] Ullman, J. D., and Gelder, A. V. Testing Applicability of Top-Down Capture Rules. *Journal of the ACM 35*, 2 (1988), 345–373.

[86] VASAK, T., AND POTTER, J. Characterisation of Terminating Logic Programs. In *Symposium on Logic Programming* (1986), IEEE Computer Society Press, pp. 140–147.

[87] VERSCHAETSE, K., AND DE SCHREYE, D. Deriving Termination Proofs for Logic Programs, Using Abstract Procedures. In *International Conference on Logic Programming* (1991), MIT Press, pp. 301–315.

[88] WARREN, D. H. D., HERMENEGILDO, M., AND DEBRAY, S. K. On the Practicality of Global Flow Analysis of Logic Programs. In *International Conference and Symposium on Logic Programming* (1988), MIT Press, pp. 684–699.

[89] WINSBOROUGH, W. *Automatic, Transparent Parallelization of Logic Programs at Compile-time.* PhD thesis, University of Wisconsin-Madison, 1988.

[90] WINSBOROUGH, W. Multiple Specialization Using Minimal-Function Graph Semantics. *Journal of Logic Programming 13*, 1, 2, 3 and 4 (1992), 259–290.

[91] WOLFE, M., AND BANERJEE, U. Data Dependence and its Applications in Parallel Processing. *International Journal of Parallel Programming 16*, 2 (April 1987), 137 – 178.

[92] XIA, H., AND GILOI, W. K. A New Application of Abstract Interpretation in Prolog Programs. In *IFIP WG 10.0 Workshop on Concepts and Characteristics of Declarative Systems* (1988), pp. 125–134.

[93] ZOBEL, J. Derivation of Polymorphic Types for Prolog Programs. In *International Conference on Logic Programming* (1987), MIT Press, pp. 817–838.