Set Based Program Analysis

Nevin Heintze

CMU-CS-92-201

School of Computer Science
Computer Science Division
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

This research was sponsored partly by IBM through a graduate fellowship and a joint study agreement, and partly by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

Keywords: compiler optimization, program analysis, dataflow analysis, abstract interpretation, set constraints, tree automata, regular term languages, operational semantics, program approximation, types, logic programming, functional programming, imperative programming, collecting semantics, environment constraints.

Abstract

The central component of standard approaches to compile-time program analysis is an abstract domain for approximating program values. Importantly, the domain must be chosen so that an iterative fixed point computation over the domain terminates. This requirement represents a substantial restriction on the accuracy of the analysis. Furthermore, it leads to complex and often chaotic behavior.

We present an alternative approach to program analysis, called set based analysis. A key feature of set based analysis is that reasoning about a program's run-time behavior is reduced to reasoning about constraints on sets of program values. Set based analysis incorporates just a single notion of approximation: all dependencies arising from the treatment of program variables are ignored. The main advantage of set based analysis is improved accuracy, due to the absence of an abstract domain. Additionally, the use of a very simple and uniform notion of approximation leads to program analysis that is easier to understand and less sensitive to minor program modifications.

The core part of this thesis presents an algorithm for set based analysis. Importantly, the standard iterative fixed point algorithms used in the program analysis literature can not be used for set based analysis (they do not terminate). We therefore employ a fundamentally different technique, based on the use of constraints on sets of values. Using these constraints, we develop algorithms for the analysis of logic, imperative and functional languages (the underlying program values in each case are data structures). A prototype implementation is described. Although a straightforward implementation of the set constraint algorithm leads to very poor performance, very substantial improvements have been obtained using appropriate representation schemes and minimization techniques. This prototype provides strong evidence that practical analysis based on set based techniques is within reach.

An underlying philosophy of set based analysis is the separation of the definition of program approximations from algorithmic considerations. This is reflected in the use of constraints to *define* program approximation, and set constraint algorithms to *compute* it. The constraints used form a flexible and declarative intermediate language for defining and reasoning about program approximations.

Contents

List of Figures

Acknowledgements

I am greatly indebted to Joxan Jaffar, who not only introduced me to logic programming, but has been a constant source of support, beer and advice ever since. Moreover, much of this thesis represents joint work with him.

I am also indebted to Peter Lee and Frank Pfenning, my advisors at CMU. Their encouragement, guidance and feedback has been particularly important, and their efforts to teach a logic programmer something about types and functional programming have been at least partially successful.

At CMU, I found myself in the somewhat ambiguous situation of having two CMU advisors and a third "advisor" at IBM. I am very grateful to Peter, Frank, Joxan and CMU that this has been a synergistic experience.

I would like to thank Neil Jones and Dana Scott for serving on my committee and providing many useful comments and suggestions. I would also like to thank Bob Harper for standing in as a proxy for Dana during my defense and for many interesting discussions, Spiro and Dean for being great office-mates, and the administrative and technical support people at CMU for providing an excellent environment. I am very grateful to IBM for a graduate fellowship, which has provided a substantial portion of my funding.

Finally, I would like to thank Tess, whose frequent reminder of the fact that $years \neq months$ has provided an important focus to this work.

Thesis Committee

Peter Lee *(co-chair)*
Frank Pfenning *(co-chair)*
Dana Scott

Joxan Jaffar *(IBM)*
Neil Jones *(DIKU, Copenhagen)*

Chapter 1

Thesis Summary

We describe an approach to compile-time program analysis which essentially treats program variables as sets of values. Specifically, for each program variable X and program point μ, we introduce a set variable which is intended to capture the set of values for X at point μ. Then, using these set variables, *set constraints* are written to capture relationships inherent in each program statement. For example, consider an imperative program statement[1] $X := cons(Y, X)$. If μ is the program point just before this statement and ν is the program point just after this statement, then we introduce set variables $\mathcal{X}^\mu, \mathcal{X}^\nu, \mathcal{Y}^\mu, \mathcal{Y}^\nu$ to capture the values of X and Y at points μ and ν respectively, and we write the constraints

$$\begin{array}{lcl} \mathcal{X}^\nu & \supseteq & cons(\mathcal{Y}^\mu,\ \mathcal{X}^\mu) \\ \mathcal{Y}^\nu & \supseteq & \mathcal{Y}^\mu. \end{array}$$

The first constraint specifies that $\mathcal{X}^\nu$ must contain all values of the form $cons(v_1, v_2)$ such that v_1 is contained in $\mathcal{Y}^\mu$ and v_2 is contained in $\mathcal{X}^\mu$. The second specifies that $\mathcal{Y}^\nu$ must contain all values from $\mathcal{Y}^\mu$. Note that these constraints approximate the variable relationships contained in $X := cons(Y, X)$ by ignoring dependencies between the values of X and Y.

Such a construction of constraints may be used to reduce the problem of obtaining information about the run-time values of each program variable into the problem of reasoning about a collection constraints between sets

[1] *cons* denotes the list construction function.

of program values. If this reduction process is to be used as the basis of a program analysis, two issues must be addressed. First, what is the relationship between the program and the corresponding set constraints (in particular, in what sense are the constraints "correct"), and second, how can we "solve" the constraints to obtain useful information about a program's run-time behavior. In essence, these are the two main components of the thesis.

To address the first question, we show that the use of set constraints corresponds to a very simple and intuitive notion of program approximation which can be characterized as follows: the one and only approximation made is that all inter-variable dependencies are ignored. This correspondence is established for a variety of languages and operational semantics. In each case, the basic plan is the same. Starting with an operational semantics, a collecting semantics defined by specifying a notion of "program point" and then projecting the operational semantics onto these program points (this simply involves collecting the environments for each program point). This definition of collecting semantics is, by itself, of little value for defining program approximation. We therefore develop a constraint formulation of the collecting semantics: given a program, we show how *environment constraints* may be constructed such that the least model of the environment constraints corresponds to the program's collecting semantics.

The advantage of the environment constraints is that they can be reinterpreted in a number of different ways. Such an alternative interpretation is used to define *set based program approximation.* In essence, we show how the constraints may be interpreted so that inter-variable dependencies may be ignored by treating program variables as sets. Then, the set based approximation of a program is defined to be the smallest such "set" interpretation which is a model of the constraints. That is, the least (standard) model of the environment constraints gives the program's collecting semantics, and the least set model gives the set based approximation of the program. Since a set based model is a (standard) model, it follows that the set based approximation of a program is correct in the sense that it contains the program's collecting semantics.

To complete the correspondence of a program and its set constraints, we show that, when interpreted using set based models, the environment constraints for a program are equivalent to the set constraints for a program. This proves two things. First it shows that any model of the set constraints provides a "safe" approximation of a program's run-time behavior. Second it shows that the least model of the set constraints is the set based approximation of a program, and so computing the least model of the set constraints corresponds to an analysis of the program in which the only approximation made is that all inter-variable dependencies are ignored.

The central technical part of the thesis addresses the issue of "solving" set constraints. What we wish to compute is a representation of the least model of the set constraints of a program which is explicit in the sense that properties of the model are immediately evident. For example, it should be straightforward to inspect the representation to find out if the set assigned to some set variable consists only of non-empty lists. To achieve this goal, we develop an algorithm that reads in a collection of set constraints obtained from a program, and essentially produces a description of the least model in the form of a regular term grammar for each set variable appearing in the constraints. This has two important corollaries. First it proves that set based approximations are recursive (in the sense that the problem of determining elementhood in the set based approximation of a program is decidable). Second, it shows that set based program approximations are, in fact, regular languages. This has important implementation consequences.

The main contributions of the thesis can be loosely classified as: the definition of set based program approximation; an algorithm for computing set based program approximations; the development of effective implementation strategies for set based analysis, and an outline of possible extensions to set based analysis. We now expand on each of these in turn.

Definition of Set Based Approximation

One of the advantages of set based analysis is that it employs a simple and intuitive definition of program approximation. This is motivated by a desire to separate the definition of program approximations from the algorithms used to compute it, and leads to *declarative* program analysis which is easier to understand and reason about. In contrast, most approaches in the program analysis literature provide only an implicit algorithmic definition of program approximation, and this typically results in analysis that is difficult to predict.

Perhaps the main advantage of set based analysis is that the definition of approximation is very uniform. In particular, there is no approximation of the underlying domain of values. This leads to important advantages in terms of accuracy (particularly for reasoning about data-structures). Moreover, we believe that this uniformity has implications for the stability and scalability of the analysis. In contrast, abstract interpretation approaches to program analysis employ an abstract domain and this domain must be chosen so that the iterative fixed point computation terminates. This requirement represents a substantial restriction on the accuracy of the analysis, and often leads to complex and chaotic behavior.

The Set Based Analysis Algorithm

The algorithm for computing set based approximations is based on the use of set constraints. While constraints have been used before to reason about programs, our algorithm advances previous work in a number of ways. First, the constraints we use yield substantially more accurate program approximations than the constraints in previous works. In particular, a number of previous algorithms have excluded intersection. While the inclusion of intersection significantly complicates the set constraint algorithms, it also leads to much more expressive constraints. The previous works that have used intersection employ an approximate form of union and have not provided complete algorithms.

Second, we develop a general framework for using constraints to analyze programs over a variety of languages and operational semantics. This is carried out using a single constraint formalism. In contrast, previous works have been tied to a particular language and operational semantics. Moreover, we extend the application of constraints to the analysis of logic programs under top-down operational models.

Third, we use constraints to compute a specific program approximation which is defined independently of set constraints. In contrast, most previous works that involve constraints (or, equivalently, various term grammar formalisms) have employed constraints for the purpose of computing *some* program approximation. Hence, proving the correctness of such algorithms requires showing that the algorithm computes a safe approximation of program's run-time behavior. However, our algorithm is designed to compute *exactly* the set based approximation of a program, and so establishing the algorithm's correctness involves establishing equivalences, not just a containments.

Implementation of Set Based Analysis

A straightforward implementation of the set constraint algorithm leads to poor performance. However, we show that very substantial improvements can be made using appropriate representation schemes and minimization techniques. Although much work remains, we can now analyze programs of the order of a hundred lines in about 5 seconds[2]. This prototype provides strong evidence that practical analysis based on set based techniques is within reach.

Extensions to Set Based Analysis

The basic set based analysis definitions and algorithm deal with computing an approximation of the run-time values of program variables. However, the ideas of set based analysis are not restricted to this kind of analysis. We shall sketch extensions to set based analysis for modes and structure sharing, functional programs, and finally, an extension for capturing some information about inter-variable dependencies.

[2]Using SML of New Jersey on a Sun Sparc 1+.

Chapter 2

Introduction

2.1 Program Analysis

Compile-time program analysis is about analyzing a program to determine properties of its run-time behavior. Such analysis is a central component of an optimizing compiler because information about run-time behavior is prerequisite for many code optimization techniques. One important kind of analysis involves finding the possible run-time values of variables. That is, it seeks to establish *invariants* such as "when this statement is executed, the value of the variable X is always positive". Such information can be exploited during compilation to identify redundant tests or unreachable statements in a program, or to find efficient representations of data. For example, when compiling a statement that involves taking the square root of X, if it is known that X will always be positive, then at run-time there is no need to check that the square root operation is well defined. As another example, consider compiling a statement that involves extracting the first element of the list L. If it is known that L is always a non-empty list, then at run-time there is no need to check that the extraction is well defined.

Finding the possible run-time values of variables is only one kind of useful information that can be obtained from a program. Others include available subexpressions, live variables, aliasing, sharing, strictness (for functional programs) and modes (for logic programs). In this thesis we concentrate on the run-time values of variables, with particular emphasis on accurate

$X :=^{-} 5;$
Ⓐ
while $X \leq 0$ **do**
  Ⓑ
  $X := X + 1;$
  Ⓒ
Ⓓ

Program Point	Values for X
Ⓐ	$\{^{-}5\}$
Ⓑ	$\{^{-}5, \ldots, 0\}$
Ⓒ	$\{^{-}4, \ldots, 1\}$
Ⓓ	$\{1\}$

Figure 2.1: **Program 1 and Its Collecting Semantics**

treatment of data structures. This analysis is fundamental in the sense that it is applicable to almost all classes of languages, and its results can be used to directly improve other kinds of analysis.

To see how information about the run-time values of variables may be computed, consider the program in Figure 2.1. In this program, Ⓐ, Ⓑ, Ⓒ and Ⓓ are used as textual markers to indicate points in the program. Point Ⓐ indicates the point just after execution of the statement $X := {}^{-}5$, points Ⓑ and Ⓒ indicate the points immediately before and after the statement $X := X + 1$, and point Ⓓ indicates the point after execution of the entire **while-do** statement. The accompanying table in figure 2.1 shows the result of tracing the execution of the program and collecting the values of the variable X at each program point. This is often called a *collecting semantics* since it collects together information about each point in the program. Such an analysis is *global* in the sense that the program properties discovered depend on the program as a whole; information about the execution of a specific statement cannot be obtained by considering the statement in isolation.

Now, the information just computed in Figure 2.1 is exact in the sense that it corresponds exactly to what happens when the program runs. In general this is not possible because it entails solving the halting problem. Moreover, even if a particular program can be analyzed exactly, it is usually not feasible to do so. Approximation therefore plays a central role in program analysis. What is desired is an approximation that is correct in the sense that it contains all possible values encountered at run-time. In other words, if a variable X takes a value v at some program point during program execution, then we require that the set of values for X specified by the approximation at this point must contain the value v.

2.2 Program Analysis by Abstract Interpretation

Most of the approaches for computing such approximations employ *abstract interpretation* or "pseudo-evaluation", which is a technique that dates back to the early 1960's and was used in some of the first compilers [29, 52]. The basic idea is to replace the underlying collection of computation values by a collection of approximate values, and then replace computation by approximate computation over the approximate values. In the case of the previous program example, the underlying values of the computation are the integers. Consider replacing this with the approximate values *pos*, *neg* and *int*, respectively denoting the set of positive integers (including zero), the set of negative integers (also including zero) and the set of all integers. Analysis can then be performed by a "symbolic execution" of the program using these approximate values. The effect of this is to simulate the program's actual executions. Specifically, information is associated with each program point about the possible values for X that may be encountered at that point. This information is then repeatedly propagated from one program point to the next until propagation of information from each program point does not yield any new information, and a "consistent" state is reached. We illustrate this using the program from Figure 2.1. Initially the approximate value of X at each program point is the empty set of values.

1. The value $^{-}5$ is approximated by *neg*, and so the value of X at point Ⓐ is approximated by *neg*.
2. Propagate from Ⓐ to Ⓑ: since any value in *neg* is less than or equal to 0, the value of X at Ⓑ is approximated by *neg*. Note that the propagation of the information at Ⓐ to Ⓓ does not yield anything since none of the values approximated by *neg* satisfy $X \not\leq 0$.
3. Propagate from Ⓑ to Ⓒ: since adding 1 to *neg* gives *int*, the value of X at Ⓒ is approximated by *int*.
4. Propagate from Ⓒ to Ⓓ: the information at Ⓓ is updated to *pos*. Note that propagation of information from Ⓒ to Ⓑ does not yield any new information.

$L := a.b.nil;$
$X := c;$
Ⓐ
while $(L \neq nil)$ **do**
  Ⓑ
  $X := car(L);$
  $L := cdr(L);$
  Ⓒ
Ⓓ

Point	Environments
Ⓐ	$\{[L \mapsto a.b.nil, X \mapsto c]\}$
Ⓑ	$\left\{ \begin{array}{ll} [L \mapsto a.b.nil, & X \mapsto c] \\ [L \mapsto b.nil, & X \mapsto a] \end{array} \right\}$
Ⓒ	$\left\{ \begin{array}{ll} [L \mapsto b.nil, & X \mapsto a] \\ [L \mapsto nil, & X \mapsto b] \end{array} \right\}$
Ⓓ	$\{[L \mapsto nil, Y \mapsto b]\}$

Figure 2.2: **Program 2 and Its Collecting Semantics**

A key observation is that if a consistent state is reached, and this is the case after the four steps shown above, then the approximation obtained is a *correct* approximation of the program's actual executions in the sense that it contains all of the actual run-time values for X. Now there are many correct approximations of a program. For example, the approximation defined by associating all possible values with each program point is conservative. However such an extreme approximation does not say anything useful about the program. It is usually desirable to obtain the smallest (or most accurate) approximation that is correct.

The example program in Figure 2.1 has only one variable, and so to capture the executions of the program, it was sufficient to associate a set of values for this variable with each program point. In the general case, where there is more than one program variable, the only essential change is that a set of *environments* is associated with each program point instead of a set of values. To illustrate this, consider the program in figure 2.2. This main loop of this program computes the last element of the list L. The operators *car* and *cdr* are the usual LISP operators for decomposing lists ($car(L)$ gives the first element of L and $cdr(L)$ gives the "rest" of L). The symbols a, b and c denote atomic constants, and $a.b.nil$ denotes the list consisting of a followed by b. Again Ⓐ, Ⓑ, Ⓒ and Ⓓ indicate points in the program. The accompanying table in Figure 2.2 gives the environments encountered at these points during program execution.

Consider an abstract interpretation over this program. Suppose that the underlying collection of values for this program consists of constants a,

Point	Actual Environments	Abstract Interpretation
Ⓐ	$\{[L\mapsto a.b.nil, X\mapsto c]\}$	$\{[L\mapsto \mathit{list}, X\mapsto \mathit{const}]\}$
Ⓑ	$\left\{\begin{array}{ll}[L\mapsto a.b.nil, & X\mapsto c] \\ [L\mapsto b.nil, & X\mapsto a]\end{array}\right\}$	$\{[L\mapsto \mathit{non\text{-}empty}, X\mapsto \mathit{any}]\}$
Ⓒ	$\left\{\begin{array}{ll}[L\mapsto b.nil, & X\mapsto a] \\ [L\mapsto nil, & X\mapsto b]\end{array}\right\}$	$\{[L\mapsto \mathit{list}, X\mapsto \mathit{any}]\}$
Ⓓ	$\{[L\mapsto \mathit{empty}, Y\mapsto b]\}$	$\{[L\mapsto \mathit{empty}, X\mapsto \mathit{any}]\}$

Figure 2.3: **Abstract Interpretation of Program 2**

b and c, and lists constructed from these constants. One very simple collection of approximate values for analyzing this program consists of *const*, *empty*, *non-empty*, *list* and *any* respectively denoting the set of constants, the empty list, the set of non-empty lists, the set of all lists, and the set of all values. Environments can be approximated using mappings from variables into approximate values. For example, the *approximate environment* $[L\mapsto \mathit{list}, X\mapsto \mathit{const}]$ represents the set of environments in which L is bound to some list and X is bound to some constant. To analyze Program 2 using this approximation of environments, note that at point Ⓐ, the environment $[L\mapsto a.b.nil, X\mapsto c]$ is approximated by $[L\mapsto \mathit{non\text{-}empty}, X\mapsto \mathit{const}]$. Propagating this information to point Ⓑ yields $[L\mapsto \mathit{non\text{-}empty}, X\mapsto \mathit{const}]$, and this in turn gives $[L\mapsto \mathit{list}, X\mapsto \mathit{any}]$ at point Ⓒ. Further propagation steps lead to the results summarized in Figure 2.3. Note that when $[L\mapsto \mathit{non\text{-}empty}, X\mapsto \mathit{any}]$ is added to point Ⓑ, $[L\mapsto \mathit{list}, X\mapsto \mathit{const}]$ is deleted because it is subsumed by $[L\mapsto \mathit{non\text{-}empty}, X\mapsto \mathit{any}]$.

In summary, abstract interpretation involves replacing the underlying values of the computation by a collection of approximate values in such a way that program analysis can be performed by doing "approximate" computation using the approximate values. The choice of approximate values determines the character of the whole analysis. For example, the analysis just given for Program 2 was not particularly interesting because the collection of approximate values used was very simple. More accurate analysis is possible if extra approximate values are used. For example, suppose that for each natural number n, an extra value $len(n)$ is added, denoting the set of lists of length n. Analyzing Program 2 using this new collection of approx-

Point	Actual Environments	Revised Abst. Int.
Ⓐ	$\{[L\mapsto a.b.nil, X\mapsto c]\}$	$\{[L\mapsto len(2), X\mapsto const]\}$
Ⓑ	$\left\{\begin{array}{ll} [L\mapsto a.b.nil, & X\mapsto c] \\ [L\mapsto b.nil, & X\mapsto a] \end{array}\right\}$	$\left\{\begin{array}{ll} [L\mapsto len(2), & X\mapsto const] \\ [L\mapsto len(1), & X\mapsto any] \end{array}\right\}$
Ⓒ	$\left\{\begin{array}{ll} [L\mapsto b.nil, & X\mapsto a] \\ [L\mapsto nil, & X\mapsto b] \end{array}\right\}$	$\left\{\begin{array}{ll} [L\mapsto len(1), & X\mapsto any] \\ [L\mapsto, & X\mapsto any] \end{array}\right\}$
Ⓓ	$\{[L\mapsto nil, Y\mapsto b]\}$	$\{[L\mapsto, X\mapsto any]\}$

Figure 2.4: **Revised Abstract Interpretation of Program 2**

L := $b.nil$;
Ⓐ
while *true* **do**
  Ⓑ
  L := $a.L$;
Ⓒ

Point	Environments	Abst. Int.
Ⓐ	$\{[L\mapsto b.nil]\}$	$\{[L\mapsto len(1)]\}$
Ⓑ	$\left\{\begin{array}{c} [L\mapsto b.nil] \\ [L\mapsto a.b.nil] \\ [L\mapsto a.a.b.nil] \\ \vdots \end{array}\right\}$	$\left\{\begin{array}{c} [L\mapsto len(1)] \\ [L\mapsto len(2)] \\ [L\mapsto len(3)] \\ ? \end{array}\right\}$
Ⓒ	$\{\}$	$\{\}$

Figure 2.5: **Program 3 and Its Abstract Interpretation**

imate values starts by approximating the environment $[L\mapsto a.b.nil, X\mapsto c]$ at point Ⓐ by $[L\mapsto len(2), X\mapsto const]$. Repeatedly propagating this information successively adds $[L\mapsto len(2), X\mapsto const]$ to Ⓑ, $[L\mapsto len(1), X\mapsto any]$ to Ⓒ, $[L\mapsto len(1), X\mapsto any]$ to Ⓑ, $[L\mapsto empty, X\mapsto any]$ to Ⓒ, and finally $[L\mapsto empty, X\mapsto any]$ to Ⓓ. The result of this analysis is summarized in Figure 2.4. Note that for some program points there are multiple abstract environments, and this means that the analysis can capture some information about the dependencies between the values of program variables. We shall return to this issue later.

2.3 Limitations of Abstract Interpretation

Clearly an arbitrary number of approximation values can be added, and with each new approximation value, the ability of the abstract interpretation to distinguish between sets of environments is enhanced, and so the analysis improves in accuracy. However a fundamental problem arises: as more approximation values are added, the exhaustive propagation becomes more expensive and eventually does not terminate. For example, consider Program 3, which appears in Figure 2.5. Here points Ⓐ, Ⓑ and Ⓒ respectively indicate the points just after the assignment $L := b.nil$, just before execution of the $L := a.L$, and after the entire **while-do** loop. The collecting semantics of this program associates an infinite collection of environments with point Ⓑ. Now, when this program is analyzed using the collection of approximate values just described, the following propagation steps are performed. First, the environment $[L{\mapsto}b.nil]$ is approximated by $[L{\mapsto}len(1)]$ at point Ⓐ. Propagating this information to point Ⓑ yields $[L{\mapsto}len(1)]$. Subsequent propagation steps lead to the addition of $[L{\mapsto}len(2)]$, $[L{\mapsto}len(3)]$, $[L{\mapsto}len(4)]$, ... to point Ⓑ. This propagation process does not terminate.

Although Program 3 is somewhat artificial, since it is easy to see that the loop in this program does not terminate, the situation illustrated by this program frequently arises. Typically the conditions that appear in a program are too complex to analyze exactly, and often the approximations used are tantamount to replacing the condition with *true*. More abstractly, program analysis must effectively deal with non-termination and infinite collections of values for two main reasons. First, we cannot in general statically determine whether a program will terminate. Even if a program is guaranteed to terminate, it is rarely feasible to analyze the program exactly, and the process of approximating the behavior of a program typically introduces non-terminating computation. Second, program analysis is usually carried out in some initial context or environment that involves descriptions of infinite sets of values. For example, we may wish to analyze the while loop of Program 2 in the context where L is an arbitrary list consisting of the constants a and b and X is an arbitrary value.

Since the treatment of infinite sets of values is a necessary part of program analysis, a fundamental questions arises: how can the exhaustive propagation process be made to terminate? Clearly if there are only a finite number of approximate values, then the propagation process always terminates. This is because the propagation process is monotonic – each step serves to increase the information at each point – and so there can only be a finite number of increments of the information at each point. However less restrictive approaches are possible. To describe these, we first formulate the exhaustive propagation process as an iterative least fixed point computation.

In essence, the process of propagating information from one program point to another can be characterized as a function. Specifically, let $\mathcal{F}$ denote the function which, when given an association of information with program points, computes the result of updating this association by performing a "single-step" propagation of its information. The exhaustive propagation of information from one point to the other then essentially corresponds to starting with the association $\perp$ that associates the empty set with each program point, and then exhaustively applying the function $\mathcal{F}$ until no new information is obtained. That is, the exhaustive propagation process corresponds to iteratively computing the least fixed point of $\mathcal{F}$ by constructing the sequence

$$\perp, \mathcal{F}(\perp), \mathcal{F}(\mathcal{F}(\perp)), \ \mathcal{F}(\mathcal{F}(\mathcal{F}(\perp))), \ldots$$

Note that this correspondence is not exact since the repeated application of $\mathcal{F}$ corresponds to a specific order of propagation of information from one point to another. However the iterative fixed point computation provides an important characterization of the termination properties of the exhaustive propagation process in the following sense: exhaustive propagation only terminates when the iterative fixed point computation terminates.

Now, the function $\mathcal{F}$ maps from and into associations of information with program points. Let $\mathcal{D}$ denote the set of such associations. These associations can be ordered according to the amount of information they contain. The association $\perp$, which associates the empty set with each program point, is the smallest association. More generally, if A_1 and A_2 are associations then we write $A_1 \subseteq A_2$ when, for each program point, the information of A_2 is at least that of A_1. According to this ordering, $\mathcal{F}$ is an increasing function in the sense that $A \subseteq \mathcal{F}(A)$. This means that the iterative fixed point computation is increasing in the sense that $\perp \subseteq \mathcal{F}(\perp) \subseteq \mathcal{F}(\mathcal{F}(\perp)) \subseteq \mathcal{F}(\mathcal{F}(\mathcal{F}(\perp))) \subseteq \cdots$. Clearly the iterative fixed point is guaranteed to terminate if $\mathcal{D}$ is finite. Termination is also guaranteed if $\mathcal{D}$ does not contain any sequences of elements of the form $A_1 \subseteq A_2 \subseteq A_3 \subseteq \cdots$ such that each A_i is distinct association. That is, termination is guaranteed if $\mathcal{D}$ does not have

$L := b.nil;$
Ⓐ
while *true* **do**
  Ⓑ
  $L := a.L;$
Ⓒ

Point	Environments	Revised Abst. Int.
Ⓐ	$\{[L\mapsto b.nil]\}$	$\{[L\mapsto b.nil]\}$
Ⓑ	$\left\{\begin{array}{c}[L\mapsto b.nil]\\ [L\mapsto a.b.nil]\\ [L\mapsto a.a.b.nil]\\ \vdots\end{array}\right\}$	$\{[L\mapsto len{+}(1)]\}$
Ⓒ	$\{\}$	$\{\}$

Figure 2.6: **Revised Abstract Interpretation of Program 3**

any infinite ascending chains.

As an example where $\mathcal{D}$ is infinite but has no infinite ascending chains, consider the abstract interpretation of Program 3 using approximate values of the form $len{+}(n)$ denoting lists of length n *or more*. Again, environments are approximated by using mappings from variables into approximate values. The set $\mathcal{D}$ of associations is infinite, but does not contain any infinite ascending chains, and so iterative fixed point computation over $\mathcal{D}$ always terminates. As an example, the analysis of Program 3 terminates, and is presented in Figure 2.6. Note that, for termination reasons, it is important in this example to maintain the information at each point in a non-redundant form. Consider, for example, the situation when the approximate environment at Ⓑ is $[L\mapsto len{+}(1)]$ and a propagation step computes the "new" approximate environment $[L\mapsto len{+}(2)]$ for Ⓑ. Since this new information is already subsumed by $[L\mapsto len{+}(1)]$, the information at Ⓑ need not be changed.

In short, only certain collections of approximate values can be used in an abstract interpretation. Although the set of associations $\mathcal{D}$ need not be finite, it must be essentially finite in the sense that only a finite number of elements of $\mathcal{D}$ are visited in any fixed point computation. This is a very significant restriction. It is worth noting that there are methods for obtaining terminating algorithms even when the collection of approximate values does not satisfy such a finiteness criteria. This is achieved by computing something other than the least fixed point of the propagation function $\mathcal{F}$. One such technique is widening [13]. To illustrate widening, consider analyzing Program 3 using the approximate values of the form $len(n)$ and $len{+}(n)$, where the former denotes lists of length n, and the latter denotes lists of length at least n. Now, the analysis of Program 3 using these approximate values is identical to the analysis outlined in the table of Figure 2.5, and does not terminate. The reason is that the propagation process successively adds the sequence of approximate environments $[L \mapsto len(1)]$, $[L \mapsto len(2)]$ $[L \mapsto len(3)]$, ... to point Ⓑ. In essence, the effect of widening is to avoid such infinite sequences by guessing their limit point.

Specifically, let I denote the information associated with a program point, and suppose that the propagation process determines that this information should be updated with the new information I'. Normally, the information for the program point is updated to $I \cup I'$ (for the above analysis of Program 3, this is obtained by taking the union of the approximate environments in I and I' and then removing any redundant approximate environments). However in widening, the program point is updated with $I \nabla I'$, where ∇ is a function that approximates the combination of I and I'. For example, an appropriate ∇ for analysis of Program 3 would inspect I and I' and check to see if there are approximate environments of the form $[L \mapsto len(i)] \in I$ and $[L \mapsto len(j)] \in I'$ such that $i < j$. If so, then the environment $[L \mapsto len(j)]$ in I' would be replaced by $[L \mapsto len{+}(i)]$ before joining I' with I. This means that the analysis of Program 3 proceeds as follows:

- Initially the environment $[L \mapsto b.nil]$ is approximated by $[L \mapsto len(1)]$ at point Ⓐ.
- Propagate from Ⓐ to Ⓑ: update Ⓑ to $\{[L \mapsto len(1)]\}$.
- Propagate from Ⓑ to Ⓑ: this yields the new approximate environment $\{[L \mapsto len(2)]\}$ at Ⓑ, and so widening is used to obtain $\{[L \mapsto len(1)]\} \nabla \{[L \mapsto len(2)]\} = \{[L \mapsto len{+}(1)]\}$ for Ⓑ.

Intuitively, the intent of ∇ is to determine if I and I' form the start of a possible infinitely increasing sequence of approximate environment sets, and if so, to "round-up" to some appropriate set of environments. Of course, this "rounding-up" could overshoot the limit of the sequence, and cause the computation to return something other than the least fixed point of $\mathcal{F}$. To see this, consider the Program 4 in Figure 2.7. This program is essentially the same as Program 3 except that it terminates after two iterations of the **while-do** loop. Now, using the same approximate values as those just

$L := b.nil;$
Ⓐ
while $len(L) \neq 3$ **do**
  Ⓑ
  $L := a.L;$
Ⓒ

Point	Environments
Ⓐ	$\{[L \mapsto b.nil]\}$
Ⓑ	$\left\{ \begin{array}{c} [L \mapsto b.nil] \\ [L \mapsto a.b.nil] \end{array} \right\}$
Ⓒ	$\{[L \mapsto a.b.nil]\}$

Figure 2.7: **Program 4 and Its Collecting Semantics**

Point	Environments	Widening	No Widening
Ⓐ	$\{[L \mapsto b.nil]\}$	$\{[L \mapsto len(1)]\}$	$\{[L \mapsto len(1)]\}$
Ⓑ	$\left\{ \begin{array}{c} [L \mapsto b.nil] \\ [L \mapsto a.b.nil] \end{array} \right\}$	$\{[L \mapsto len{+}(1)]\}$	$\left\{ \begin{array}{c} [L \mapsto len(1)] \\ [L \mapsto len(2)] \end{array} \right\}$
Ⓒ	$\{[L \mapsto a.b.nil]\}$	$\{[L \mapsto len(3)]\}$	$\{[L \mapsto len(3)]\}$

Figure 2.8: **Abstract Interpretation of Program 4**

used to analyze Program 3, and using the same widening operator ∇, the analysis of Program 4 proceeds identically to that of Program 3, and the result of the analysis appears in the second column of the table in Figure 2.8. The analysis without widening terminates in this case, and its result is given in the third column of the table. For this example the widening operator over-approximates when the approximate environment $[L \mapsto len(2)]$ is added to point Ⓑ, and hence the analysis does not yield the least fixed point of $\mathcal{F}$, but rather some arbitrary approximation of it. A complementary technique of narrowing has been developed that partly compensates for the over-approximation inherent in widening [13], however this is not sufficient to regain the least fixed point.

In summary, abstract interpretation requires that the collection of approximate values be essentially finite in character to ensure termination of the iterative fixed point computation. The techniques of widening and narrowing can be used to address this restriction, but at the cost of introducing extra approximation – that is, widening introduces another level of approximation over and above that introduced by the use of approximate values. Due to the *ad hoc* nature of widening and narrowing, it is difficult to give a formal characterization of the restrictions on the collection of approximate values necessary for termination. However the following general observation can be made: abstract interpretation (with or without widening and narrowing) explicitly constructs a succession of elements from $\mathcal{D}$ in order to find the least fixed point of $\mathcal{F}$ (or some approximation thereof), and this succession of elements must be finite. This implies that only a finite part of the function space of $\mathcal{F}$ can be investigated during this fixed point computation.

This observation has two important implications. First, the finitary nature of abstract interpretation implies that there is a fundamental limitation on the accuracy of this approach to program analysis. There are decidable kinds of analysis that *cannot* be computed using abstract interpretation (even with widening and narrowing). The set based analysis considered in this thesis is one example.

Second, the finitary nature of abstract interpretation means that there are typically very subtle interactions between the collection of approximate values used and the operations of the language being analyzed. This frequently leads to chaotic and unintuitive behavior. In particular, it is often very difficult for a programmer to determine what an abstract interpretation based analysis will yield.

For example, consider Program 5 in Figure 2.9, which flattens out and reverses the input list L so that, on termination, FL is $4.3.2.1.nil$. Suppose that the basic values of this program are integers, characters and lists. Consider an abstract interpretation of this program in which, corresponding to the basic values, there are approximate values *int*, *char* and *atomic* respectively denoting the sets of integers, characters and non-list values. Also add a family of approximate values $list(\alpha)$ where α ranges over approximate values. For example, $list(int)$ and $list(list(int))$ are both approximate values with the obvious meanings. Now, we might expect that the analysis of Program 5 using this collection of approximate values might lead to the approximation of FL at point Ⓒ by $list(int)$. However this is not the case because at point Ⓑ the program variable L is in general bound to a list whose elements are integers *and* integer lists. Since there is no approximate value corresponding to such lists, L must be approximated by *list* at this point, and it follows that the best that can be obtained for FL at point Ⓒ is $list(atomic)$. In essence, the problem relates to the intermediate values computed by a program. Even though this collection of approximate values is sufficiently expressive to represent the results of a computation, it

$L := (1.2.nil).(3.4.nil);$
$FL := nil;$
Ⓐ
while$(L \neq nil)$ **do**
  Ⓑ
  if $car(L) = nil$ **then**
    $L := cdr(L);$
  else if $islist(car(L))$ **then**
    $L := car(car(L)).cdr(car(L)).cdr(L);$
  else
    $FL := car(L).FL;$
    $L := cdr(L);$
Ⓒ

Figure 2.9: **Program 5**

is not sufficiently expressive to represent intermediate parts of the computation. Such a lack of uniformity is unavoidable in abstract interpretation approaches. Program 5 is a very simple program and it is quite easy to identify why the expected result was not obtained. However, in large programs this is not usually possible. These deficiencies of abstract interpretation – lack of uniformity, predictability and stability – are particularly relevant for scaling up abstract interpretation based approaches to large systems.

To address these deficiencies, this thesis seeks an approach to program analysis that is:

- **declarative**: we desire a simple definition of approximation that has an intuitive relationship to program meaning and is independent of algorithmic considerations;
- **accurate**: the approximation must be meaningful for program analysis; and
- **decidable**: there must be algorithms to compute the approximation.

2.4 Approximation of Values and Variables

At the heart of program analysis is the notion of program approximation. The example abstract interpretations given in the previous two sections have, for simplicity of presentation, focussed on approximation of the underlying values of computation. For example, in the analysis of Program 1, the integers were approximated using *pos*, *neg* and *int*. In the analysis of Program 2, atomic constants and lists were approximated using *const*, *empty*, *non-empty*, *list* and *any*. These approximate values were then used to approximate environments by simply considering mappings from variables into approximate values. What was not considered in these previous examples was the possibility for introducing approximation in the treatment of environments. For example, given a collection of approximate values such as *pos*, *neg* and *int*, one can either approximate environments by using collections of mappings from variables into approximate values, or one can approximate environments using a single mapping from variables into approximate values. The essential difference between these approaches is that the former captures some dependencies between possible variable values, whereas the latter ignores all dependencies between variable values. Consider approximating the environments $\{[X \mapsto {}^{-}1, Y \mapsto 1], [X \mapsto 3, Y \mapsto {}^{-}3]\}$. Using the former approach, this is approximated by $\{[X \mapsto neg, Y \mapsto pos], [X \mapsto pos, Y \mapsto neg]\}$. Using the latter approach, it is approximated by $\{[X \mapsto int, Y \mapsto int]\}$. It is also possible to extend the ability to represent variable interdependencies further and, for example, approximate these environments by the formula $X = {}^{-}Y$.

In other words, approximation can be introduced in two ways: in the treatment of the underlying values, and in the treatment of variables. Approximation may be introduced in the treatment of values through the use of approximate (or *abstract*) values that are essentially finite descriptions of sets of program values. Intuitively, approximation in this case appears when set of values are "rounding up" to the nearest approximate value during the abstract interpretation process. In contrast, approximation may be introduced in the treatment of variables by forgetting some of the dependencies that arise in the treatment of variables. To illustrate the kinds of dependencies that may arise, consider again the question of representing $\{[X \mapsto {}^{-}1, Y \mapsto 1], [X \mapsto 3, Y \mapsto {}^{-}3]\}$. As noted previously, we could choose to represent these environments using $\{[X \mapsto neg, Y \mapsto pos], [X \mapsto pos, Y \mapsto neg]\}$ (which captures some dependencies between X and Y) or $\{[X \mapsto int, Y \mapsto int]\}$

(which ignores all dependencies between X and Y). As another example of dependencies introduced through the treatment of variables, consider the program statement $Y := pair(X, X)$. When such a statement is analyzed, there is a choice of how the two occurrences of X are to be treated. For example, if the approximate environment before execution of this statement maps X into pos, then we could represent the result for Y by $pair(pos, pos)$ indicating the Y is a pair whose first and second components are positive numbers, or by "$pair(v, v) \wedge v \in pos$" indicating a dependency between the arguments of *pair*. We shall collectively refer to all dependencies that may be introduced by variables as *inter-variable dependencies*. We remark that the notion of inter-variable dependency was first considered by Jones and Muchnick [33]], who used the term independent attribute analysis.

Note that the distinction between approximations of values and approximations of inter-variable dependencies is sometimes obscured by interactions between inter-variable dependencies and values. For example, the environments $\{(X \mapsto {}^-1, Y \mapsto 1], [X \mapsto 3, Y \mapsto {}^-3]\}$ could be approximated by the formula $(X =^- Y) \wedge ((X = Y - 2) \vee (X = Y + 6))$, and even though there may be some underlying approximation of values, the expressive power of the inter-variable dependencies mechanism allows the actual values of X and Y to be completely recovered, and so, in effect, there is no approximation of the values.

Most program analysis algorithms incorporate approximation of the underlying values as well as approximation of inter-variable dependencies. (For efficiency reasons, analyzers often completely omit reasoning about inter-variable dependencies [33].) A fundamental question is: what are the minimal notions of approximation required for the decidability of the resulting analysis? Some approximation of inter-variable dependencies is necessary because exactness on such dependencies essentially gives exact program analysis. What has not been addressed is whether or not program analysis is decidable when ignoring inter-variable dependencies is the *only* approximation used. Furthermore, can this be used as the basis of a practical program analysis system? We will show that ignoring inter-variable dependencies is sufficient for decidability and can be used for practical program analysis.

2.5 Set Based Analysis

The set based approach to program analysis has its origins in the use of constraints to perform type analysis of programs [32, 48, 63]. In essence, set based analysis involves first writing set constraints (a calculus for expressing relationships between sets of program values) to describe the run-time behavior of a program, and then solving these constraints to find their most accurate solution. The fundamental difference between set based analysis and abstract interpretation approaches is that set based analysis does not use an iterative fixed point computation over an (essentially finite) collection of approximate values. In particular, there are no depth bounds or other *a priori* restrictions on the sets of values that can be manipulated during the analysis.

If one takes a very broad view of abstract interpretation as a framework for defining program approximations (as opposed to the more algorithmic iterative fixed point view), then set based analysis can be formulated as an abstract interpretation. However, the corresponding iterative fixed point computations do not terminate, and so iterative fixed point computation cannot be used in set based analysis. One of the main issues addressed by this thesis is the development of algorithms to show that set based analysis is decidable.

Set Constraints

Consider analyzing a program in such a way that inter-variable dependencies are ignored. That is, we wish to avoid using reasoning such as "variable X takes value a iff variable Y takes value b, and X takes value c iff Y takes value d". Instead, we wish to reason about the program by considering the *sets* of values that each program variable can assume, and ignoring dependencies between the elements of these sets. In essence, program variables are treated as sets, and this is the motivation for describing such analysis as *set based analysis*. Specifically, for each program variable X and program point μ, we introduce a set variable $\mathcal{X}^\mu$ to denote the set of values of the program variable X at the point μ. Then, by inspecting the program, we construct constraints between these set variables to capture the relationships between program variables that are contained in P. We shall now illustrate how these constraints may be constructed. Note that the constraints demonstrated in

1. $L := cons(a, cons(b, nil));$

2. $X := c;$

Ⓐ

3. **while** $(L \neq nil)$ **do**

Ⓑ

4. $\quad X := car(L);$

Ⓒ

5. $\quad L := cdr(L);$

Ⓓ

Ⓔ

Figure 2.10: **Program 2 (Revisited)**

this section are only suggestive of how this may be done, and the actual set constraints used in set based analysis are somewhat more complicated and more accurate.

Consider again the imperative program in Figure 2.10; for clarity, the following discussion uses *cons* for lists rather than the infix "." notation. Introduce set variables to denote the values of each variable at each program point. For example, let $\mathcal{L}^{Ⓑ}$ and $\mathcal{X}^{Ⓑ}$ respectively be the set variables corresponding to L and X at program point Ⓑ (that is, just before execution of statement 4), and let $\mathcal{L}^{Ⓒ}$ and $\mathcal{X}^{Ⓒ}$ respectively be the set variables corresponding to L and X at point Ⓒ. Now corresponding to statement 4, $X := car(L)$, consider the following constraints:

$$\begin{array}{lcl} \mathcal{L}^{Ⓒ} & \supseteq & \mathcal{L}^{Ⓑ} \\ \mathcal{X}^{Ⓒ} & \supseteq & car(\mathcal{L}^{Ⓑ}). \end{array}$$

The first constraint specifies that the values for L after execution of statement 4 must include all those before the execution of statement 4. The second constraint specifies that the values for X after execution of statement 4 must include the *car* of values of L before statement 4. The symbol *car* in this last constraint denotes the set-wise version of the program symbol *car*. Specifically, where S is a set of values, $car(S)$ denotes $\{v_1 : cons(v_1, v_2) \in S\}$. Note that these constraints could have been expressed using set equality rather containment. However, the use of containment yields constraints that express a minimal notion of consistency between the sets at each program point. It also simplifies the treatment of converging paths of control flow. We shall return to this issue later, but first we give the rest of the

$$
\begin{array}{lllllll}
\mathcal{L}^{Ⓐ} & \supseteq & cons(a, cons(b, nil)) & \quad & \mathcal{X}^{Ⓐ} & \supseteq & c \\
\mathcal{L}^{Ⓑ} & \supseteq & \mathcal{L}^{Ⓐ} \cap \overline{nil} & & \mathcal{X}^{Ⓑ} & \supseteq & \mathcal{X}^{Ⓐ} \\
\mathcal{L}^{Ⓑ} & \supseteq & \mathcal{L}^{Ⓓ} \cap \overline{nil} & & \mathcal{X}^{Ⓑ} & \supseteq & \mathcal{X}^{Ⓓ} \\
\mathcal{L}^{Ⓒ} & \supseteq & \mathcal{L}^{Ⓑ} & & \mathcal{X}^{Ⓒ} & \supseteq & car(\mathcal{L}^{Ⓑ}) \\
\mathcal{L}^{Ⓓ} & \supseteq & cdr(\mathcal{L}^{Ⓒ}) & & \mathcal{X}^{Ⓓ} & \supseteq & \mathcal{X}^{Ⓒ} \\
\mathcal{L}^{Ⓔ} & \supseteq & \mathcal{L}^{Ⓐ} \cap nil & & \mathcal{X}^{Ⓔ} & \supseteq & \mathcal{X}^{Ⓐ} \\
\mathcal{L}^{Ⓔ} & \supseteq & \mathcal{L}^{Ⓓ} \cap nil & & \mathcal{X}^{Ⓔ} & \supseteq & \mathcal{X}^{Ⓓ}
\end{array}
$$

Figure 2.11: **Set Constraints for Program 2**

constraints for the program.

Consider statement 5. Introduce set variables $\mathcal{L}^{Ⓓ}$ and $\mathcal{X}^{Ⓓ}$ to describe the values of L and X at point Ⓓ, and construct the following constraints:

$$
\begin{array}{lll}
\mathcal{L}^{Ⓓ} & \supseteq & cdr(\mathcal{L}^{Ⓒ}) \\
\mathcal{X}^{Ⓓ} & \supseteq & \mathcal{X}^{Ⓒ}
\end{array}
$$

The first constraint specifies that the values for L after statement 5 must include the *car* of all values for L before statement 4, and the second specifies that the values for X after statement 5 must include all values for X before the statement.

Adding constraints for the remaining statements leads to the following collection of constraints in Figure 2.11, where the symbol *nil* in the set constraints denotes the singleton set of values $\{nil\}$, $\overline{nil}$ denotes the set of all values different from *nil*, and $\cap$ has its usual set theoretic meaning. Note the treatment of the **while-do** statement. For example, the constraint $\mathcal{L}^{Ⓑ} \supseteq \mathcal{L}^{Ⓐ} \cap \overline{nil}$ corresponds to the possibility of flow of control from point Ⓐ to point Ⓑ, and states that the values for L at Ⓑ must contain the L values at Ⓐ that are different from *nil*. Similarly the constraint $\mathcal{L}^{Ⓑ} \supseteq \mathcal{L}^{Ⓓ} \cap \overline{nil}$ corresponds to the possibility of flow of control from point Ⓓ to point Ⓑ.

These constraints represent an approximation of the relationships implicit in the imperative program. They are conservative in the sense any assignment of sets to the set variables that satisfies each constraint is guaranteed to contain all of the possible values encountered at run-time. It is therefore natural to consider the least such assignment of sets because this yields the most accurate information. For the constraints in figure 2.11, the least assignment of sets that satisfies the constraints is

$$\begin{array}{lclclcl}
\mathcal{L}^{Ⓐ} & \mapsto & \{cons(a, cons(b, nil))\} & & \mathcal{X}^{Ⓐ} & \mapsto & \{c\} \\
\mathcal{L}^{Ⓑ} & \mapsto & \{cons(a, cons(b, nil)), cons(b, nil)\} & & \mathcal{X}^{Ⓑ} & \mapsto & \{a, b, c\} \\
\mathcal{L}^{Ⓒ} & \mapsto & \{cons(a, cons(b, nil)), cons(b, nil)\} & & \mathcal{X}^{Ⓒ} & \mapsto & \{a, b\} \\
\mathcal{L}^{Ⓓ} & \mapsto & \{cons(b, nil), nil\} & & \mathcal{X}^{Ⓓ} & \mapsto & \{a, b\} \\
\mathcal{L}^{Ⓔ} & \mapsto & \{nil\} & & \mathcal{X}^{Ⓔ} & \mapsto & \{a, b, c\}
\end{array}$$

This clearly represents an approximation of the values that are encountered at run-time. For example, at point Ⓔ, the only value encountered at run-time for X is b, but this is approximated here by $\{a, b, c\}$. The reason for this approximation is the dependencies between L and X have been ignored, so that the relationship "L takes value *nil* iff X takes value b" is ignored. In general, we shall refer to an assignment of sets to set variables that satisfies a collection of constraints as a *model* of the constraints. The above assignment is the *least* model of the constraints in Figure 2.11.

As noted earlier, the constraints we have used for modeling programs could have been expressed using set equality rather than containment. For example the equality

$$\mathcal{L}^{Ⓑ} \;=\; \left(\mathcal{L}^{Ⓐ} \cap \overline{nil}\right) \;\cup\; \left(\mathcal{L}^{Ⓓ} \cap \overline{nil}\right)$$

could be used in the place of the two constraints $\mathcal{L}^{Ⓑ} \supseteq \mathcal{L}^{Ⓐ} \cap \overline{nil}$ and $\mathcal{L}^{Ⓑ} \supseteq \mathcal{L}^{Ⓓ} \cap \overline{nil}$ in Figure 2.11. However, the use of containment has a number of advantages. First, it yields constraints that have a more intuitive reading – they correspond to minimal consistency relationships between the sets at each program point. The use of equality is in some sense an over-specification of the relationships inherent in the program. (One minor advantage of equality is that it reduces the number of models of the constraints, but note that the constraints still would not define a unique model.) Second, the use of containment simplifies the construction of the constraints because it allows constraints to be constructed on a statement by statement basis, rather than having to predetermine the possible paths of control into a program point. Third, the correctness of the set constraints (and the justification that they embody a natural and intuitive notion of approximation) employs *environment constraints*, which are like set constraints except that they describe relationships between sets of *environments* at each

$\leftarrow p(f(X,Y))$.	$\leftarrow$ Ⓐ, $p(f(X,Y))$, Ⓑ.
$p(f(X,Y)) \leftarrow q(X,Y)$.	$p(f(X,Y)) \leftarrow$ Ⓒ, $q(X,Y)$, Ⓓ.
$q(a,b)$.	$q(a,b) \leftarrow$ Ⓔ.
$q(c,d)$.	$q(c,d) \leftarrow$ Ⓕ.

Figure 2.12: **Program 6 (With and Without Program Points)**

program point instead of between sets of variable values. The use of containment rather than equality in these environments constraints is crucial for defining notions of program approximation. Since the set constraints are derived from these environment constraints, the use of containment in set constraints is a matter of consistency and convenience. Fourth, the use of containment simplifies the presentation of the algorithms for solving set constraints because it allows the form of the constraints to be significantly simplified.

Note that for convenience of presentation, certain aspects of the behavior of the program have been omitted from the constraints in figure 2.11. The main omission is the treatment of program errors. For example, following a statement such as $X := car(L)$, the values for L must be of the form $cons(\cdots)$, because otherwise an error must have occurred during the execution of $car(L)$. The more accurate set constraints described in the body of this thesis shall take into account such reasoning. Implicit in this reasoning is an assumption that after a computation encounters an error condition, we can ignore the remainder of its execution. Specifically, it is assumed that either (i) when such an error occurs the program aborts, and so control never reaches the point following the statement $X := car(L)$ with the non-*cons* L value, or (ii) if an error occurs, then we are not interested in the subsequent execution of the program, and so it is permissible for the analysis to be "unsafe" with respect to executions because any compiler optimizations made on the basis of such information can only alter the behavior of a program *after* an error has occurred.

We now show how constraints may be used to analyze logic programs. Consider the logic program in Figure 2.12. Figure 2.12 also contains a version of this program annotated with program points Ⓐ, Ⓑ, Ⓒ, Ⓓ, Ⓔ and Ⓕ. The points Ⓑ, Ⓓ, Ⓔ and Ⓕ represent the points at the end of the execution of the goal or rule in which they appear. The points Ⓐ and Ⓒ denote the points just before execution of $p(f(X,Y))$ and $q(X,Y)$ respectively. Suppose that we wish to analyze this program to determine the set of ground instances of the bindings for each program variable during a PROLOG style top-down left-to-right execution of the program. As before, the first step in the construction of the constraints is the introduction of set variables $\mathcal{X}^{Ⓐ}, \mathcal{Y}^{Ⓐ}, \mathcal{X}^{Ⓑ}, \mathcal{Y}^{Ⓑ}, \ldots$ to collect the values for each program variable at each program point. Four new set variables $Call_p$, $Call_q$, Ret_p and Ret_q are also introduced. To explain the purpose of these variables, recall that top-down left-to-right execution of a program involves repeatedly applying the following step: inspect the left-most atom of the goal and choose a (suitably renamed) program rule such that the left-most goal atom and the head of the rule unify and let their most general unifier be θ, replace the left-most goal atom with the body of the rule, and apply θ to the resulting goal. In essence, the left-most goal atom acts like a procedure "call". Such a call is completed (or "solved") when all of the subgoals introduced by the call have themselves been completed, and this is analogous to a procedure return. Now, the variables $Call_p$ and $Call_q$ respectively correspond to the ground instances of the calls made to the predicates p and q during program execution, and Ret_p and Ret_q respectively correspond to the ground instances of the returns involving p and q during program execution.

To illustrate how constraints are constructed, consider the second rule of the logic program. In essence, this rule says that one way to solve a call of the form $p(f(X, Y))$ is by calling $q(X, Y)$. Hence, the values for variable X at point Ⓒ (just before the calling of $q(X, Y)$) are those values of X such that $p(f(X, \ldots))$ is an element of $Call_p$, and this can be written as the constraint

$$\mathcal{X}^{Ⓒ} \supseteq \Big\{X : \exists Y \left(p(f(X,Y)) \in Call_p\right)\Big\}.$$

At point Ⓓ, the value of X must be such that $p(f(X, \cdots))$ is an element of $Call_p$ and $q(X, Y)$ is an element of Ret_q, and this can be written as

$$\mathcal{X}^{Ⓓ} \supseteq \{X : \exists Y \left(p(f(X,Y)) \in Call_p \quad \wedge \quad q(X,Y) \in Ret_q\right)\}.$$

This rule also contributes to the sets $Call_q$ and Ret_p. Specifically, the body of the rule initiates a call to q, and the rule as a whole may be used to solve a call to p. This leads to

$$\begin{array}{rcl} Call_q & \supseteq & q(X^{Ⓒ},\ Y^{Ⓒ}) \\ Ret_p & \supseteq & p(f(X^{Ⓓ},\ Y^{Ⓓ})) \end{array}$$

$$
\begin{array}{lcl}
Call_p & \supseteq & p(f(X^{Ⓐ},\ Y^{Ⓐ})) \\
Call_q & \supseteq & q(X^{Ⓒ},\ Y^{Ⓒ}) \\
Ret_p & \supseteq & p(f(X^{Ⓓ},\ Y^{Ⓓ})) \\
Ret_q & \supseteq & q(a,b) \\
Ret_q & \supseteq & q(c,d) \\
\mathcal{X}^{Ⓐ} & \supseteq & \top \\
\mathcal{Y}^{Ⓐ} & \supseteq & \top \\
\mathcal{X}^{Ⓑ} & \supseteq & \{X : \exists Y\ (p(f(X,Y)) \in Call_p)\} \\
\mathcal{Y}^{Ⓑ} & \supseteq & \{Y : \exists X\ (p(f(X,Y)) \in Call_p)\} \\
\mathcal{X}^{Ⓒ} & \supseteq & \{X : \exists Y\ (p(f(X,Y)) \in Call_p)\} \\
\mathcal{Y}^{Ⓒ} & \supseteq & \{Y : \exists X\ (p(f(X,Y)) \in Call_p)\} \\
\mathcal{X}^{Ⓓ} & \supseteq & \{X : \exists Y\ (p(f(X,Y)) \in Call_p \ \wedge\ q(X,Y) \in Ret_q)\} \\
\mathcal{Y}^{Ⓓ} & \supseteq & \{Y : \exists X\ (p(f(X,Y)) \in Call_p \ \wedge\ q(X,Y) \in Ret_q)\}
\end{array}
$$

Figure 2.13: **Set Constraints for Program 6**

The complete constraints for the logic program appear in Figure 2.13, in which the symbol $\top$ denotes the set of all values. Note that there are no constraints for points Ⓔ and Ⓕ because there are no variables in the rules in which these points appear. The least assignment of sets to set variables that satisfies the constraints is given by:

$$
\begin{array}{lcl}
Call_p & \mapsto & \{p(f(s_1,s_2)) : s_1 \text{ and } s_2 \text{ are values}\} \\
Call_q & \mapsto & \{q(s_1,s_2) : s_1 \text{ and } s_2 \text{ are values}\} \\
Ret_p & \mapsto & \{p(f(a,b)), p(f(a,d)), p(f(c,b)), p(f(c,d))\} \\
Ret_q & \mapsto & \{q(a,b), q(a,d), q(c,b), q(c,d)\}
\end{array}
$$

$$
\begin{array}{lcllcl}
\mathcal{X}^{Ⓐ} & \mapsto & \{s : s \text{ is a value}\} & \mathcal{Y}^{Ⓐ} & \mapsto & \{s : s \text{ is a value}\} \\
\mathcal{X}^{Ⓑ} & \mapsto & \{a,c\} & \mathcal{Y}^{Ⓑ} & \mapsto & \{b,d\} \\
\mathcal{X}^{Ⓒ} & \mapsto & \{s : s \text{ is a value}\} & \mathcal{Y}^{Ⓒ} & \mapsto & \{s : s \text{ is a value}\}. \\
\mathcal{X}^{Ⓓ} & \mapsto & \{a,c\} & \mathcal{Y}^{Ⓓ} & \mapsto & \{b,d\}
\end{array}
$$

Solving Set Constraints

The purpose of set constraints is to capture consistency conditions between set variables in such a way that the relationships between the variables in a program are safely approximated. In other words, the constraints are constructed so that any assignment to the set variables that satisfies the constraints is correct in the following sense: if $\mathcal{X}^{\mu}$ is the set variable corresponding to program variable X at point μ, and if program variable X assumes value v at point μ during some program execution, then v appears in the set assigned to $\mathcal{X}^{\mu}$. Although any model of the constraints yields correct information, the least model (which is guaranteed to exist) is preferred because it is the most accurate and has a canonical definition. Thus, the problem of analyzing a program can be reduced to computing the least model of a collection of set constraints.

Since the least model of such constraints may be infinite, computing this model entails constructing a representation of a potentially infinite object. Moreover, for such a representation to be useful, it must be explicit in the sense that the structure of the model is self-evident and questions relating the membership and non-emptiness can easily be answered. A key result of this thesis is that the sets assigned to variables in the least model of a collection of set constraints are *regular* sets of terms in the sense that they can be described by regular term grammars. Regular term grammars are a generalization of regular grammars to terms. For example, the regular term grammar

$$\begin{array}{lcl} L & \Longrightarrow & nil \\ L & \Longrightarrow & cons(1, L) \end{array}$$

defines the set of all lists of 1's. Regular term grammars form the core of our explicit representation of models. Specifically, the algorithm presented in this thesis for solving set constraints, inputs a collection of constraints constructed from a program, and outputs, for each set variable appearing in the constraints, a regular term grammar describing of the set of terms assigned to that variable in the least model of the constraints.

In summary, the process of constructing set constraints from a program is essentially just setting up the analysis problem, and is analogous to writing out the definition of the semantic operator $\mathcal{F}$ in an abstract interpretation based analysis. The real work of analyzing the program is carried out by the algorithm to construct a representation of the least model of the constraints. This latter process of solving the constraints takes the place of the iterative fixed point computation in an abstract interpretation style analysis. Note that in abstract interpretation style analysis, the use of constraints is somewhat optional – although the constraints are always implicitly present, they do not have to be explicitly constructed. On the other hand, set based analysis places a greater emphasis on constraints. This is mainly because the algorithm for solving the constraints is not based on the notion of locally propagating information from one program point to another, but rather the algorithm reasons about the program as a whole. Such reasoning is most conveniently carried out using constraints.

2.6 Overview of Thesis

This thesis is structured in three parts. Part I deals with the definition of set based analysis. It justifies the process of constructing set constraints from a program that has been informally outlined in the previous section, and shows how these constraints correspond to reasoning about a program by ignoring inter-variable dependencies. Starting with an operational semantics, we define the notion of collecting semantics. This collecting semantics is defined directly in terms of the operational semantics by simply collecting together the appropriate objects (environments or term equations) for each program point encountered during program execution. Although such a definition of collecting semantics is simple and natural, it sheds little light onto how collecting semantics may be approximated and computed. The next step is therefore a constraint formulation of the collecting semantics. Given a program, we show how *environment constraints* may be constructed such that the least model of the environment constraints corresponds to the program's collecting semantics. The main advantage of the environment constraints is that they can be re-interpreted in a number of different ways. Such an alternative interpretation is used to define set based program approximation. In essence, we show how the constraints may be interpreted so that inter-variable dependencies may be ignored through a process of treating program variables as sets. Then, the set based approximation of a program is defined to be the smallest such "set" interpretation that is a model of the constraints. That is, the least (standard) model of the environment constraints gives the program's collecting semantics, and the least set model gives the set based approximation of the program.

The same basic plan of operational semantics, collecting semantics, environment constraints and set based approximation is carried out for both imperative and logic programs (under a variety of execution strategies). The operational semantics, collecting semantics and environment constraints for imperative programs are given in Chapter 3. The corresponding chapter for logic programs is Chapter 4. Chapter 5 defines the set based interpretation of the environment constraints, and thus defines set based program approximation.

Part II describes how set based approximations may be computed. Chapter 6 introduces set constraints, which is the key formalism for computing set based approximations. Most importantly, this chapter shows how environment constraints may be converted to set constraints such that the least set model of the environment constraints corresponds to the least model of the set constraints. In other words, this translation shows how the set based approximation of a program may be represented as the least model of a collection of set constraints. Chapter 7 then presents an algorithm for solving set constraints. This algorithm is presented in a number of stages. First, a generic set constraint algorithm is presented that abstracts the key features of the algorithm. Then, an instance of the algorithm is defined that deals with intersection and projection. Finally the complete algorithm is presented for solving the kinds of set constraints obtained when environment constraints are translated to set constraints. The last chapter of part II describes experience with a prototype implementation. A naive implementation of the basic set constraint algorithm is completely unusable except for very small collections of constraints. However substantial progress has been made by using specialized representation techniques and dealing with the redundancy that is typically present. Although much work remains, the results obtained so far demonstrate that practicality is within reach.

Part III describes some extensions to set based analysis. While the main body of this thesis deals with the problem of analyzing a program to determine the possible values that variables can be bound to during program execution, many of the techniques developed can be applied to other analysis problems. In particular, many of the algorithms preserve numerous structural properties of a program, and it is in fact easy to modify the algorithms to compute instantiation information (for logic programs) as well as information about sharing. This is the subject of Chapter 9. Chapter

10 shows how the techniques of set based analysis can be extended in another direction – by adding a limited ability to reason about inter-variable dependencies. The motivation for this work is that some kinds of analysis require both accurate treatment of data-structures as well as reasoning about inter-variable dependencies. The algorithm presented in this chapter combines the ability of set constraints to reason about data-structures can be combined with the ability of abstract interpretation to reason about inter-variable dependencies in a way that is more accurate than just running both algorithms. The last chapter of part III is Chapter 11 where we outline the application of set based analysis to functional programs, with particular focus on the language ML. This chapter is largely illustrative, showing connections between set based analysis and type inference in subtype systems, as well as relationships to control flow analysis.

Much of the work in this thesis unifies and extends joint work in earlier papers such as [21, 22, 23, 24, 25, 26]. For details about these papers and how they related to this thesis, and for details about related work by other authors, see Sections 5.6 and 7.7 (the former deals with work in the area of program approximations, and the latter deals with work in the area of decidability results and algorithms for set constraints).

Part I

Set-Based Approximation

The general scheme for obtaining set based approximations can be described as follows. Starting with a collecting semantics for a program, the first step is to characterize this semantics in terms of consistency conditions between the collections of environments identified by the semantics. This is achieved by introducing a variable to represent each environment collection, and then constructing *environment constraints* on these variables to capture consistency conditions between neighboring program points, in such a way that the least model of these constraints is exactly the collecting semantics. The next step consists of reinterpreting the environment constraints of the program so that each environment variable becomes a mapping from program variables into sets of values. The *set based approximation* of a program is then defined to be the least model of the environment constraints under this new interpretation.

This part consists of four chapters. The first two chapters present a variety of operational semantics and corresponding collecting semantics and environment constraints for the two main language paradigms considered in this thesis – imperative and logic programs. The purpose of these chapters is to provide essential definitions. Most of the ideas they contain have appeared elsewhere in the literature in one form or another. One exception is perhaps the heavy emphasis on constraints, in contrast with the more usual denotational semantics approach. The third chapter in this part contains the definition of *set based approximation* and is the core chapter of this part. The concluding chapter contains a discussion of this definition.

Chapter 3

Imperative Programs

We consider a simple imperative language over data structures with assignment, conditional and iteration statements. An operational semantics for this language is presented using a rewrite relation. A collecting semantics is then defined by specifying an appropriate notion of program point, and then projecting the operational semantics onto these points. Although this definition of collecting semantics is a very natural one in the sense that it directly captures the notion of what "happens" at each program point, it provides little insight into how a program may be analyzed. This motivates an alternative characterization of the collecting semantics using *environment constraints* that express notions of local consistency between neighboring program points. The environment constraints are similar in spirit to equational formulations of collecting semantics used widely in the program analysis literature.

3.1 Imperative Programs

The underlying values of the language are data structures. Specifically, there are constructors such as *nil* and *cons* to build up data structures, projections such as *car* and *cdr* to decompose data structures, and some basic primitives for testing the outermost constructor of a term. The language is untyped. We now describe the details.

Let VAR and Σ be disjoint sets, respectively denoting the set of program variables and the set of data constructors. Each data constructor is assumed to have a unique arity. A data constructor with arity 0 is called a *constant*. Corresponding to each data constructor f of arity $n \geq 1$, there are n projection operations, denoted $f^{-1}_{(1)}, \ldots, f^{-1}_{(n)}$. For example, *car* and *cdr* may be denoted by $cons^{-1}_{(1)}$ and $cons^{-1}_{(2)}$ respectively. An (imperative program) *term* is either a variable from VAR or of the form $f(t_1, \ldots, t_n)$ or $f^{-1}_{(j)}(t_1)$, where $n \geq 0$, f is an n-ary data constructor from Σ, each t_i is a term, and in the projection case, $1 \leq j \leq n$. An *atomic program condition* is of the form $s = t$ or $match_f(t)$ where f is a data constructor from Σ and s and t are program terms constructed from program variables and projection symbols. A *program condition* is any combination of atomic program conditions using the usual boolean connectives $\wedge$, $\vee$ and $\neg$.

An *imperative program* P is a sequence of program statements, *Seq*, defined as follows

Stat	::=	$X{:=}t$
	|	**if** *cond* **then** *Seq*
	|	**while** *cond* **do** *Seq*
Seq	::=	*Stat*
	|	$Seq_1 ; Seq_2$

where ";" is an associative sequencing operator. Figure 3.1 contains an example imperative program that computes the last element of the list *a.b.nil*. If *Stat* is of the form **if** *cond* **then** *Seq* or **while** *cond* **do** *Seq*, then *Seq* is called the *body* of *Stat*. The expressions *first*(*Seq*), *second*(*Seq*) and *last*(*Seq*) respectively denote the first, second and last statements in *Seq*, if they exist. The expression *tail*(*Seq*) denotes the sequence Seq' whenever *Seq* is of the form $Stat; Seq'$. In the context of a program P, each statement occurrence in P is assumed to be labeled with a unique integer; labels are denoted by

1. $L := cons(a, cons(b, nil))$;
2. $X := c$;
3. **while** $(match_{cons}(L))$ **do**
4. $\quad X := car(L)$;
5. $\quad L := cdr(L)$;

Figure 3.1: **Program 2 (Revisited)**

α, β, γ (possibly subscripted). Writing $Stat^{\alpha}$ denotes the unique statement occurrence in P with label α. Writing ${}^{\alpha}Seq^{\beta}$ denotes that Seq is a sequence of statements such that $first(Seq)$ is labeled with α and $last(Seq)$ is labeled with β. If $Stat^{\alpha}$ and $Stat^{\beta}$ appear as statements somewhere in P and $Stat^{\beta}$ appears immediately after $Stat^{\alpha}$, then $Stat^{\alpha}$ and $Stat^{\beta}$ are *consecutive statements* in P. For example the program in Figure 3.1, statements 1 and 2 are consecutive statements and so are statements 4 and 5.

3.2 Operational Semantics

We now present an operational semantics for imperative programs. A *value* is a program term that contains only symbols from Σ. For example, nil, $cons(a, nil)$ and $cons(a, b)$ are values, but $cdr(cons(b, nil))$ and $cons(X, Y)$ are not. An *environment* ρ is a mapping from VAR into values. We shall write $[X_1{\mapsto}v_1, \ldots, X_n{\mapsto}v_n]$ to denote an environment that maps X_i into v_i, $i = 1..n$. The expression $\rho[X{\mapsto}v]$ denotes the environment that maps X into v and maps all other variables Y into $\rho(Y)$. An environment can be extended to become a partial function from program terms t to values as follows:

- $\rho(f(t_1, \ldots, t_n)) \;=\; f(\rho(t_1), \ldots, \rho(t_n))$.
- $f^{-1}_{(i)}(t') \;=\; v_i$ if $\rho(t') = f(v_1, \ldots, v_n)$ for some values $v_1, \ldots, v_n$.

For some program terms t, such as $cons(car(nil), Y)$, $\rho(t)$ is not defined. The notation $\rho \rhd t$ shall be used to indicate that $\rho(t)$ is defined.

Note that environments are defined to be total functions from (the possibly infinite set) VAR into values. The reason for adopting this somewhat non-standard definition is twofold. First, it leads to greater uniformity in later definitions, such as those dealing with the semantics of logic programs (Chapter 4). Second, this definition means that all environments have the same fixed domain, and this results in some significant simplifications in later definitions, particularly those involving environment constraints and the translation of environment constraints to set constraints.

We now specify the meaning of program conditions. First define that a program condition *cond* is defined under ρ, denoted $\rho \rhd cond$, if $\rho \rhd t$ for each program term t appearing in *cond*. Now, for environments ρ such that $\rho \rhd cond$, define the relation $\rho \models cond$ as follows.

- $\rho \models s = t$ iff $\rho(s) = \rho(t)$.
- $\rho \models match_f(t)$ iff $\rho(t)$ is of the form $f(v_1, \ldots, v_n)$.
- $\rho \models cond_1 \wedge cond_2$ iff $\rho \models cond_1$ and $\rho \models cond_2$.
- $\rho \models cond_1 \vee cond_2$ iff either $\rho \models cond_1$ or $\rho \models cond_2$.
- $\rho \models \neg cond$ iff it is not the case that $\rho \models cond$.

Since $\rho \models cond$ is only defined if $\rho \rhd cond$, both $\rho \models car(nil)$ and $\rho \models \neg car(nil)$ are undefined. In what follows, we shall only write the expression $\rho \models cond$ when it is clear from context that $\rho \rhd cond$.

A *state* is a pair of the form $\langle \rho : Seq \rangle$ where ρ is an environment and *Seq* is either a sequence of statements or the special symbol *empty* denoting the empty sequence of statements. The sequencing operator ";" is extended in the obvious way to deal with *empty*: $Seq; empty = Seq = empty; Seq$. The meaning of an imperative program is defined via a rewrite relation on states.

$$\langle \rho : X := t; Seq\rangle \rightarrow \langle \rho[X \mapsto \rho(t)] : Seq\rangle \qquad \text{if } \rho \rhd t.$$

$$\langle \rho : (\textbf{if } cond \textbf{ then } Seq'); Seq\rangle \rightarrow \langle \rho : Seq'; Seq\rangle \qquad \begin{array}{l}\text{if } \rho \rhd cond\\ \text{and } \rho \models cond.\end{array}$$

$$\langle \rho : (\textbf{if } cond \textbf{ then } Seq'); Seq\rangle \rightarrow \langle \rho : Seq\rangle \qquad \begin{array}{l}\text{if } \rho \rhd cond\\ \text{and } \rho \models \neg cond.\end{array}$$

$$\begin{array}{l}\langle \rho : (\textbf{while } cond \textbf{ do } Seq'); Seq\rangle\\ \qquad \rightarrow \langle \rho : Seq'; (\textbf{while } cond \textbf{ do } Seq'); Seq\rangle\end{array} \qquad \begin{array}{l}\rho \rhd cond\\ \text{and } \rho \models cond.\end{array}$$

$$\langle \rho : (\textbf{while } cond \textbf{ do } Seq'); Seq\rangle \rightarrow \langle \rho : Seq\rangle \qquad \begin{array}{l}\text{if } \rho \rhd cond\\ \text{and } \rho \models \neg cond.\end{array}$$

A *derivation* is a sequence of rewrite steps of the form

$$\langle \rho_0 : Seq_0\rangle \rightarrow \langle \rho_1 : Seq_1\rangle \rightarrow \cdots \rightarrow \langle \rho_{n-1} : Seq_{n-1}\rangle \rightarrow \langle \rho_n : Seq_n\rangle.$$

We frequently write $\langle \rho_0 : Seq_0\rangle \rightarrow^\star \langle \rho_n : Seq_n\rangle$ or $\langle \rho_0 : Seq_0\rangle \rightarrow^n \langle \rho_n : Seq_n\rangle$ to denote the existence of such a derivation from $\langle \rho_0 : Seq_0\rangle$ to $\langle \rho_n : Seq_n\rangle$. We shall consistently use ρ_0 to denote the starting environment of a derivation.

The program *terminates* on environment ρ_0 if there is a maximal derivation of the form $\langle \rho_0 : P\rangle \rightarrow^\star \langle \rho : Seq\rangle$. Since there is at most one rewrite step applicable to a state, there is at most one such derivation. If the final state is of the form $\langle \rho : empty\rangle$ then the program is said to *terminate with environment* ρ. If the final state is not of this form then the program is said to *terminate with an error*. This is the case, for example, if the program attempts to evaluate a term such as $car(nil)$. There is no special "error" value, but rather an error corresponds to a state from which no transition is possible.

Although statement labels have been ignored in the above definition of rewrite steps and derivations, it is straightforward to extend the definitions so that $\rightarrow$ relates states of the form $\langle \rho : Seq\rangle$ where Seq is a sequence of *labeled* statements. For example,

$$\begin{array}{rcl}\big\langle (X := 1)^a; (Y := b)^2 : \rho\big\rangle & \rightarrow & \big\langle (Y := b)^2 : \rho[X \mapsto a]\big\rangle\\ & \rightarrow & \langle empty : \rho[X \mapsto a][Y \mapsto b]\rangle\end{array}$$

is a valid derivation. In what follows, we shall implicitly assume that derivation involve labeled statements.

3.3 Collecting Semantics

The operational semantics described in the previous section defines how a program is executed. Importantly, given a program P and a starting environment ρ_0, it defines the environment resulting from the computation of P starting with ρ_0. That is, the operational semantics can be thought of as a mapping from a program P into its meaning $[\![P]\!]$ where $[\![P]\!](\rho_0)$ is the environment ρ such that $\langle P : \rho_0 \rangle \rightarrow^\star \langle \mathit{empty} : \rho \rangle$. Note that $[\![P]\!]$ is, in general, a partial function.

However, such a view of the operational semantics does not say anything about what happens *during* the execution of the program. For example, it does not describe the values that a program variable may take during execution. Such information is clearly central to program analysis. What is required therefore, is a view of the operational semantics that makes explicit what happens *during* program execution. That is, we need to collect the environments encountered at each point in the program during the program execution, and this is called a *collecting semantics.*

The notion of collecting semantics is the starting point of all formal treatments of program analysis. Our collecting semantics is just an explication of information already implicit in the operational semantics. In essence, the operational semantics is projected onto the notion of program point. As an aside, we note that since a collecting semantics describes what happens part way through a computation (including computations that lead to an error or do not terminate), a natural semantics style presentation of the operational semantics [37, 54] would be significantly less convenient than the transition style system we have employed.

We first establish a notation for referring to points in a program. As mentioned earlier, we assume that each statement occurrence in a program is uniquely labeled. Now, with each statement occurrence Stat^α, we associate two program points $\uparrow\alpha$ and $\downarrow\alpha$ to respectively indicate the execution states just before and just after Stat^α is executed. A *collecting interpretation* for an imperative program is an association of a collection of environments to each program point. The *collecting semantics* of an imperative program is the specific collecting interpretation that associates with each program point the collection of environments encountered at that point during program execution.

To see how the collecting semantics may be formalized, first observe that the execution of a program P starting with environment ρ_0 is completely characterized by the maximal derivation of the form

$$\langle \rho_0 : P \rangle \;\rightarrow\; \langle \rho_1 : Seq_1 \rangle \;\rightarrow\; \langle \rho_2 : Seq_2 \rangle \;\rightarrow\; \cdots \tag{3.1}$$

Now, consider collecting the environments for a program point of the form $\uparrow\!\alpha$. That is, we wish to collect all of the environments encountered in the derivation (3.1) just before statement α is executed. This can be simply stated as:

$$\{\rho : \langle \rho_0 : P \rangle \rightarrow^{\star} \langle \rho : Stat^{\alpha} ; Seq \rangle\}\,.$$

The collection of environments for a program point $\downarrow\!\alpha$ is more involved because first the notion of "just after statement execution" must be formalized. To this end, define a reflexive transitive ordering on sequences of statements: $Seq_1 \geq Seq_2$ if

Seq_1 is of the form $Seq;Seq_2$ for some Seq.

In other words $Seq_1 \geq Seq_2$ if Seq_2 is equal to Seq_1 or else Seq_2 is a final subsequence of Seq_1. For example $Stat_1;Stat_2;Stat_3 \geq Stat_2;Stat_3$, but it is not the case that $Stat_1;Stat_2;Stat_3 \geq Stat_1;Stat_2$. Also define that $Seq_1 > Seq_2$ if $Seq_1 \geq Seq_2$ and $Seq_1 \neq Seq_2$. Now, suppose that derivation (3.1) has the form:

$$\langle \rho_0 : P \rangle \rightarrow^{n} \langle \rho_n : Stat^{\alpha} ; Seq \rangle \rightarrow \langle \rho_{n+1} : Seq_{n+1} \rangle \rightarrow^{i-1} \langle \rho_{n+i} : Seq_{n+i} \rangle \rightarrow \cdots$$

The transition from state $\langle \rho_n : Stat^{\alpha} ; Seq \rangle$ starts an execution of statement $Stat^{\alpha}$. Now, the execution of this statement could be completed in one step (this is the case, for example, if $Stat^{\alpha}$ is an assignment), and then the state $\langle \rho_{n+1} : Seq_{n+1} \rangle$ is in fact $\langle \rho_{n+1} : Seq \rangle$. On the other hand, $Stat^{\alpha}$ may take more than one step to execute (this is the case, for example, if $Stat^{\alpha}$ is a **while-do** statement whose condition is satisfied by ρ_n), and then the state $\langle \rho_{n+1} : Seq_{n+1} \rangle$ will be such that $Seq_{n+1} > Seq$. In fact the execution of $Stat^{\alpha}$ continues while the subsequent states $\langle \rho_{n+i} : Seq_{n+i} \rangle$ are such that $Seq_{n+i} > Seq$. Two possibilities arise: either (a) $Seq_{n+i} > Seq$ for all i, and $Stat^\alpha$ never completes execution, or (b) $Seq_{n+j} = Seq$ for some $j \geq 1$, and this means that $Stat^\alpha$ completes execution when state $\langle \rho_{n+j} : Seq \rangle$ is reached. In case (b), ρ_{n+j} is an environment encountered just after $Stat^\alpha$ has completed execution. So, corresponding to each point $\downarrow\alpha$, we wish to collect environments ρ_{n+j} such that

$$\langle \rho_0 : P \rangle \to^n \langle \rho_n : Stat^\alpha ; Seq \rangle \to^j \left\langle \rho_{n+j} : Seq_{n+j} \right\rangle \tag{3.2}$$

where Seq_{n+j} is Seq and $Seq_{n+i} > Seq$ for $1 \leq i < j$. For notational convenience, we write $\langle \rho_1 : Seq_1 \rangle \to^m_{Seq} \langle \rho_m : Seq_m \rangle$ if there exists a derivation $\langle \rho_1 : Seq_1 \rangle \to \cdots \to \langle \rho_m : Seq_m \rangle$ where $Seq_i > Seq$, $1 \leq i < m$. (We shall also sometimes omit the length of the derivation and just write $\langle \rho_1 : Seq_1 \rangle \to^\star_{Seq} \langle \rho_m : Seq_m \rangle$.) Using this notation, the set of environments defined by (3.2) can be more concisely described as the set of ρ such that

$$\langle \rho_0 : P \rangle \to^n \langle \rho' : Stat^\alpha ; Seq \rangle \to^j_{Seq} \langle \rho : Seq \rangle$$

To give some intuition about this definition, note that the property $\langle \rho' : Stat^\alpha ; Seq \rangle \to^j_{Seq} \langle \rho : Seq \rangle$ holds iff there exist environments $\rho_2, \ldots, \rho_{j-1}$ and statement sequences $Seq_2, \ldots, Seq_{j-1}$ such that

$$\langle \rho' : Stat^\alpha ; Seq \rangle \to \langle \rho_2 : Seq_2 ; Seq \rangle \to \cdots \to \left\langle \rho_{j-1} : Seq_{j-1} ; Seq \right\rangle \to \langle \rho : Seq \rangle.$$

Moreover, from the definition of $\to$, this is a derivation iff $\langle \rho' : Stat^\alpha \rangle \to \langle \rho_2 : Seq_2 \rangle \to \cdots \to \left\langle \rho_{j-1} : Seq_{j-1} \right\rangle \to \langle \rho : empty \rangle$. Hence (3.2) is equivalent to the existence of two derivations

$$\langle \rho_0 : P \rangle \to^n \langle \rho_n : Stat^\alpha ; Seq \rangle \quad \text{and} \quad \langle \rho_n : Stat^\alpha \rangle \to^j \langle \rho_{n+j} : empty \rangle$$

where the first derivation corresponds to execution reaching statement $Stat^\alpha$ and the second corresponds to the execution of $Stat^\alpha$.

Before presenting the complete collecting semantics, we address the issue of starting environments. In the operational semantics, it was appropriate to define program execution from some given starting environment ρ_0. This was carried over in the above discussion of collecting semantics. However, when performing analysis of a program, the initial environment may not be known. This issue may be addressed in a number of ways. First, program execution could be defined to start in a fixed initial environment (which, say, maps all variables to *nil*). Second, programs could be defined to begin with a sequence of assignment statements that initialize all program

points	environments
$\uparrow 1$	{all environments}
$\downarrow 1$, $\uparrow 2$	$\{[X \mapsto a.b.nil, Y \mapsto v]$: for any value $v\}$
$\downarrow 2$	$\{[X \mapsto a.b.nil, Y \mapsto c]\}$
$\uparrow 3$	$\{[X \mapsto a.b.nil, Y \mapsto c], [X \mapsto b.nil, Y \mapsto a], [X \mapsto nil, Y \mapsto b]\}$
$\uparrow 4$	$\{[X \mapsto a.b.nil, Y \mapsto c], [X \mapsto b.nil, Y \mapsto a]\}$
$\downarrow 4$, $\uparrow 5$	$\{[X \mapsto a.b.nil, Y \mapsto a], [X \mapsto b.nil, Y \mapsto b]\}$
$\downarrow 5$	$\{[X \mapsto b.nil, Y \mapsto a], [X \mapsto nil, Y \mapsto b]\}$
$\downarrow 3$	$\{[X \mapsto nil, Y \mapsto b]\}$

Figure 3.2: **Collecting Semantics for Program 2**

variables, and then the initial environment is essentially irrelevant. Third, the collecting semantics could be defined with respect to a set S of starting environments (although note that this introduces the issue of how S is represented). Fourth, collecting semantics could be defined to be the environments encountered over executions from *all* possible starting environments. Of these four possibilities, the last two are the most reasonable. For simplicity, we choose the last one, although note that the definitions and algorithms presented in this thesis can easily be adapted to deal with the third possibility if S is represented using regular tree grammars (see Section 7.1). The collecting semantics of an imperative program P can now be presented.

Definition 1 *The* <u>collecting semantics</u> $\mathcal{CS}_P$ *of an imperative program* P *is the mapping:*

$$\begin{array}{lll} \uparrow\alpha & \mapsto & \left\{\rho : \exists \rho_0 \ s.t. \ \langle \rho_0 : P\rangle \rightarrow^\star \langle \rho : Stat^\alpha ; Seq\rangle \right\}, \\ \downarrow\alpha & \mapsto & \left\{\rho : \exists \rho_0 \ s.t. \ \langle \rho_0 : P\rangle \rightarrow^\star \langle \rho' : Stat^\alpha ; Seq\rangle \rightarrow^\star_{Seq} \langle \rho : Seq\rangle \right\} \end{array} \quad \square$$

Figure 3.2 presents the collecting semantics for Program 2 (see Figure 3.1).

A definition of collecting semantics is the starting point of program analysis. In particular it provides the primary definition of correctness: an (approximate) analysis is *correct* if it yields a *conservative approximation* of the collecting semantics. In other words, an analysis is correct if, for each program point, the set of environments described by the analysis for that point is a superset of the set of environments described by the collecting semantics. However, this definition of correctness provides little insight into how program analysis might be performed. We therefore present an alternative definition of collecting semantics that provides a more concrete basis for computing information about the collecting semantics.

3.4 Environment Constraints

In essence, environment constraints characterize the collecting semantics of a program by capturing a notion of "local consistency" between the collections of environments associated with neighboring program points. We begin by defining the general form and interpretation of the constraints used. The following definitions are made in the context of a program P. An *environment variable* is a variable that ranges over sets of environments, and shall be denoted by the symbol Ψ. For each program point μ, there is a distinguished environment variable denoted Ψ^μ, whose purpose is to describe the environments corresponding to point μ. An *environment expression* is either an environment variable or an expression of the form $\top$, $\Psi[X \mapsto t]$ or $\Psi[cond]$, where Ψ is an environment variable, X is a program variable, t is a program term and *cond* is a program condition. Informally, $\top$ denotes all environments, $\Psi[X \mapsto t]$ is used to model assignment statements and $\Psi[cond]$ is used to model **if-then** and **while-do** statements. An *environment constraint* is of the form $\Psi \supseteq ee$ where Ψ is an environment variable and *ee* is an environment expression.

The meaning of environment expressions and constraints is defined in the context of an interpretation $\mathcal{I}$ that maps each environment variable into a set of environments. Such a mapping $\mathcal{I}$ is extended to map from environment expressions into sets of environments as follows:

- $\mathcal{I}(\top) \;=\; \{\text{all environments}\}$.
- $\mathcal{I}(\Psi[X \mapsto t]) \;=\; \{\rho[X{\mapsto}\rho(t)] : \rho \in \Theta\}$ where Θ is $\{\rho \in \mathcal{I}(\Psi) : \rho \rhd t\}$.
- $\mathcal{I}(\Psi[cond]) \;=\; \{\rho \in \Theta : \rho \models cond\}$ where Θ is $\{\rho \in \mathcal{I}(\Psi) : \rho \rhd cond\}$.

An interpretation $\mathcal{I}$ is a model of a collection of environment constraints if, for each constraint $\Psi \supseteq ee$ in the collection, $\mathcal{I}(\Psi) \supseteq \mathcal{I}(ee)$. Interpretations of environment constraints are ordered componentwise: $\mathcal{I} \subseteq \mathcal{I}'$ if $\mathcal{I}(\Psi) \subseteq \mathcal{I}'(\Psi)$ for all environment variables Ψ.

Now, corresponding to an imperative program P, we construct environment constraints to capture the local consistency conditions between neighboring points in the program.

Definition 2 *The* environment constraints $\mathcal{EC}_P$ *corresponding to an imperative program* P *consist of the following collections of constraints:*

a) $\Psi^{\uparrow 1} \supseteq \top$;

b) $\Psi^{\uparrow\beta} \supseteq \Psi^{\downarrow\alpha}$ *if* $Stat^\alpha$ *and* $Stat^\beta$ *are consecutive statements in* P;

c) $\Psi^{\downarrow\alpha} \supseteq \Psi^{\uparrow\alpha}[X \mapsto t]$ *for each* $Stat^\alpha$ *in* P *of the form* X :=t;

d) $\left.\begin{array}{l} \Psi^{\uparrow\beta} \supseteq \Psi^{\uparrow\alpha}[cond] \\ \Psi^{\downarrow\alpha} \supseteq \Psi^{\uparrow\alpha}[\neg cond] \\ \Psi^{\downarrow\alpha} \supseteq \Psi^{\downarrow\gamma} \end{array}\right\}$ *for each* $Stat^\alpha$ *in* P *of the form* **if** cond **then** ${}^\beta Seq^\gamma$;

e) $\left.\begin{array}{l} \Psi^{\uparrow\alpha} \supseteq \Psi^{\downarrow\gamma} \\ \Psi^{\uparrow\beta} \supseteq \Psi^{\uparrow\alpha}[cond] \\ \Psi^{\downarrow\alpha} \supseteq \Psi^{\uparrow\alpha}[\neg cond] \end{array}\right\}$ *for each* $Stat^\alpha$ *in* P *of the form* **while** cond **do** ${}^\beta Seq^\gamma$;

□

Before giving a formal statement of the correctness of these constraints, we shall first provide some motivation for their construction. The constraint in (a) corresponds to the adopted convention that programs start in an arbitrary environment. The constraint in (b) expresses a simple containment relationship for consecutive statements. The constraints in (c), (d) and (e) correspond to assignment, conditional and iterative statements in the program.

For example, consider again Program 2 from Figure 3.1. The constraint corresponding to the statement $X := car(L)$ is $\Psi^{\downarrow 4} \supseteq \Psi^{\uparrow 4}[X \mapsto cons^{-1}_{(1)}(L)]$, denoting that $\Psi^{\downarrow 4}$ contains the environments from $\Psi^{\uparrow 4}$ after they are modified to map X into the value of $car(X)$. The complete collection of environment constraints for this program appears in figure 3.3.

Environment constraints are stated as set containment relationships instead of set equalities because containment leads to a much more flexible

1. $L := cons(a, cons(b, nil));$
2. $X := c;$
3. **while** $(match_{cons}(L))$ **do**
4. $\quad X := car(L);$
5. $\quad L := cdr(L);$

$$
\begin{array}{lcl}
\Psi^{\uparrow 1} & \supseteq & \top \\
\Psi^{\downarrow 1} & \supseteq & \Psi^{\uparrow 1}[L \mapsto a.b.nil] \\
\Psi^{\uparrow 2} & \supseteq & \Psi^{\downarrow 1} \\
\Psi^{\downarrow 2} & \supseteq & \Psi^{\uparrow 2}[X \mapsto c] \\
\Psi^{\uparrow 3} & \supseteq & \Psi^{\downarrow 2} \\
\Psi^{\uparrow 3} & \supseteq & \Psi^{\downarrow 5} \\
\Psi^{\uparrow 4} & \supseteq & \Psi^{\uparrow 3}[match_{cons}(L)] \\
\Psi^{\downarrow 4} & \supseteq & \Psi^{\uparrow 4}[X \mapsto cons^{-1}_{(1)}(L)] \\
\Psi^{\uparrow 5} & \supseteq & \Psi^{\downarrow 4} \\
\Psi^{\downarrow 5} & \supseteq & \Psi^{\uparrow 5}[L \mapsto cons^{-1}_{(2)}(L)] \\
\Psi^{\downarrow 3} & \supseteq & \Psi^{\uparrow 3}[\neg match_{cons}(L)]
\end{array}
$$

Figure 3.3: **Program 2 and Its Environment Constraints**

definition. In particular, the use of containment leads to a very weak notion of local consistency, and so it admits the possibility of models of the constraints in which the environments associated to a program points may be *larger* than necessary. In doing so, it allows approximations of the collecting semantics to be models of the environment constraints. The use of equality would essentially exclude this possibility.

Importantly, the environment constraints $\mathcal{EC}_P$ of a program P possess a least model, denoted $lm(\mathcal{EC}_P)$. This follows from corollary 4 in Appendix I, noting that the operators of the environment constraint calculus – that is the constant $\top$ and the postfix operators $[X \mapsto t]$ and $[cond]$ – are all monotonic operators over sets of environments. This least model provides an alternative definition of collecting semantics.

Theorem 1 (Environment Constraint Correctness)
The collecting semantics of an imperative program P maps any program point μ into $lm(\mathcal{EC}_P)(\Psi^{\mu})$. $\square$

The proof of this theorem is developed in the next section. It is rather lengthy and tedious and is included mainly for the sake of completeness. Note that many accounts of program analysis in the literature simply start with an equational version of the collecting semantics, and so the step of proving the equivalence of the collecting semantics induced by an underlying operational semantics and the equational version of the collecting semantics is effectively bypassed. However, we believe that an operational semantics version of the collecting semantics is a more appropriate starting point for program analysis, and so the issue of proving the equivalence of the two formulations of collecting semantics must be addressed.

3.5 Environment Constraint Correctness

We begin with some initial properties of the operational semantics of imperative programs. The following two propositions are simple observations about the definition of $\rightarrow$. The first describes the ways in which a derivation step can change the sequence of statements in a state. The second shows that a statement must appear at the front of the statement sequence before it can be removed.

Proposition 1 *If* $\langle \rho_a : Seq_a \rangle \rightarrow \langle \rho_b : Seq_b \rangle$ *then either*

(a) $Seq_a = Stat;Seq_b$ *for some statement* $Stat$,

(b) $Seq_b = Seq;tail(Seq_a)$ *and* $first(Seq_a)$ *is a statement of the form* **if** $cond$ **then** Seq *such that* $\rho_a \rhd cond$ *and* $\rho_a \models cond$, *or*

(c) $Seq_b = Seq;Seq_a$ *and* $first(Seq_a)$ *is a statement of the form* **while** $cond$ **do** Seq *such that* $\rho_a \rhd cond$ *and* $\rho_a \models cond$. □

Proposition 2 *If* $\langle \rho_0 : Seq_0 \rangle \rightarrow \langle \rho_1 : Seq_1 \rangle \rightarrow \cdots \rightarrow \langle \rho_n : Seq_n \rangle$ *such that* $Seq_0 > Seq$ *and* $Seq_n \not> Seq$ *then, for some* $k \leq n$, $Seq_k = Seq$.

Proof: Proposition 1 implies that if $\langle \rho_a : Seq_a \rangle \rightarrow \langle \rho_b : Seq_b \rangle$ and $Seq_a > Seq$ then $Seq_b \geq Seq$. Consider applying this fact to the first step $\langle \rho_0 : Seq_0 \rangle \rightarrow \langle \rho_1 : Seq_1 \rangle$ in the above derivation. Since $Seq_0 > Seq$, the fact implies that $Seq_1 \geq Seq$. Hence, either $Seq_1 > Seq$ or $Seq_1 = Seq$. In the latter case the proposition is proved. In the former case, the fact can be applied again, this time to the step $\langle \rho_1 : Seq_1 \rangle \rightarrow \langle \rho_2 : Seq_2 \rangle$. Repeating this argument proves that either there is a k such that $Seq_k = Seq$, or else $Seq_n > Seq$. Since it is assumed that $Seq_n \not> Seq$, the proposition is proved. □

The next two propositions deal with statements *Stat* that are either **if-then** or **while-do** statements. They prove that the last statement in the body of such a statement *Stat* can only be introduced by an execution of *Stat*. These propositions provide an important connection between the environments encountered after execution of the last statement in the body of *Stat* and the environments encountered after the execution of *Stat* itself.

Proposition 3 *Let* $Stat^\alpha$ *be* **if** *cond* **then** *Seq and let* $last(Seq)$ *be* $Stat^\beta$. *If* $\langle \rho_0 : P\rangle \rightarrow^\star \left\langle \rho : Stat^\beta ; Seq' \right\rangle$ *then there exists an environment* ρ' *such that* $\rho' \vartriangleright cond$, $\rho' \models cond$ *and*

$$\langle \rho_0 : P\rangle \rightarrow^\star \langle \rho' : Stat^\alpha ; Seq'\rangle \rightarrow^\star_{Seq'} \left\langle \rho : Stat^\beta ; Seq' \right\rangle.$$

Proof: If $\langle \rho_0 : P\rangle \rightarrow^\star \left\langle \rho : Stat^\beta ; Seq' \right\rangle$ then there exists a derivation of the form

$$\langle \rho_0 : Seq_0\rangle \rightarrow \langle \rho_1 : Seq_1\rangle \rightarrow \cdots \rightarrow \langle \rho_{n-1} : Seq_{n-1}\rangle \rightarrow \langle \rho_n : Seq_n\rangle$$

where $\langle \rho_0 : Seq_0\rangle$ is $\langle \rho_0 : P\rangle$ and $\langle \rho_n : Seq_n\rangle$ is $\left\langle \rho : Stat^\beta ; Seq' \right\rangle$. Now, pick the largest i such that $Seq_i \not\geq Seq_n$. Such an i exists because Seq_0 (which is just P) cannot be of the form $Seq_a ; Stat^\beta ; Seq_b$, and so $Seq_0 \not\geq Seq_n$. Also i is less than n because $Seq_n \geq Seq_n$. By construction, $Seq_i \not\geq Seq_n$ and $Seq_{i+1} \geq Seq_n$. Hence the i^{th} step in the derivation must introduce the statement $Stat^\beta$. From Proposition 1, the only statement that could introduce $Stat^\beta$ is $Stat^\alpha$. Hence $first(Seq_i)$ must be $Stat^\alpha$, $\rho_i = \rho_{i+1}$, $\rho_i \vartriangleright cond$, $\rho_i \models cond$ and $Seq_{i+1} = Seq ; rest(Seq_i)$. Moreover, since $Seq_j \geq Seq_n$, $i < j \leq n$, and Seq_n is $Stat^\beta ; Seq'$, it must be the case that $Seq_j > Seq'$, $i < j \leq n$. In summary:

$$\langle \rho_0 : Seq_0\rangle \rightarrow^\star \langle \rho_i : Stat^\alpha ; Seq'\rangle \rightarrow \langle \rho_i : Seq ; Seq'\rangle \rightarrow^\star_{Seq'} \left\langle \rho : Stat^\beta ; Seq' \right\rangle.$$

□

Proposition 4 *Let* $Stat^\alpha$ *be* **while** *cond* **do** *Seq and let* $last(Seq)$ *be* $Stat^\beta$. *If* $\langle \rho_0 : P\rangle \rightarrow^\star \left\langle \rho : Stat^\beta ; Seq' \right\rangle$ *then* $first(Seq')$ *is* $Stat^\alpha$ *and there exists an environment* ρ' *such that* $\rho' \vartriangleright cond$, $\rho' \models cond$ *and*

$$\langle \rho_0 : P\rangle \rightarrow^\star \langle \rho' : Seq'\rangle \rightarrow^\star \left\langle \rho : Stat^\beta ; Seq' \right\rangle.$$

Proof: The proof similar to Proposition 3. Again, the assumptions of the proposition imply the existence of a derivation of the form

$$\langle \rho_0 : Seq_0 \rangle \rightarrow \langle \rho_1 : Seq_1 \rangle \rightarrow \cdots \rightarrow \langle \rho_{n-1} : Seq_{n-1} \rangle \rightarrow \langle \rho_n : Seq_n \rangle$$

where $\langle \rho_0 : Seq_0 \rangle$ is $\langle \rho_0 : P \rangle$ and $\langle \rho_n : Seq_n \rangle$ is $\left\langle \rho : Stat^{\beta} ; Seq' \right\rangle$. Pick the largest i such that $Seq_i \not\geq Seq_n$. By construction, $Seq_i \not\geq Seq_n$ and $Seq_{i+1} \geq Seq_n$. Hence the i^{th} step in the derivation must introduce the statement $Stat^{\beta}$. From proposition 1, it follows that $first(Seq_i)$ is $Stat^{\alpha}$, $\rho_i = \rho_{i+1}$, $\rho_i \rhd cond$, $\rho_i \models cond$ and $Seq_{i+1} = Seq ; Seq_i$. In summary:

$$\langle \rho_0 : Seq_0 \rangle \rightarrow^{\star} \langle \rho_i : Seq_i \rangle \rightarrow \langle \rho_i : Seq ; Seq_i \rangle \rightarrow^{\star} \left\langle \rho : Stat^{\beta} ; Seq' \right\rangle.$$

Now, since $Seq_i \not\geq (Stat^{\beta} ; Seq')$ and $(Seq ; Seq_i) \geq (Stat^{\beta} ; Seq')$, it follows that Seq must be of the form $Seq_a ; Seq_b$ such that $Seq_b ; Seq_i = Stat^{\beta} ; Seq'$ where Seq_b is not *empty*. Since $Stat^{\beta}$ is the last element in Seq (and occurs no where else in Seq), it follows that Seq_b is $Stat^{\beta}$. Hence $Seq_i = Seq'$, and the proposition is proved. $\square$

Proposition 5 *Let $Stat^{\alpha}$ and $Stat^{\beta}$ appear as consecutive statements in P. If $\langle \rho_0 : P \rangle \rightarrow^{\star} \langle \rho : Seq_a ; Stat^{\alpha} ; Seq_b \rangle$ then $first(Seq_b) = Stat^{\beta}$.*

Proof: From the definition of consecutive statement, it must be the case that either P or the body of some statement in P is of the form

$$Seq_1 ; Stat^{\alpha} ; Stat^{\beta} ; Seq_2.$$

Since each statement in P has a unique label, it follows that if P or the body of some statement in P is of the form $Seq_a ; Stat^{\alpha} ; Seq_b$, then $first(Seq_b)$ must be $Stat^{\beta}$. Using this observation, the proposition can be established by a simple induction argument on the length of derivations. In the base case of a length 0 derivation, the sequence $Seq_a ; Stat^{\alpha} ; Seq_b$ is just P, and so it is immediate that $first(Seq_b)$ is $Stat^{\beta}$.

Now suppose that the proposition holds for $\langle \rho_0 : P \rangle \rightarrow^n \langle \rho_n : Seq_n \rangle$, and suppose that

$$\langle \rho_0 : P \rangle \rightarrow^n \langle \rho_n : Seq_n \rangle \rightarrow \langle \rho_{n+1} : Seq_{n+1} \rangle.$$

Consider the three cases of $\langle \rho_n : Seq_n \rangle \rightarrow \langle \rho_{n+1} : Seq_{n+1} \rangle$ outlined in Proposition 1. In case (a), $Seq_n = (Stat ; Seq_{n+1})$, and so if Seq_{n+1} is of the form

$Seq_a;Stat^\alpha;Seq_b$ then Seq_n is $Stat;Seq_a;Stat^\alpha;Seq_b$, and so $first(Seq_b) = Stat^\beta$ follows directly from the induction hypothesis. In cases (b) and (c), Seq_{n+1} is $Seq;Seq'_{n+1}$ where Seq is the body of an **if-then** or **while-do** statement and Seq'_{n+1} is either $rest(Seq_n)$ or Seq_n. Now, if Seq_{n+1} is of the form $Seq_a;Stat^\alpha;Seq_b$ there are three possibilities: either (i) $Seq = Seq_a;Stat^\alpha;first(Seq_b);Seq'$ for some Seq', (ii) $Seq = Seq_a;Stat^\alpha$ or (iii) $Seq'_{n+1} = Seq';Stat^\alpha;Seq_b$ for some Seq'. In case (i), $first(Seq_b)$ is $Stat^\beta$ because Seq is the body of some statement in P. In case (ii), $Stat^\alpha$ is the last statement in Seq, which implies that $Stat^\alpha$ and $Stat^\beta$ cannot be consecutive statements in P, and so this case is not possible. In case (iii), $Seq';Stat^\alpha;Seq_b$ is a subsequence of Seq_n, and so the fact that $first(Seq_b)$ is $Stat^\beta$ follows from the induction hypothesis. $\square$

Now, the collecting semantics of a program P can be thought of as an interpretation of the environment constraints $\mathcal{EC}_P$. Specifically, let $\mathcal{I}_{cs}$ denote the interpretation that maps Ψ^μ into the set of environments associated with μ in the collecting semantics. Importantly, $\mathcal{I}_{cs}$ is not only an interpretation of $\mathcal{EC}_P$, but it is also a model of $\mathcal{EC}_P$.

Lemma 1 *$\mathcal{I}_{cs}$ is a model of $\mathcal{EC}_P$.*

Proof: Consider each possible form of constraint in $\mathcal{EC}_P$ in turn. First, consider a constraint of the form $\Psi^{\uparrow 1} \supseteq \top$. Such a constraint is trivially satisfied since $\mathcal{I}_{cs}(\Psi^{\uparrow 1})$ contains all environments, since the collecting semantics is defined to be the collections of environments encountered when the program is started in an arbitrary environment.

Second, consider a constraint of the form $\Psi^{\uparrow\beta} \supseteq \Psi^{\downarrow\alpha}$. Such a constraint is present in the environment constraints of P if $Stat^\alpha$ and $Stat^\beta$ are consecutive statements of P. Suppose that $\rho \in \mathcal{I}_{cs}(\Psi^{\downarrow\alpha})$. Then by definition, there exists an environment ρ_0 such that $\langle \rho_0 : P\rangle \rightarrow^\star \langle \rho' : Stat^\alpha;Seq\rangle \rightarrow^\star_{Seq} \langle \rho : Seq\rangle$. By Proposition 5, the first statement of Seq must be $Stat^\beta$, and hence $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\beta})$.

Third, consider a constraint of the form $\Psi^{\downarrow\alpha} \supseteq \Psi^{\uparrow\alpha}[X \mapsto t]$, corresponding to a statement $Stat^\alpha$ of the form $X := t$. Now, suppose that $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha}[X \mapsto t])$. Then, there is an environment ρ' such that $\rho' \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha})$, $\rho' \rhd t$ and ρ is $\rho'[X \mapsto v]$ where v is $\rho'(t)$. Hence $\langle \rho_0 : P\rangle \rightarrow^\star \langle \rho' : Stat^\alpha;Seq\rangle$ and $\langle \rho' : Stat^\alpha;Seq\rangle \rightarrow \langle \rho : Seq\rangle$. Combining these two facts proves that

$\langle \rho_0 : P \rangle \to^\star \langle \rho' : \mathit{Stat}^\alpha ; \mathit{Seq} \rangle \to \langle \rho : \mathit{Seq} \rangle$ and so $\rho \in \mathcal{I}_{cs}(\Psi^{\downarrow\alpha})$.

Fourth, consider a constraint of the form $\Psi^{\uparrow\beta} \supseteq \Psi^{\uparrow\alpha}[\mathit{cond}]$, corresponding to a statement Stat^α of the form **if** cond **then** ${}^\beta\mathit{Seq}^\gamma$. Now, suppose that $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha}[\mathit{cond}])$. Then $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha})$, $\rho \rhd \mathit{cond}$ and $\rho \models \mathit{cond}$. Hence $\langle \rho_0 : P \rangle \to^\star \langle \rho : \mathit{Seq} ; \mathit{Seq}' \rangle$ and so $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\beta})$ since the first statement in Seq is Stat^β.

Fifth, consider a constraint of the form $\Psi^{\downarrow\alpha} \supseteq \Psi^{\uparrow\alpha}[\neg\mathit{cond}]$, corresponding to a statement Stat^α of the form **if** cond **then** ${}^\beta\mathit{Seq}^\gamma$. Suppose that $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha}[\neg\mathit{cond}])$. This implies that $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha})$, $\rho \rhd \neg\mathit{cond}$ and $\rho \models \neg\mathit{cond}$, and hence $\langle \rho_0 : P \rangle \to^\star \langle \rho : \mathit{Stat}^\alpha ; \mathit{Seq}' \rangle \to \langle \rho : \mathit{Seq}' \rangle$ and it immediately follows that $\rho \in \mathcal{I}_{cs}(\Psi^{\downarrow\alpha})$.

Sixth, consider a constraint of the form $\Psi^{\downarrow\alpha} \supseteq \Psi^{\downarrow\gamma}$, corresponding to a statement Stat^α of the form **if** cond **then** ${}^\beta\mathit{Seq}^\gamma$. Suppose that $\rho \in \mathcal{I}_{cs}(\Psi^{\downarrow\gamma})$. Hence $\langle \rho_0 : P \rangle \to^\star \langle \rho' : \mathit{Stat}^\gamma ; \mathit{Seq}' \rangle \to^\star_{\mathit{Seq}'} \langle \rho : \mathit{Seq}' \rangle$. Applying Proposition 3 to $\langle \rho_0 : P \rangle \to^\star \langle \rho' : \mathit{Stat}^\gamma ; \mathit{Seq}' \rangle$ proves that there exists an environment ρ'' such that

$$\langle \rho_0 : P \rangle \to^\star \langle \rho'' : \mathit{Stat}^\alpha ; \mathit{Seq}' \rangle \to^\star_{\mathit{Seq}'} \langle \rho' : \mathit{Stat}^\gamma ; \mathit{Seq}' \rangle.$$

It follows that $\langle \rho_0 : P \rangle \to^\star \langle \rho'' : \mathit{Stat}^\alpha ; \mathit{Seq}' \rangle \to^\star_{\mathit{Seq}'} \langle \rho : \mathit{Seq}' \rangle$, and this implies that $\rho \in \mathcal{I}_{cs}(\Psi^{\downarrow\alpha})$.

Seventh, consider a constraint of the form $\Psi^{\uparrow\alpha} \supseteq \Psi^{\downarrow\gamma}$ corresponding to a statement Stat^α of the form **while** cond **do** ${}^\beta\mathit{Seq}^\gamma$. Suppose that $\rho \in \mathcal{I}_{cs}(\Psi^{\downarrow\gamma})$. Then $\langle \rho_0 : P \rangle \to^\star \langle \rho' : \mathit{Stat}^\gamma ; \mathit{Seq}' \rangle \to^\star_{\mathit{Seq}'} \langle \rho : \mathit{Seq}' \rangle$. Applying Proposition 4 to $\langle \rho_0 : P \rangle \to^\star \langle \rho' : \mathit{Stat}^\gamma ; \mathit{Seq}' \rangle$ proves that there exists an environment ρ'' such that

$$\langle \rho_0 : P \rangle \to^\star \langle \rho'' : \mathit{Seq}' \rangle \to^\star \langle \rho' : \mathit{Stat}^\gamma ; \mathit{Seq}' \rangle,$$

and that the first statement of Seq' is Stat^α. Combining this with the fact that $\langle \rho' : \mathit{Stat}^\gamma ; \mathit{Seq}' \rangle \to^\star \langle \rho : \mathit{Seq}' \rangle$ proves that $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha})$.

Eighth, consider a constraint of the form $\Psi^{\uparrow\beta} \supseteq \Psi^{\uparrow\alpha}[\mathit{cond}]$, corresponding to a statement Stat^α of the form **while** cond **do** ${}^\beta\mathit{Seq}^\gamma$. Suppose that $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha}[\mathit{cond}])$. This implies that $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha})$, $\rho \rhd \mathit{cond}$ and $\rho \models \mathit{cond}$. Hence there is a derivation $\langle \rho_0 : P \rangle \to^\star \langle \rho : \mathit{Seq}' \rangle$ such that the first statement of Seq' is Stat^α. Combining this with $\langle \rho : \mathit{Seq}' \rangle \to \langle \rho : \mathit{Seq} ; \mathit{Seq}' \rangle$ proves that $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\beta})$.

Finally, consider a constraint of the form $\Psi^{\downarrow\alpha} \supseteq \Psi^{\uparrow\alpha}[\neg cond]$ corresponding to a statement $Stat^{\alpha}$ of the form **while** $cond$ **do** ${}^{\beta}Seq^{\gamma}$. Suppose that $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha}[\neg cond])$. This implies that $\rho \in \mathcal{I}_{cs}(\Psi^{\uparrow\alpha})$, $\rho \rhd \neg cond$ and $\rho \models \neg cond$. Hence there is a derivation $\langle \rho_0 : P\rangle \rightarrow^{\star} \langle \rho : Stat^{\alpha}; Seq'\rangle \rightarrow \langle \rho : Seq'\rangle$ and so $\rho \in \mathcal{I}_{cs}(\Psi^{\downarrow\alpha})$. In summary then, we have show that $\mathcal{I}_{cs}$, the collecting semantics of P, is a model of each of the constraints in $\mathcal{EC}_P$, the environment constraints of P. $\square$

We now complete the proof that $\mathcal{I}_{cs}$ corresponds to the collecting semantics by showing that any model of the environment constraints must contain $\mathcal{I}_{cs}$. We begin by showing an important correspondence between the sets of environments associated to points before and after program statements in any model of the environments constraints.

Proposition 6 *Let $\mathcal{I}$ be a model of the environment constraints for P. If $\langle \rho_0 : P\rangle \rightarrow^{n} \left\langle \rho : Seq_a; Stat^{\alpha}; Stat^{\beta}; Seq_b\right\rangle$ then $\mathcal{I}$ satisfies $\Psi^{\uparrow\beta} \supseteq \Psi^{\downarrow\alpha}$.*

Proof: The proof is by induction on n. If $n = 0$, then $\langle \rho_0 : P\rangle \rightarrow^{n} \left\langle \rho : Seq_a; Stat^{\alpha}; Stat^{\beta}; Seq_b\right\rangle$ implies that $Seq_a; Stat^{\alpha}; Stat^{\beta}; Seq_b$ is P, and so $Stat^{\alpha}$ and $Stat^{\beta}$ must be consecutive statements in P. Hence the environment constraints for P include the constraint $\Psi^{\uparrow\beta} \supseteq \Psi^{\downarrow\alpha}$.

Now, suppose that the proposition holds for n and consider the case of $n+1$. If $\langle \rho_0 : P\rangle \rightarrow^{n+1} \left\langle \rho : Seq_a; Stat^{\alpha}; Stat^{\beta}; Seq_b\right\rangle$ then there exists a state $\langle \rho_n : Seq_n\rangle$ such that

$$\langle \rho_0 : P\rangle \rightarrow^{n} \langle \rho_n : Seq_n\rangle \rightarrow \left\langle \rho : Seq_a; Stat^{\alpha}; Stat^{\beta}; Seq_b\right\rangle$$

Now, if Seq_n is of the form $Seq'_a; Stat^{\alpha}; Stat^{\beta}; Seq'_b$ then the proposition follows from immediately from the induction hypothesis. If Seq_n is not of this form, then the first statement of Seq_n must be an **if-then** or **while-do** statement whose condition is satisfied by ρ_n and whose body introduces one or both of the statements $Stat^{\alpha}$ and $Stat^{\beta}$. If $\mathit{first}(Seq_n)$ introduces both statements, then $Stat^{\alpha}$ and $Stat^{\beta}$ are again consecutive statements in P, and so $\Psi^{\uparrow\beta} \supseteq \Psi^{\downarrow\alpha}$ is a constraint in the environment constraints of P.

On the other hand, if $\mathit{first}(\mathit{Seq}_n)$ introduces only one of Stat^α and Stat^β, then it must be the case that Stat^α is the last statement in the body of $\mathit{first}(\mathit{Seq}_n)$. If $\mathit{first}(\mathit{Seq}_n)$ is an **if-then** statement, then $\langle \rho_0 : P\rangle \rightarrow^{n+1} \langle \rho_{n+1} : \mathit{Seq}_{n+1}\rangle$ implies that

$$\langle \rho_0 : P\rangle \rightarrow^n \left\langle \rho_n : \mathit{Stat}^\gamma; \mathit{Stat}^\beta; \mathit{Seq}\right\rangle \rightarrow \left\langle \rho_n : \mathit{Seq}'; \mathit{Stat}^\alpha; \mathit{Stat}^\beta; \mathit{Seq}\right\rangle$$

where the body of Stat^γ is $\mathit{Seq}'; \mathit{Stat}^\alpha$. Now, on applying the induction hypothesis to Stat^γ and Stat^β, we have that $\mathcal{I}$ satisfies $\Psi^{\uparrow\beta} \supseteq \Psi^{\downarrow\gamma}$. Moreover, the environment constraints corresponding to Stat^γ contain the constraint $\Psi^{\downarrow\gamma} \supseteq \Psi^{\downarrow\alpha}$. It follows that $\mathcal{I}$ must satisfy $\Psi^{\uparrow\beta} \supseteq \Psi^{\downarrow\alpha}$.

If $\mathit{first}(\mathit{Seq}_n)$ is a **while-do** statement, then $\langle \rho_0 : P\rangle \rightarrow^\star \langle \rho_{n+1} : \mathit{Seq}_{n+1}\rangle$ implies that

$$\langle \rho_0 : P\rangle \rightarrow^n \left\langle \rho_n : \mathit{Stat}^\beta; \mathit{Seq}\right\rangle \rightarrow \left\langle \rho_{n+1} : \mathit{Seq}'; \mathit{Stat}^\alpha; \mathit{Stat}^\beta; \mathit{Seq}\right\rangle$$

where the body of Stat^β is $\mathit{Seq}'; \mathit{Stat}^\alpha$. This implies that the environment constraints contain $\Psi^{\uparrow\beta} \supseteq \Psi^{\downarrow\alpha}$. $\square$

Lemma 2 *$\mathcal{I}_{cs}$ is smaller than any model of $\mathcal{EC}_P$.*

Proof: To prove the lemma, we need to show that if $\mathcal{I}$ is a model of $\mathcal{EC}_P$ then

$\mathcal{I}(\Psi^\mu) \supseteq \mathcal{I}_{cs}(\Psi^\mu)$ for all program points μ.

To prove this, it suffices to show the following two properties

(i) If $\langle \rho_0 : P\rangle \rightarrow^m \langle \rho : \mathit{Stat}^\alpha; \mathit{Seq}\rangle$ then $\rho \in \mathcal{I}(\Psi^{\uparrow\alpha})$.

(ii) If $\langle \rho_0 : P\rangle \rightarrow^{m-n} \langle \rho' : \mathit{Stat}^\alpha; \mathit{Seq}\rangle \rightarrow^n_{\mathit{Seq}} \langle \rho : \mathit{Seq}\rangle$ then $\rho \in \mathcal{I}(\Psi^{\downarrow\alpha})$.

where $m \geq 0$ and $1 \leq n \leq m$. These two properties shall be proved simultaneously by induction. The primary induction shall be m, with a secondary induction on n. In the base case of $m = 0$, (i) reduces to $\rho_0 \in \mathcal{I}(\Psi^{\uparrow 1})$, and this is trivially true since $\mathcal{I}$ satisfies $\Psi^{\uparrow 1} \supseteq \top$. On the other hand, (ii) is vacuously true since its preconditions cannot be met unless $m \geq 1$.

Now, suppose that for some m', (i) and (ii) hold for all $m < m'$, and we seek proofs of (i) and (ii) when $m = m'$. First consider (ii). The proof for this case employs a secondary induction on n. In the base case of $n = 1$, the assumptions of (ii) reduce to

$$\langle \rho_0 : P \rangle \rightarrow^{m'-1} \langle \rho' : \mathit{Stat}^\alpha ; \mathit{Seq} \rangle \rightarrow \langle \rho : \mathit{Seq} \rangle.$$

From part (i) of the induction hypothesis, $\rho' \in \mathcal{I}(\Psi^{\uparrow\alpha})$. Now, Stat^α must either an assignment statement, or an **if-then** or **while-do** statement whose condition is not satisfied by ρ'. If the first case, let the statement be $X := t$. This means that $\rho' \rhd t$ and ρ is $\rho'[X \mapsto v]$ where v is $\rho'(t)$. Now, $\mathcal{I}$ must satisfy the constraint $\Psi^{\downarrow\alpha} \supseteq \Psi^{\uparrow\alpha}[X \mapsto t]$, and hence combining this with $\rho' \in \mathcal{I}(\Psi^{\uparrow\alpha})$ proves that $\rho \in \mathcal{I}(\Psi^{\downarrow\alpha})$. On the other hand, if Stat^α is an **if-then** or **while-do** statement whose condition is not satisfied by ρ', then since $\mathcal{I}$ must satisfy $\Psi^{\downarrow\alpha} \supseteq \Psi^{\uparrow\alpha}[\neg \mathit{cond}]$, it is again immediate that $\rho = \rho' \in \mathcal{I}(\Psi^{\downarrow\alpha})$.

Now suppose that, for some $n' > 1$, (ii) holds when $m = m'$ and $n < n'$, and assume that

$$\langle \rho_0 : P \rangle \rightarrow^{m'-n'} \langle \rho' : \mathit{Stat}^\alpha ; \mathit{Seq} \rangle \rightarrow^{n'}_{\mathit{Seq}} \langle \rho : \mathit{Seq} \rangle.$$

Again, part (i) of the induction hypothesis implies that $\rho' \in \mathcal{I}(\Psi^{\uparrow\alpha})$. Now, since $n' > 1$, the statement Stat^α must be an **if-then** or **while-do** statement whose condition is satisfied by ρ'. Hence, it must either be the case that

$$\begin{array}{l} \langle \rho' : \mathit{Stat}^\alpha ; \mathit{Seq} \rangle \rightarrow \langle \rho' : \mathit{Seq}' ; \mathit{Seq} \rangle \rightarrow^{n'-1}_{\mathit{Seq}} \langle \rho : \mathit{Seq} \rangle \quad \text{or} \\ \langle \rho' : \mathit{Stat}^\alpha ; \mathit{Seq} \rangle \rightarrow \langle \rho' : \mathit{Seq}' ; \mathit{Stat}^\alpha ; \mathit{Seq} \rangle \rightarrow^{n'-1}_{\mathit{Seq}} \langle \rho : \mathit{Seq} \rangle \end{array}$$

where Seq' is the body of Stat^α. Consider these two possibilities in turn. In the first case, let Stat^β is the last statement in Seq. Proposition 2 implies that there exists an environment ρ'' and integers $j \geq 0$ and $k > 1$ such that $j + k = n' - 1$ and

$$\langle \rho' : \mathit{Seq}' ; \mathit{Seq} \rangle \rightarrow^{j} \left\langle \rho'' : \mathit{Stat}^\beta ; \mathit{Seq} \right\rangle \rightarrow^{k}_{\mathit{Seq}} \langle \rho : \mathit{Seq} \rangle .$$

This implies that

$$\langle \rho_0 : P \rangle \rightarrow^{m'-k} \left\langle \rho'' : \mathit{Stat}^\beta ; \mathit{Seq} \right\rangle \rightarrow^{k}_{\mathit{Seq}} \langle \rho : \mathit{Seq} \rangle.$$

Since $k < n'$, part (ii) of the induction hypothesis implies that $\rho \in \mathcal{I}(\Psi^{\downarrow\beta})$. Now, $\mathcal{I}$ satisfies the constraint $\Psi^{\downarrow\alpha} \supseteq \Psi^{\downarrow\beta}$, and this proves that $\rho \in \mathcal{I}(\Psi^{\downarrow\alpha})$.

On the other hand, if the derivation has the second form, then proposition 2 can again be applied, this time to show that there exists an environment ρ'' and integers $j \geq 0$ and $k > 1$ such that $j + k = n' - 1$ and

$$\langle \rho' : Seq' ; Stat^{\alpha} ; Seq \rangle \rightarrow^{j} \langle \rho'' : Stat^{\alpha} ; Seq \rangle \rightarrow^{k}_{Seq} \langle \rho : Seq \rangle$$

This implies that

$$\langle \rho_0 : P \rangle \rightarrow^{m'-k} \langle \rho'' : Stat^{\alpha} ; Seq \rangle \rightarrow^{k}_{Seq} \langle \rho : Seq \rangle .$$

Since $k < n'$, part (ii) of the induction hypothesis implies that $\rho \in \mathcal{I}(\Psi^{\downarrow\alpha})$.

It remains to prove the inductive case for (i). Assume that $\langle \rho_0 : P \rangle \rightarrow^{m'}$ $\langle \rho : Stat^{\alpha} ; Seq \rangle$. Since $m' \geq 1$, there exist ρ', $Stat^{\beta}$ and Seq' such that

$$\langle \rho_0 : P \rangle \rightarrow^{m'-1} \left\langle \rho' : Stat^{\beta} ; Seq' \right\rangle \rightarrow \langle \rho : Stat^{\alpha} ; Seq \rangle .$$

Now, consider the cases of $Stat^{\beta}$. If $Stat^{\beta}$ is an assignment statement or an **if-then** or **while-do** statement whose condition is not satisfied by ρ', then Seq' must be $Stat^{\alpha} ; Seq$, and

$$\langle \rho_0 : P \rangle \rightarrow^{m'-1} \left\langle \rho' : Stat^{\beta} ; Stat^{\alpha} ; Seq \right\rangle \rightarrow \langle \rho : Stat^{\alpha} ; Seq \rangle .$$

Now, (ii) has just been proved in the case where $m = m'$, and so $\rho \in \mathcal{I}(\Psi^{\downarrow\beta})$. Furthermore, Proposition 6 proves that $\mathcal{I}(\Psi^{\uparrow\alpha}) \supseteq \mathcal{I}(\Psi^{\downarrow\beta})$. Hence $\rho \in \mathcal{I}(\Psi^{\uparrow\alpha})$.

On the other hand, if $Stat^{\beta}$ is an **if-then** or **while-do** statement whose condition is satisfied by ρ', then $Stat^{\alpha}$ must be the first statement appearing in the body of $Stat^{\beta}$. Hence the environment constraints contain $\Psi^{\uparrow\alpha} \supseteq \Psi^{\uparrow\beta}$. Moreover, (i) can be applied to the derivation $\langle \rho_0 : P \rangle \rightarrow^{m'-1}$ $\left\langle \rho' : Stat^{\beta} ; Seq' \right\rangle$ to prove that $\rho \in \mathcal{I}(\Psi^{\uparrow\beta})$, and so $\rho = \rho' \in \mathcal{I}(\Psi^{\uparrow\alpha})$. This completes the induction argument for (i), and thus completes the proof of the lemma. $\square$

Theorem 2 *The collecting semantics $\mathcal{I}_{cs}$ of an imperative program P is the least model of the environment constraints for P.*

Proof: From Lemma 1, $\mathcal{I}_{cs}$ is a model of the environment constraints for P. From Lemma 2, $\mathcal{I}_{cs}$ is smaller than all other models of the environment constraints. It follows that $\mathcal{I}_{cs}$ is exactly the least model of the environment constraints. $\square$

Chapter 4

Logic Programs

This chapter presents the background definitions on operational and collecting semantics for logic programs. Three different semantics are considered: top-down execution using the PROLOG left-to-right atom selection strategy, top-down execution using a non-deterministic atom selection strategy (for modeling certain aspects of parallel execution), and bottom-up execution. Corresponding to each operational semantics, a collecting semantics is given. In essence, this involves defining an appropriate notion of program point for the operational semantics, and then collecting information about program executions for each program point. The core part of the chapter deals with constraint based formulations of the collecting semantics. These formulations are similar in nature to equational formulations of collecting semantics used in other works on logic program analysis. One difference is that we use constraints instead of equations, and this leads to a more general and flexible framework.

4.1 Logic Programs

We begin with some preliminary definitions about logic programs. Let Σ denote a set of function symbols, let Π denote the set of predicate symbols and let VAR be a denumerable set of program variables. It is assumed that Σ, Π and VAR are disjoint. Each function symbol $f \in \Sigma$ and each predicate symbol $p \in \Pi$ is assumed to have a unique arity. A function symbol of arity 0 is called a *constant*.

A *(logic program) term* is either a program variable from VAR, or of the form $f(t_1, \ldots, t_n)$ where $n \geq 0$, f is a function symbol from Σ with arity n, and each t_i is a term. An *atom* is of the form $p(t_1, \ldots, t_n)$ where $n \geq 0$, p is a predicate symbol from Π with arity n, and each t_i is a term. Atoms shall be denoted by A, B or C. A *rule* is of the form $A_0 \leftarrow A_1, \ldots, A_n$ where each A_i is an atom. The atom A_0 is called the *head* of the rule and the sequence $A_1, \ldots, A_n$ is called the *body* of the rule. Each A_i, $i \geq 1$ is called a *body atom*. In the case where $n = 0$, the rule is called a *fact*. Rules shall be denoted by R. A term, atom or rule is *ground* if it does not contain any program variables.

A *logic program* P is a finite set of rules. Each rule in a program P is labeled with a unique integer, called a *rule label*. Likewise each body atom in P is labeled with a unique integer called a *body atom label*. We again denote labels by α, β, γ (possibly subscripted). Writing $R^\alpha \in P$ indicates that the rule R appears in P with label α. We say that the rule in P with label α is $A_0 \leftarrow A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$ if the rule $R^\alpha \in P$ has head A_0, body $A_1, \ldots, A_n$ and body atom labels $\alpha_1, \ldots, \alpha_n$. Similarly we say that the body of R^α is $A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$ if the body of $R^\alpha \in P$ is $A_1, \ldots, A_n$ and the labels of these body atoms are $\alpha_1, \ldots, \alpha_n$ respectively. A^α is a *body atom in* P if α is a body atom label and A is the body atom that appears in P with label α. A^α is a *head atom in* P if α is a rule label and the head of the rule $R^\alpha \in P$ is A.

A *substitution* θ is a mapping from VAR into terms. Note that this definition is somewhat non-standard. In the literature, substitution θ is typically required to satisfy $\theta(X) = X$ for all but a finite number of variables. However, for our purposes it is convenient to drop this restriction. Substitutions shall be written in prefix notation. For example the result of applying θ to X shall be written as $\theta(X)$. A substitution θ can be extended to map from terms, (or atoms or rules) into terms (or atoms or rules, respectively) in the usual way:

$$\theta\Big(f(t_1,\ldots,t_n)\Big) \stackrel{\text{def}}{=} f(\theta(t_1),\ldots,\theta(t_n)).$$

A *renaming* is a substitution that is a bijection on VAR. If θ is a renaming, then θ^{-1} denotes the substitution that maps $\theta(X)$ into X for all variables X.

An *environment* (or *valuation*) ρ is a substitution such that, for each variable $X \in$ VAR, $\rho(X)$ is a ground term (or *value*). Although "valuation" is the more standard terminology in the context of logic programs, we use "environment" to maintain consistency with previous definitions. If ρ is an environment and exp is term, atom or rule, then $\rho(exp)$ is a *ground instance* of exp. If ρ is an environment and θ is a renaming substitution, then $\rho \circ \theta$ denotes the environment that maps X into $\rho(\theta(X))$ for all program variables X.

An *equation* is of the form $s = t$ where s and t are both terms or both atoms. An *equation conjunction* E is a finite collection of equations, and is written in the form $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$ (the empty conjunction is denoted by *true*). An environment ρ *satisfies* an equation $s = t$ if $\rho(s)$ and $\rho(t)$ are identical ground terms or atoms. An environment ρ satisfies an equation conjunction if it satisfies each equation in the conjunction. We write $\rho \models E$ to denote that ρ satisfies E.

Two atoms A and B are *unifiable* if $A = B$ is satisfiable. A and B are *compatible* if A and $\theta(B)$ are unifiable for some renaming substitution θ. For example, $p(X)$ and $p(f(X))$ are compatible but not unifiable. Compatibility corresponds to unifiability where renaming can be performed to avoid variable name clashes. Where $exp_1,\ldots,exp_n$, $n \geq 1$, is a sequence of terms, atoms, rules or equation conjunctions $var(exp_1,\ldots,exp_n)$ denotes the set of all program variables that appear in $exp_1,\ldots,exp_n$.

4.2 Operational Semantics

We now present three different operational semantics for logic programs – the first two are top-down semantics, and the last is a bottom-up semantics. The motivation for presenting more than one operational semantics for logic programs is twofold. First, by using a variety of operational models, we are better able to illustrate the process of constructing environment constraints (and, subsequently, set constraints) from a program. Second, it provides some evidence to support the wider claim that set based analysis is a general methodology for analyzing programs and is not tied to a particular notion of operational semantics. We begin with the top-down definition.

We begin by describing the two top-down semantics. First, define that a *goal* is of the form $\leftarrow A_1, \ldots, A_n$, $n \geq 1$, where each A_i is an atom. Now, given such a goal, the usual definition of logic program execution involves repeatedly reducing this goal using the rules of the program. Informally, this process can be described as follows: given a goal G_0, a sequence of goals $G_0, G_1, \ldots$ is defined such that each goal G_{i+1} in the sequence is obtained from its predecessor G_i by selecting an atom A from G_i and a (renamed) rule $B_0 \leftarrow B_1, \ldots, B_r$ from P, unifying A and B_0 to obtain a unifier θ, replacing A in G_i by $B_1, \ldots, B_r$, and then applying θ to the result. We shall instead adopt a CLP style formulation of program semantics [28]. This is done for two reasons. First, it simplifies certain aspects of the presentation, and second, it leads to a more general formulation that is directly applicable to other CLP languages. The main difference in the CLP approach is that to notion of unification is replaced by equations (and, more generally, constraints), and the notion of goal is generalized to include two components – an equation conjunction and a sequence of atoms. We now present the details.

An *atom selection function* is a function that maps any sequence of atoms $A_1, \ldots, A_n$ into an index i, $1 \leq i \leq n$. We say that A_i is *selected* from $A_1, \ldots, A_n$ and refer to A_i as the *selected atom*. A (top-down) *state* is of the form $\langle E : G \rangle$ where E is a satisfiable equation conjunction and G is a sequence of atoms (the empty sequence of atoms is denoted by *empty*). The top-down operational semantics is defined using a rewrite relation between states. Specifically, in the context of some atom selection function, there is a *derivation step* $\langle E : G \rangle \xrightarrow[i,\alpha,\theta]{} \langle E' : G' \rangle$ if

(i) G is $A_1, \ldots, A_m$ and the atom selection function maps G into i;

(ii) the rule with label α in P has of the form $B_0 \leftarrow B_1, \ldots, B_r$;

(iii) θ is a renaming substitution such that $var(\theta(R)) \cap var(E, G) = \{\}$;

(iv) G' is $A_1, \ldots, A_{i-1}, \theta(B_1), \ldots, \theta(B_r), A_{i+1}, \ldots, A_m$, and

(v) E' is $E \wedge (A_i = \theta(B_0))$.

Note that, by definition of states, the equation conjunction $E \wedge (A = \theta(B_0))$ must be satisfiable, and this implicitly requires that A and $\theta(B_0)$ be unifiable.

A *derivation* $\mathcal{D}$ is a sequence of derivations steps of the form

$$\langle E_0 : G_0 \rangle \xrightarrow[i_1, \alpha_1, \theta_1]{} \langle E_1 : G_1 \rangle \xrightarrow[i_2, \alpha_2, \theta_2]{} \cdots \xrightarrow[i_n, \alpha_n, \theta_n]{} \langle E_n : G_n \rangle.$$

We say that $\mathcal{D}$ is a derivation from $\langle E_0 : G_0 \rangle$ to $\langle E_n : G_n \rangle$. Note that some works on logic programming semantics define derivations to be maximal (finite or infinite) sequences of derivation steps. However, for our purposes it is more convenience to use finite derivations and to omit the requirement of maximality. In particular this means that any subsequence of the steps in a derivation is also a derivation.

The meaning of a program P is defined using the *successful* derivations, which are the derivations that end in a state of the form $\langle E : \mathit{empty} \rangle$. Specifically, a program P defines a function $[\![\cdot]\!]_P$ that maps goals $\leftarrow G$ into sets of equation conjunctions as follows:

$$[\![\leftarrow G]\!]_P \stackrel{\text{def}}{=} \{E : \text{there is a derivation from } \langle \mathit{true} : G \rangle \text{ to } \langle E : \mathit{empty} \rangle\}.$$

By varying the atom selection function in the above definitions, different operational semantics are obtained. If the selection function always selects the leftmost atom from a sequence, then the usual PROLOG style left-to-right semantics is obtained. To illustrate this selection function, consider the logic program and goal in Figure 4.1. Consider the following derivation (in which subscripts on derivation steps have been omitted for clarity) starting from the state $\langle \mathit{true} : \leftarrow p(X) \rangle$

$$\langle \mathit{true} : p(X) \rangle \rightarrow \langle E_1 : q(Y), r(Y) \rangle \rightarrow \langle E_2 : r(Y) \rangle \rightarrow \langle E_3 : \mathit{empty} \rangle$$

where E_1, E_2 and E_3 are given by

$$\begin{array}{ll} E_1 : & (p(X) = p(Y))\,. \\ E_2 : & (p(X) = p(Y) \wedge q(Y) = q(b))\,. \\ E_3 : & (p(X) = p(Y) \wedge q(Y) = q(b) \wedge r(Y) = r(b))\,. \end{array}$$

This derivation is a maximal, and is in fact the only maximal derivation for $\langle \mathit{true} : \leftarrow p(X) \rangle$, modulo variable renaming. Now, consider a selection

$$\begin{array}{l}
\leftarrow p(X). \\
p(Y) \leftarrow q(Y), r(Y). \\
q(b). \\
r(a). \\
r(b).
\end{array}$$

Figure 4.1: **Program 7**

function that chooses an atom non-deterministically. Using such a selection function, there are two additional maximal derivations from $\langle true : p(X) \rangle$:

$$\begin{array}{l}
\langle true : p(X) \rangle \rightarrow \langle E_1 : q(Y), r(Y) \rangle \rightarrow \langle E_2'' : q(Y) \rangle, \text{ and} \\
\langle true : p(X) \rangle \rightarrow \langle E_1 : q(Y), r(Y) \rangle \rightarrow \langle E_2' : q(Y) \rangle \rightarrow \langle E_3 : empty \rangle
\end{array}$$

where E_2' and E_2'' are given by

$$\begin{array}{ll}
E_2' : & (p(X) = p(Y) \wedge r(Y) = r(a)). \\
E_2'' : & (p(X) = p(Y) \wedge r(Y) = r(b)).
\end{array}$$

We refer to the semantics induced by this non-deterministic atom selection as the *interleaving semantics* since it allows arbitrary interleaving of the solving of atoms. Although this semantics is still sequential in nature, it does capture the essence of certain aspects of parallel execution since it makes no commitment to the order in which atoms are selected. In doing so, it provides a basis for illustrating how set based analysis might be used to analyze parallel logic programs, without having to deal with the specific details of a parallel logic programming language. The development of collecting semantics and environment constraints shall use these two top-down semantics, although it is possible to accommodate other atom selection functions.

We now present the third operational semantics for logic programs. A (bottom-up) *state* is of the form $\langle E : A \rangle$ where E is a satisfiable equation conjunction and A is an atom. Such a state is *bottom-up derivable* if, for some $n \geq 0$, there is a rule R of the form $A \leftarrow A_1, \ldots, A_n$ in P, renaming substitutions θ_1, …, θ_n, and bottom-up derivable states $\langle E_1 : B_1 \rangle, \ldots, \langle E_n : B_n \rangle$ such that

- $var(R)$, $var(\theta_1(E_1), \theta_1(B_1))$, …, $var(\theta_n(E_n), \theta_n(B_n))$ are all disjoint sets, and

- E is $(A_1 = \theta_1(B_1)) \wedge \theta_1(E_1) \wedge \cdots \wedge (A_n = \theta_n(B_n)) \wedge \theta_n(E_n)$.

Thus, for example, if P is the program consisting of $p(a)$ and $q(X) \leftarrow p(X)$ then $\langle \mathit{true} : p(a) \rangle$ and $\langle X = a : q(X) \rangle$ are both bottom-up derivable states.

As before, the meaning of a program P is defined as a mapping $[\![\cdot]\!]$ from goals into equation conjunctions. Specifically, each goal $\leftarrow A_1, \ldots, A_n$ is mapped into:

$$[\![\leftarrow A_1, \ldots, A_n]\!]_P \stackrel{\text{def}}{=} \left\{ \begin{array}{c} A_1 = \theta_1(B_1) \;\wedge\; \theta_1(E_1) \\ \wedge \;\cdots\; \wedge \\ A_n = \theta_n(B_n) \;\wedge\; \theta_n(E_n) \end{array} \quad : \quad \begin{array}{l} \text{the } \langle E_i : B_i \rangle \\ \text{are bottom-up} \\ \text{derivable} \end{array} \right\}$$

where the $\theta_1, \ldots, \theta_n$ are renaming substitutions such that the sets $\mathit{var}(R)$, $\mathit{var}\Big(\theta_1(E_1), \theta_1(B_1)\Big)$, $\ldots$, $\mathit{var}\Big(\theta_n(E_n), \theta_n(B_n)\Big)$ are all disjoint.

We now compare our definition of bottom-up semantics with the more usual definition of bottom-up semantics that is based on the T_{P} function. The T_{P} function maps from and into sets of ground atoms and can be defined as follows:

$$T_{\text{P}}(S) = \left\{ \rho(A_0) \;:\; \begin{array}{c} A_0 \leftarrow A_1, \ldots, A_n \text{ is a rule in } P \\ \text{and } \rho(A_i) \in S,\; i = 1..n \end{array} \right\}.$$

In essence, given a set S of "assumptions", $T_{\text{P}}(S)$ is the set of consequences that are derivable in one step using P. Hence T_{P} is often called the immediate consequence operator. Now, T_{P} is a continuous function, and hence it has a least fixed point, denoted $\mathit{lfp}(T_{\text{P}})$, which can be computed as the limit of the sequence $\{\}, T_p(\{\}), T_p(T_p(\{\})), \ldots$. One of the main results of the standard semantics of logic programs is that, given a program P, the set of successful ground atoms, the least fixed point of T_{P} and the least Herbrand model of P all coincide (see [7, 41]).

Our definition of bottom-up semantics is more operational in nature. It is structured in a way that emphasizes similarities with the top-down operational semantics. This includes using equations (as opposed to using ground atoms) and defining program semantics as a mapping from the program goals into equations (as opposed to defining program semantics as a set of ground atoms). A formal correspondence between our bottom-up semantics and the T_{P} semantics appears in the discussion following Lemma 6 on page 99. However these differences between the two are not fundamental in

$\leftarrow r(f(U))$.
$r(X) \leftarrow p(X), q(X)$.
$p(f(a))$.
$p(f(b))$.
$p(g(a))$.
$q(f(Z))$.
$q(g(a))$.

Figure 4.2: **Program 8**

nature.

Note that the definition of bottom-up derivable is closely related to the variation of the T_p function due to Jaffar and Lassez [28], which can be described as follows. Let $\mathcal{I}$ be a set of goals and defined $T_p(\mathcal{I})$ to be

$$\left\{ \left\langle \bigwedge_{1 \leq i \leq n} (\theta_i(E_i) \wedge A_i = \theta_i(B_i)) : A_0 \right\rangle : \begin{array}{r} A_0 \leftarrow A_1, \ldots, A_n \text{ is a rule in } P \\ \text{and } \langle E_i : B_i \rangle \in \mathcal{I},\ 1 \leq i \leq n \end{array} \right\}$$

such that each θ_i is renaming substitutions and the sets $var(A_0, \ldots, A_n)$, $var(\theta_1(E_1), \theta_1(B_1))$, $\ldots$, $var(\theta_n(E_n), \theta_n(B_n))$ are all disjoint. The least fixed point of this function is exactly the set of bottom-up derivable states.

4.3 Comparison of Operational Semantics

To illustrate the differences between the three definitions of operational semantics, consider the logic program and goal in Figure 4.2. The three definitions of semantics given in the previous section are all equivalent in the sense that the equation conjunctions collected for each goal are equivalent. For goal indicated $\leftarrow r(f(U))$, they all give the following set of equation conjunctions (after simplifying $r(f(U)) = r(X)$ into $f(U) = X$, etc.).

$$\left\{ \begin{array}{l} f(U) = X \wedge X = f(a) \wedge X = f(Z), \\ f(U) = X \wedge X = f(b) \wedge X = f(Z) \end{array} \right\}$$

However the definitions differ in how this set is obtained, and this has important consequences for the corresponding collecting semantics as well as for approximation of these semantics. For example, the following is a valid derivation in the top-down interleaving semantics

$$\begin{array}{lll} \langle true : r(f(U)) \rangle & \to & \langle f(U) = X : p(X), q(X) \rangle \\ & \to & \langle f(U) = X \wedge X = f(Z) : p(X) \rangle \\ & \to & \langle f(U) = X \wedge X = f(Z) \wedge X = f(a) : empty \rangle \end{array}$$

whereas it is not a valid derivation in the top-down left-to-right semantics. This means that properties of derivations relating to the equations encountered at points during program execution vary from semantics to semantics. Hence the notions of collecting semantics arising from the two top-down semantics shall differ significantly. There is even a greater distinction between the collecting semantics arising from the bottom-up semantics because the sets of program points used are different.

One useful way to compare all three semantics is to consider the equations collected for each goal and focus on the *order* in which the equations are collected. Specifically, consider the equation conjunction

$$f(U) = X \wedge X = f(a) \wedge X = f(Z)$$

which corresponds to the matching of $r(f(U))$ with $r(X)$, $p(X)$ with $p(f(a))$, and $q(X)$ with $q(f(Z))$. This equation conjunction can be viewed as the composition of the three basic equations $f(U) = X$, $X = f(Z)$ and $X = f(a)$; the difference between the three semantics is in the order in which these basic equations are combined. We illustrate this in the following table, where parentheses are used to indicate the order of the combination of basic equations.

top-down left-to-right	$\big(f(U) = X \wedge X = f(a)\big) \wedge X = f(Z)$
top-down interleaving	$\big(f(U) = X \wedge X = f(a)\big) \wedge X = f(Z)$ $\big(f(U) = X \wedge X = f(Z)\big) \wedge X = f(a)$
bottom-up	$f(U) = X \wedge \big(X = f(Z) \wedge X = f(a)\big)$

Note that although the resulting equation conjunctions are equivalent in each case, since $\wedge$ is associative and commutative, there are often important differences when $\wedge$ is replaced by some approximate notion of conjunction.

In particular, many works on program analysis can be understood by replacing the operation of $\wedge$ by some conservative approximation of $\wedge$, and this new operation is in many cases not associative or commutative, and hence the approximations of a program induced by the three different semantics often differ. We have, for simplicity, ignored the treatment of disjunction. However the basic observation about analysis of logic programs can be generalized as follows: exact operations are replaced by approximate operations, and since the algebraic properties of the underlying operations (such as associativity and commutative of conjunction and disjunction, and distributivity of conjunction and disjunction) rarely hold for their approximate counterparts, the approximations induced by the various semantics do not usually coincide.

4.4 Collecting Semantics

In program analysis, we are not primarily interested in the result of a computation, but rather in what happens during the computation. In the context of logic programs, what we desire is information about the equations that arise at various points during program execution. A collecting semantics formalizes this notion by explicitly collecting the set of equations encountered at each program point. In other words, the collecting semantics serves to make explicit information that is already implicit in the operational semantics.

For the two top-down semantics, we shall define collecting semantics in the context of an initial goal $\leftarrow G_0$. As for the imperative program case, this represents a choice. We could, for example, define the collecting semantics using a set of initial goals, or perhaps all goals. In the context of logic program analysis, the use of a restricted set of initial goals seems most appropriate, and we have used a single initial goal mainly for presentational simplicity. We note that it is straightforward to extend the set based analysis of logic programs to deal with a set of initial goals that is either finite or can be described using regular term grammars. In what follows, it shall often be convenient to treat the initial goal as part of the program, and treat the goal itself as a rule without a head. In particular, the atoms in the initial goal shall be referred to as body atoms.

We begin by discussing appropriate notions of program point for logic programs. Recall that each rule and body atom in a program has a unique label. We also assume that the initial goal (if any) is labeled with a rule label and that each atom of the initial goal is labeled with a body atom label. These labels shall be used to denote program points as follows. A rule label shall indicate the execution state just *after* the rule has finished executing, or in other words, just after every body atom has been solved. A body atom label indicates the state just *before* execution of the body atom, or in other words, just before the body atom is selected. In essence, this definition is just a formalization of the notion of program point employed in the introduction (Chapter 3). However, there is a small difference in the details. In particular, the use of textual markers Ⓐ, Ⓑ, etc., is somewhat inconvenient for dealing with programs in a uniform way, and so we have chosen to use labels attached to program atoms. As an example, where previously we may have written $p(f(X,Y)) \leftarrow$ Ⓐ, $q(X,Y)$, Ⓑ to indicate that Ⓐ denotes the point just before execution of $q(X,Y)$ and Ⓑ indicates the point just after the execution of the rule body, we now write the rule

2. $p(f(X,Y)) \leftarrow q(X,Y)^1$

where the atom label 1 indicates the point just before the execution of $q(X,Y)$, and the rule label 2 (which labels the entire rule) indicates the point just after the execution of the rule body.

We also observe that the notion of program point captured in the above formalization represents a choice among many possible approaches. It is possible to consider more elaborate notions of program point that take into account the context in which an atom is "called". For example, consider extending the notion of "program point" so that it includes an additional label that indicates an atom's "parent". The issue of choosing a notion of program point arises in any approach to program analysis, and is largely orthogonal to the details of set based analysis. We have chosen a notion of program point that is simple, intuitive and has proven to be useful.

Note that a notion of program point that is appropriate for one operational model may not be particularly appropriate for another operational model. For example, the formalization used here is appropriate for a top-down left-to-right operational model because the atoms in the body of a rule are selected in left-to-right order (and so the atoms can be thought of as successive statements or procedure calls). It is marginally less appropriate for the top-down interleaving model because in this case there is no notion of order between body atoms and the execution of body atoms can be interleaved. It is even less appropriate for the bottom-up operational model because in this model there is no notion of the state "before" a body atom is selected. (In fact the collecting semantics corresponding to the bottom-up semantics shall completely ignore the program points corresponding to body atom labels).

Before defining the collecting semantics, we need some preliminary definitions. First, we extend the operational semantics to take into account program labels. This involves using labeled atoms in states, and refining the definition of derivation step so that when the body of a rule is inserted into a goal, the labels on the body atoms are retained. Specifically, let the rule used in a derivation step be $A_0 \leftarrow A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$, and let the renaming used is θ, then the new atoms introduced by the derivation step are $\theta(A_1^{\alpha_1}), \ldots, \theta(A_n^{\alpha_n})$.

Now, consider a derivation $\mathcal{D}$ and suppose that $\mathcal{D}$ contains a step of the form

$$\langle E : G \rangle \xrightarrow[i,\,\alpha,\,\theta]{} \langle E' : G' \rangle.$$

Let G have the form $A_1, \ldots, A_m$ and let the rule in P with label α be $B_0 \leftarrow B_1^{\alpha_1}, \ldots, B_r^{\alpha_r}$. The goal G' contains the atoms $\theta(B_1^{\alpha_1}), \ldots \theta(B_r^{\alpha_r})$ that do not appear in G. These new atoms are said to be *introduced using* θ. We also define that each of the atoms $\theta(B_1), \ldots \theta(B_r)$ is called a *child* of the atom A_i selected from G. The transitive closure of the child relation is used to define an *descendant* relation. Specifically, an atom A^α is a *descendant* of an atom B^β if either A^α is a child of B^β or else A^α is a child of a descendant of B^β.

A derivation $\mathcal{D}$ *solves* an atom A^α if A^α is selected at some step in the derivation and all of the descendants of A^α are solved in subsequent steps of $\mathcal{D}$. A derivation $\mathcal{D}$ *minimally solves* an atom A^α if A^α is solved by $\mathcal{D}$ but is not solved by the derivation consisting of all but the last step of $\mathcal{D}$. In other words, A^α is minimally solved in $\mathcal{D}$ if (i) A^α is selected from the j^{th} state in $\mathcal{D}$, (ii) the last state in $\mathcal{D}$ does not contain any descendants of A^α, and (iii) all states between the j^{th} state and the last state contain a descendant of A^α. Intuitively, this occurs when the subproof of A^α (contained in $\mathcal{D}$) is completed by the last step of $\mathcal{D}$. Strictly speaking, the notions we have just defined require that atoms in a derivation to be labeled with auxiliary information such as the derivation step where they were introduced. Clearly this can be done, and we omit the details.

The collecting semantics is defined using the class of derivations whose first state is $\langle true : G_0\rangle$. Let $\mathcal{D}$ be such a derivation. $\mathcal{D}$ is said to *select* A^α *under* renaming θ if A^α is the atom selected from the last state in $\mathcal{D}$ and A^α is introduced using the renaming θ. In the case where the selected atom A^α is an atom from the starting state $\langle true : G_0\rangle$, $\mathcal{D}$ is said to *select* A^α under θ_{id} where θ_{id} is the identity renaming substitution. The derivation $\mathcal{D}$ *returns from* the rule R^α *under* renaming θ if some step of $\mathcal{D}$ is of the form $\langle E_a : G_a\rangle \xrightarrow[i,\alpha,\theta]{} \langle E_b : G_b\rangle$ and the atoms introduced by this step are solved in the subsequent steps of $\mathcal{D}$ and one of the introduced atoms is minimally solved in the subsequent steps of $\mathcal{D}$. In other words, the last descendant of the atoms introduced by $\langle E_a : G_a\rangle \xrightarrow[i,\alpha,\theta]{} \langle E_b : G_b\rangle$ is solved in the last step of $\mathcal{D}$. $\mathcal{D}$ *returns from* G_0^α *under* renaming θ_{id} if the last state of $\mathcal{D}$ has the form $\langle E : empty\rangle$ and α is the label of G_0.

We can now present the collecting semantics corresponding to the top-down and bottom-up operational models. In each case this consists of a mapping $\mathcal{CS}_P$ from (some subset of) the program points into sets of equation conjunction. We begin with the top-down collecting semantics. The following definition is parameterized by the atom selection function; it serves to define both the top-down left-to-right collecting semantics and the top-down interleaving semantics.

Definition 3 *Given an initial goal* $\leftarrow G_0$*, the* <u>*top-down collecting semantics*</u> *of a logic program* P *is the mapping* $\mathcal{CS}_P$ *such that*

$$\mathcal{CS}_P(\alpha) \stackrel{\text{def}}{=} \left\{\theta^{-1}(E) : \begin{array}{l} \textit{there is a derivation from } \langle true : G_0\rangle \textit{ to} \\ \langle E : G\rangle \textit{ that selects } A^\alpha \textit{ under } \theta \end{array}\right\}$$

$$\mathcal{CS}_P(\beta) \stackrel{\text{def}}{=} \left\{\theta^{-1}(E) : \begin{array}{l} \textit{there is a derivation from } \langle true : G_0\rangle \textit{ to} \\ \langle E : G\rangle \textit{ that returns from } R^\beta \textit{ under } \theta \end{array}\right\}$$

where α *ranges over all body atom labels and* β *ranges over all rule labels.* $\square$

Crucial to this definition is the specific formulation of "selects" and "returns from". In particular, the definition of "selects" refers only to the last state of the derivation. In other words, a derivation selects A^α under θ if A^α is the atom selected from the last state of the derivation. This means that the equation conjunction at the end of the derivation is the conjunction encountered just as A^α is about to start execution. Importantly, if an atom A^α is selected from some state other than the last state in a derivation $\mathcal{D}$, then by considering the sequence of derivation steps up to the point that A^α is selected, we can construct another definition $\mathcal{D}'$ such that A^α is selected from the last state in $\mathcal{D}'$. Similarly the definition of "returns from" refers to rule uses that are completed during the last step of the derivation. That is, a derivation returns from R^α under θ if the solving of the atoms introduced by the indicated use R^α is completed during the last step of the derivation. Again, this is done so that the equation conjunction at the end of the derivation is the conjunction encountered just after execution of the rule has been completed.

Definition 4 *The <u>bottom-up collecting semantics</u> of a logic program P is the mapping $\mathcal{CS}_P$ such that, for each rule R^α with body $A_1, \ldots, A_n$,*

$$\mathcal{CS}_P(\alpha) \;=\; \left\{ \begin{array}{c} \theta_1(E_1) \wedge (A_1 = \theta_1(B_1)) \\ \wedge \cdots \wedge \\ \theta_n(E_n) \wedge (A_n = \theta_n(B_n)) \end{array} \;:\; \begin{array}{c} \langle E_1 : B_1 \rangle, \ldots, \langle E_n : B_n \rangle \\ \textit{are bottom-up derivable} \end{array} \right\}$$

where $\theta_1, \ldots, \theta_n$ are such that $var(\theta(R))$, $var(E_1, B_1)$, $\ldots$, $var(E_n, B_n)$ are disjoint sets of variables. $\square$

To illustrate the differences between these different collecting semantics, consider the logic program in Figure 4.3. An immediate difference between the bottom-up and top-down collecting semantics is that the bottom-up collecting semantics does not say anything about program points 1 and 2. The two top-down semantics differ at points 1 and 2. Note that for point 6, each collecting semantics collects the singleton set consisting of the equation conjunction $W = f(X) \;\wedge\; W = f(Z)$ (strictly speaking, the top-down interleaved semantics also collects $W = f(Z) \;\wedge\; W = f(X)$, but since $\wedge$ is commutative we consider this equation conjunction to be identical to $W = f(X) \;\wedge\; W = f(Z)$).

3. $\leftarrow p(W)^1, \; q(W)^2.$
4. $p(f(X)).$
5. $p(g(Y)).$
6. $q(f(Z)).$

program point	Top-Down left-to-right	Top-Down interleaving	Bottom-Up
1	$\{true\}$	$\left\{ \begin{array}{c} true, \\ W = f(Z) \end{array} \right\}$	–
2	$\left\{ \begin{array}{c} W = f(X), \\ W = g(Y) \end{array} \right\}$	$\left\{ \begin{array}{c} true, \\ W = f(X), \\ W = g(Y) \end{array} \right\}$	–
6	$\left\{ \begin{array}{c} W = f(X) \\ \wedge\, W = f(Z) \end{array} \right\}$	$\left\{ \begin{array}{c} W = f(X) \\ \wedge\, W = f(Z) \end{array} \right\}$	$\left\{ \begin{array}{c} W = f(X) \\ \wedge\, W = f(Z) \end{array} \right\}$
4, 5, 6	$\{true\}$	$\{true\}$	$\{true\}$

Figure 4.3: **Program 9 and Its Collecting Semantics**

4.5 Environment Constraints

The main focus of this thesis is on analysis involving the possible values that each program variable may assume (this kind of analysis is often called typing analysis). However, set based analysis is by no means restricted to such analysis. For example, mode analysis and sharing analysis can also be performed using set based analysis techniques (see Chapter 9). The main reason for focusing on the values of program variables is that it enables a clearer presentation of the concepts of set based analysis. In particular, it enables program semantics to be characterized using environments instead of the more general notion of equations. In turn, this allows a fairly simple characterization of the collecting semantics using *environment constraints*.

The environment constraints used for logic programs are closely related to those used for imperative programs. As before, the operations in the environment constraints are intimately connected to the operators of the underlying semantics. However, since the operators involved in the semantics of logic programs differ from those for imperative semantics, we require some new environment constraint operations.

An *environment variable* is a variable that ranges over sets of environments, and shall be denoted by the symbol Ψ. For each program point α, there is a distinguished environment variable denoted Ψ^α, whose purpose is to describe the environments corresponding to point α. An *environment expression* is an expression of the form $[A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]$, where the A_i and B_i are atoms and the Ψ_i are environment variables. An *environment constraint* is of the form $\Psi \supseteq ee$ where Ψ is an environment variable and ee is an environment expression.

The meaning of environment constraints is defined in the context of an *interpretation* $\mathcal{I}$ that maps each environment variable into a set of environments. If ee is $[A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]$ then $\mathcal{I}(ee)$ is defined by

$$\mathcal{I}(ee) \stackrel{\text{def}}{=} \Big\{\rho : \rho(A_i) \in \{\rho'(B_i) : \rho' \in \mathcal{I}(\Psi)_i\},\ i = 1..n\Big\}.$$

To explain this definition, first note that each expression $B_i.\Psi_i$ essentially represents a collection of ground atoms under an interpretation $\mathcal{I}$, as follows

$$\mathcal{I}(B_i.\Psi_i) \ = \ \{\rho'(B_i) : \rho' \in \mathcal{I}(\Psi_i)\}.$$

Now, each element $A_i \in B_i.\Psi_i$ can be thought of as a condition on environments ρ that holds whenever $\rho(A_i)$ is contained in the set of ground atoms specified by $B_i.\Psi_i$. That is, $A_i \in B_i.\Psi_i$ is satisfied by an environment ρ under interpretation $\mathcal{I}$ if $\rho(A_i) \in \mathcal{I}(B_i.\Psi_i)$. Finally, the meaning of the entire expression is just the set of environments that satisfy all of these conditions:

$$\mathcal{I}([A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]) \;=\; \Big\{\rho : \rho(A_i) \in \mathcal{I}(B_i \Psi_i),\ i = 1..n\Big\}.$$

Note that if $n = 0$ then $\mathcal{I}([A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n])$ simply reduces to the set of all environments. Now, an interpretation $\mathcal{I}$ is a *model* of a constraint $\Psi \supseteq ee$ if $\mathcal{I}(\Psi) \supseteq \mathcal{I}(ee)$. An interpretation is a *model* of a collection of environment constraints if it is a model of each constraint in the collection.

We now present the environment constraints corresponding to each collecting semantics. In each case, P is a program, and we seek environment constraints $\mathcal{EC}_P$ such that the least model of $\mathcal{EC}_P$ coincides with $\mathcal{CS}_P$.

Definition 5 (Top-Down Left-to-Right Environment Constraints) *For each rule $R^\alpha \in P$ with body $A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$ and head A_0 (if it exists), $\mathcal{EC}_P$ contains the constraints*

$$\begin{array}{ccl}
\Psi^{\alpha_1} & \supseteq & \Big[A_0 \in B_0.\Psi^{\beta_0}\Big] \\
\Psi^{\alpha_2} & \supseteq & \Big[A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1},\Big] \\
\vdots & \vdots & \quad\vdots \qquad\qquad\quad \vdots \\
\Psi^{\alpha_n} & \supseteq & \Big[A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_{n-1} \in B_{n-1}.\Psi^{\beta_{n-1}}\Big] \\
\Psi^{\alpha} & \supseteq & \Big[A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_n \in B_n.\Psi^{\beta_n}\Big]
\end{array}$$

where β_0 ranges over body atom labels such that $B_0^{\beta_0}$ is a body atom in P and A_0 and B_0 are compatible, and the β_i, $i \geq 1$, range over rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible. □

If the rule R is a goal, then A_0 does not exist, and the entry $A_0 \in B_0.\Psi^{\beta_0}$ is simply deleted from each environment expression. For example, the constraint $\Psi^{\alpha_1} \supseteq [A_0 \in B_0.\Psi^{\beta_0}]$ becomes $\Psi^{\alpha_1} \supseteq [\,]$.

Intuitively, the top-down left-to-right constraints each have the form

$$\Psi^{\alpha_{j+1}} \supseteq \left[A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_j \in B_j.\Psi^{\beta_j}\right]$$

where the first entry $A_0 \in B_0.\Psi^{\beta_0}$ corresponds to the "calling" of the rule α via the body atom $B_0^{\beta_0}$, and the remaining entries $A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_j \in B_j.\Psi^{\beta_j}$ correspond to the "solving" of $A_1, \ldots, A_j$ via the rules $\beta_1, \ldots, \beta_j$. In other words, in the least model of the constraints, the environments encountered just before body atom A_{j+1} are those such that the rule is "called", and all of the atoms to the left of A_{j+1} have been solved. Figure 4.4 illustrates the construction of top-down environment constraints for Program 9, whereas Figure 4.5 gives the top-down constraints for a slightly more complex program involving recursion.

The environment constraints for the interleaved semantics are similar to those for the left-to-right. The main difference is that when considering a body atom A_i in a rule $A_0 \leftarrow A_1, \ldots, A_n$, the entries in the environment constraints corresponding to the solving of the body atoms to the left of A_i are omitted because the interleaved semantics specifies that goal atoms may be selected in any order.

Definition 6 (Top-Down Interleaving Environment Constraints) *For each rule R^α with body $A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$ and head A_0 (if it exists), $\mathcal{EC}_P$ contains the constraints:*

$$\begin{aligned}
\Psi^{\alpha_j} &\supseteq \left[A_0 \in B_0.\Psi^{\beta_0}\right], \quad j = 1..n \\
\Psi^{\alpha} &\supseteq \left[A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_n \in B_n.\Psi^{\beta_n}\right]
\end{aligned}$$

where β_0 ranges over body atom labels such that $B_0^{\beta_0}$ is a body atom in P and A_0 and B_0 are compatible, and the β_i, $i \geq 1$, range over rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible. □

Again, if R is a goal, then the entry $A_0 \in B_0.\Psi^{\beta_0}$ is simply deleted. Figure 4.6 illustrates the construction of the environment constraints for the interleaved semantics using Program 10. Note that in the case of Program 9 (Figure 4.4), the constraints for the interleaved semantics are the same as those for the top-down left-to-right semantics since the rules in Program 9 have less than two body atoms.

The environment constraints for the bottom-up semantics are essentially stripped down versions of the top-down constraints. In particular, all entries

3. $\leftarrow p(W)^1;\ q(W)^2.$
4. $p(f(X)).$
5. $p(g(Y)).$
6. $q(f(Z)).$

$$
\begin{aligned}
\Psi^1 &\supseteq [\,] \\
\Psi^2 &\supseteq [p(W) \in p(f(X)).\Psi^4] \\
\Psi^2 &\supseteq [p(W) \in p(g(Y)).\Psi^5] \\
\Psi^3 &\supseteq [p(W) \in p(f(X)).\Psi^4, q(W) \in q(f(Z)).\Psi^6] \\
\Psi^3 &\supseteq [p(W) \in p(g(Y)).\Psi^5, q(W) \in q(f(Z)).\Psi^6] \\
\Psi^4 &\supseteq [p(f(X)) \in p(W).\Psi^1] \\
\Psi^5 &\supseteq [p(g(Y)) \in p(W).\Psi^1] \\
\Psi^6 &\supseteq [q(f(Z)) \in q(W).\Psi^2]
\end{aligned}
$$

Figure 4.4: **Program 9 and Its Top-Down Left-To-Right Constraints**

2. $\leftarrow loop(a.b.nil, V)^1$
3. $loop(W.nil, W).$
5. $loop(X.L, Y) \leftarrow loop(L, Y)^4.$

$$
\begin{aligned}
\Psi^1 &\supseteq [\,] \\
\Psi^2 &\supseteq [loop(a.b.nil, V) \in loop(W.nil, W).\Psi^3] \\
\Psi^2 &\supseteq [loop(a.b.nil, V) \in loop(X.L, Y).\Psi^5] \\
\Psi^3 &\supseteq [loop(W.nil, W) \in loop(a.b.nil, V).\Psi^1] \\
\Psi^3 &\supseteq [loop(W.nil, W) \in loop(L, Y).\Psi^4] \\
\Psi^4 &\supseteq [loop(X.L, Y) \in loop(a.b.nil, V).\Psi^1] \\
\Psi^4 &\supseteq [loop(X.L, Y) \in loop(L, Y).\Psi^4] \\
\Psi^5 &\supseteq [loop(X.L, Y) \in loop(a.b.nil, V).\Psi^1, loop(L, Y) \in loop(W.nil, W).\Psi^3] \\
\Psi^5 &\supseteq [loop(X.L, Y) \in loop(a.b.nil, V).\Psi^1, loop(L, Y) \in loop(X.L, Y).\Psi^5] \\
\Psi^5 &\supseteq [loop(X.L, Y) \in loop(L, Y).\Psi^4, loop(L, Y) \in loop(W.nil, W).\Psi^3] \\
\Psi^5 &\supseteq [loop(X.L, Y) \in loop(L, Y).\Psi^4, loop(L, Y) \in loop(X.L, Y).\Psi^5]
\end{aligned}
$$

Figure 4.5: **Program 10 and Its Top-Down Left-To-Right Constraints**

3. $\leftarrow p(W)^1; \; q(W)^2.$
4. $p(f(X)).$
5. $p(g(Y)).$
6. $q(f(Z)).$

$$
\begin{aligned}
\Psi^1 &\supseteq [\,] \\
\Psi^2 &\supseteq [\,] \\
\Psi^3 &\supseteq [p(W) \in p(f(X)).\Psi^4, q(W) \in q(f(Z)).\Psi^6] \\
\Psi^3 &\supseteq [p(W) \in p(g(Y)).\Psi^5, q(W) \in q(f(Z)).\Psi^6] \\
\Psi^4 &\supseteq [p(f(X)) \in p(W).\Psi^1] \\
\Psi^5 &\supseteq [p(g(Y)) \in p(W).\Psi^1] \\
\Psi^6 &\supseteq [q(f(Z)) \in q(W).\Psi^2]
\end{aligned}
$$

Figure 4.6: **Program 9 and Its Top-Down Interleaved Constraints**

3. $\leftarrow p(W)^1; \; q(W)^2.$
4. $p(f(X)).$
5. $p(g(Y)).$
6. $q(f(Z)).$

$$
\begin{aligned}
\Psi^3 &\supseteq [p(W) \in p(f(X)).\Psi^4, q(W) \in q(f(Z)).\Psi^6] \\
\Psi^3 &\supseteq [p(W) \in p(g(Y)).\Psi^5, q(W) \in q(f(Z)).\Psi^6] \\
\Psi^4 &\supseteq [\,] \\
\Psi^5 &\supseteq [\,] \\
\Psi^6 &\supseteq [\,]
\end{aligned}
$$

Figure 4.7: **Program 9 and Its Bottom-Up Constraints**

dealing with the "calling" of the rule are deleted. This means that the only relevant program points are those corresponding to rules and goals.

Definition 7 (Bottom-Up Environment Constraints)
For each rule R^α with body $A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$, $\mathcal{EC}_P$ contains the constraints:

$$\Psi^\alpha \supseteq \left[A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_n \in B_n.\Psi^{\beta_n}\right]$$

where the β_i, $i \geq 1$, range over rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible. □

Figures 4.7 and 4.8 illustrate the construction of the bottom-up constraints for programs 9 and 10.

We now address the correctness of the environment constraints, which essentially states that the least model of a program's environment constraints

	$\Psi^2 \supseteq [loop(a.b.nil, V) \in loop(W.nil, W).\Psi^3]$
2. $\leftarrow loop(a.b.nil, V)^1$	$\Psi^2 \supseteq [loop(a.b.nil, V) \in loop(X.L, Y).\Psi^5]$
3. $loop(W.nil, W)$.	$\Psi^3 \supseteq [\,]$
5. $loop(X.L, Y) \leftarrow loop(L, Y)^4$.	$\Psi^5 \supseteq [loop(L, Y) \in loop(W.nil, W).\Psi^3]$
	$\Psi^5 \supseteq [loop(L, Y) \in loop(X.L, Y).\Psi^5]$

Figure 4.8: **Program 10 and Its Bottom-Up Constraints**

corresponds to the program's collecting semantics. To formalize this statement, first recall that environment constraints focus on the run-time values of program variables (as opposed to other run-time properties such as variable instantiation, aliasing or sharing between variables), and that these values are captured using sets of environments. In essence, the environment constraints of a program are written to capture minimal consistency conditions (with respect to the operational semantics at hand) between sets of environments associated with neighboring points in the program. The least model of these constraints defines the smallest (or most accurate) consistent assignment of environments to program points.

In contrast, the collecting semantics of a program defines a set of equation conjunctions for each program point. These equation conjunctions implicitly contain information about the possible values of program variables, in addition to other information. The information about variable values can be made explicit by considering the environments that satisfy the equation conjunctions. Specifically, for each program point α, the collecting semantics $\mathcal{CS}_P$ defines a set of environments

$$\{\rho : \rho \models E \text{ and } E \in \mathcal{CS}_P(\alpha)\}.$$

In essence, this set of environments is the same as the set of environments associated with α by the least model of the environment constraints. However, for technical reasons, the correspondence is not exact. Rather, the sets are equivalent only in the context of the program variables relevant to the point α. Specifically, define $var(\alpha)$, the set of variables relevant to point α, is defined as follows. If α is a rule label, then $var(\alpha) = var(R)$ where R is the rule in P with label α. If α is a body atom label, then $var(\alpha) = var(R)$ where R is the rule in P that contains the body atom with label α. Now, where var is a set of program variables and ρ and ρ' are environments, define that $\rho =_{var} \rho'$ if $\rho(X) = \rho(X)$ for each $X \in var$. Also, where S and S' are sets of environments, define that $S =_{var} S'$ if, for every environment $\rho \in S$, there is an environment $\rho' \in S'$ such that $\rho =_{var} \rho'$, and vice-versa. Finally, the correctness of the environment constraints can be stated as follows.

Theorem 3 (Environment Constraint Correctness)
The following correspondence holds under the top-down left-to-right semantics, the top-down interleaving semantics and the bottom-up semantics:

$$lm(\mathcal{EC}_P)(\Psi^\alpha) =_{var(\alpha)} \{\rho : \rho \models E \textit{ and } E \in \mathcal{CS}_P(\alpha)\}, \quad \textit{for all points } \alpha.$$

□

The proof of this theorem is contained in the next section. Again the proof is lengthy and tedious and is included mainly for the sake of completeness. Note that since environment constraints reason at the ground level (that is, they use environments as opposed to substitutions or constraints), an important component of this proof involves proving that reasoning at the ground level is adequate for the purposes of determining the possible run-time values of program variables. In contrast, reasoning at the ground level is not adequate for determining other run-time properties such as variable instantiation, aliasing or sharing.

4.6 Environment Constraint Correctness

We now prove the correctness of the environment constraints for the top-down left-to-right and interleaved semantics as well as for the bottom-up semantics (Theorem 3). This involves establishing an equivalence between a program's collecting semantics and its environment constraints. In essence, the definition of collecting semantics and the environment constraints differ in two respects. First, they deal with different objects – the collecting semantics deals with equation conjunctions and the environment constraints deal with environments. Second, the structures of the definitions are different – the collecting semantics is based on a rewrite relation, whereas the environment constraints use constraints that express local consistency conditions. The proof of correctness of the environment constraints is correspondingly in two parts. We begin by defining a *ground* collecting semantics that uses environments instead of equation conjunctions. The first part of the correctness proof then establishes a connection between the (original) collecting semantics and the ground collecting semantics, and the second step relates the ground collecting semantics with environment constraints. We begin with the top-down environment constraints.

4.6.1 Correctness of Top-Down Constraints

In essence, the states of the ground semantics are constructed from states that do not contain any program variables. Now, the equation conjunction of such a state is just a conjunction of equations between ground expressions, and these are simply *true* (since the equation conjunction in a state is required to be satisfiable), and hence can be omitted. Therefore, the states of the ground semantics are just sequences of ground atoms, that is, ground goals (we shall often abuse notation and omit the "$\leftarrow$" part of a goal). In the context of some atom selection function, we define that there is a *derivation step* $G \xrightarrow[i,\alpha,\rho]{} G'$, where G and G' are ground goals, if

(i) G is $A_1^{\alpha_1}, \ldots, A_m^{\alpha_m}$ and the atom selection function maps G into i;

(ii) the rule with label α in P is $B_0 \leftarrow B_1^{\beta_1}, \ldots, B_r^{\beta_r}$;

(iii) ρ is an environment such that $A_i = \rho(B_0)$, and

(iv) G' is $A_1^{\alpha_1}, \ldots, A_{i-1}^{\alpha_{i-1}}, \rho(B_1^{\beta_1}), \ldots, \rho(B_r^{\beta_r}), A_{i+1}^{\alpha_{i+1}}, \ldots, A_m^{\alpha_m}$;

A *ground derivation* is a sequence of derivations steps of the form

$$G_0 \xrightarrow[i_1,\alpha_1,\rho_1]{} G_1 \xrightarrow[i_2,\alpha_2,\rho_2]{} \cdots \xrightarrow[i_n,\alpha_n,\rho_n]{} G_n.$$

Let $\mathcal{D}$ be a ground derivation and suppose that $\mathcal{D}$ contains a ground derivation step of the form $G \xrightarrow[i,\alpha,\rho]{} G'$. Let G have the form $A_1, \ldots, A_m$ and let R be $B_0 \leftarrow B_1, \ldots, B_r$. Now, the ground goal G' contains the new atoms $\rho(B_1), \ldots \rho(B_r)$, which do not appear in G. Each new atom $\rho(B_j)$ is called a *child* of the atom A_i selected from G. An atom A^α is a *descendant* of an atom B^β if either A^α is a child of B^β or else A^α is a child of a descendant of B^β.

A ground derivation $\mathcal{D}$ *solves* an atom A^α if A^α is selected at some step in $\mathcal{D}$ and all of the descendants of A^α are solved in subsequent steps of $\mathcal{D}$. A ground derivation $\mathcal{D}$ *minimally solves* an atom A^α if A^α is solved by $\mathcal{D}$

but is not solved by the ground derivation consisting of all but the last step of $\mathcal{D}$.

The ground collecting semantics is defined using the class of ground derivations whose first goal is a ground instance of the initial goal G_0. Let $\mathcal{D}$ be such a derivation. $\mathcal{D}$ *introduces* A^α *at step* k *using* environment ρ if either (a) the k^{th} step of $\mathcal{D}$ is $G \xrightarrow[i,\alpha,\rho]{} G'$ and A^α is an atom that appears in G' but not in G, or else (b) $k = 0$, the first goal of $\mathcal{D}$ is $\rho(G_0)$ and A^α appears in $\rho(G_0)$. Similarly, $\mathcal{D}$ *uses* rule R^α *and* environment ρ *at* step k if either (a) the k^{th} step of $\mathcal{D}$ is $G \xrightarrow[i,\alpha,\rho]{} G'$ and R is the rule in P with label α, or else (b) $k = 0$, the first goal of $\mathcal{D}$ is $\rho(G_0)$ and the label of $\mathcal{G}_0$ is α. $\mathcal{D}$ is said to *select* A^α *under* renaming θ if A^α is the atom selected from G_n and A^α is introduced using ρ. The derivation $\mathcal{D}$ *returns from* the rule R^α *under* environment ρ if, for some $k \geq 0$, the atoms introduced by step k are solved in the subsequent steps of $\mathcal{D}$, one of the introduced atoms is minimally solved in the subsequent steps of $\mathcal{D}$ and $\mathcal{D}$ uses R^α (a rule or goal) and environment ρ at step k.

Now, in analogy to the collecting semantics CS defined using derivations, a *ground* collecting semantics can be defined using ground derivations.

Definition 8
Given an initial goal G_0, the top-down ground collecting semantics of a logic program P is the mapping GCS such that

$$\mathcal{GCS}_P(\alpha) \stackrel{\text{def}}{=} \left\{\rho : \begin{array}{l}\textit{there is a ground derivation from } G'_0 \textit{ that} \\ \textit{selects } A^\alpha \textit{ under } \rho\end{array}\right\},$$

$$\mathcal{GCS}_P(\beta) \stackrel{\text{def}}{=} \left\{\rho : \begin{array}{l}\textit{there is a ground derivation from } G'_0 \textit{ that} \\ \textit{returns from } R^\beta \textit{ under } \rho\end{array}\right\}$$

where α ranges over all body atom labels, β ranges over all rule labels and G'_0 ranges over ground instances of G_0. $\square$

We now prove the $\mathcal{CS}_P$ and $\mathcal{GCS}_P$ are essentially equivalent. We begin by proving two important connections between derivations and ground derivations. These connections shall only hold if the atom selection function satisfies the following property: the atom selection function maps $A_1, \ldots, A_n$ into i iff it maps $\theta(A_1), \ldots, \theta(A_n)$ into i, where θ is an arbitrary substitution. In other words, we require that the atom selection function selects atoms only on the basis of their place in the sequence and their predicate symbol. This is clearly the case for the selection function used in the top-down left-to-right semantics (which always selects the leftmost atom). In an intuitive sense, it also holds for the atom selection function used in the interleaved semantics, since in this case the atom selection function non-deterministically chooses one of the atoms from the goal. However, since the application of the atom selection function to a goal is not uniquely defined, the property is perhaps more accurately stated as: the atom selection function *may* map $A_1, \ldots, A_n$ into i iff it *may* map $\theta(A_1), \ldots, \theta(A_n)$ into i, where θ is an arbitrary substitution. Before stating the next proposition, we recall that the composition $\rho \circ \theta$ of an environment ρ with a renaming substitution θ yields the environment that maps each program variable X into $\rho(\theta(X))$.

Proposition 7 *Let D be a derivation of length n ending in state $\langle E : G \rangle$. If $\rho \models E$ then there exists a ground derivation $\mathcal{D}'$ such that, for $j = 1..n$, the j^{th} steps of $\mathcal{D}$ and $\mathcal{D}'$ are respectively*

$$\langle E_a : G_a \rangle \xrightarrow[i,\,\alpha,\,\theta]{} \langle E_b : G_b \rangle \quad \textit{and} \quad \rho(G_a) \xrightarrow[i,\,\alpha,\,\rho \circ \theta]{} \rho(G_b).$$

Proof: The proof is by induction on the length of $\mathcal{D}$. If $\mathcal{D}$ has length 0, then it consists of a single state, call it $\langle E : G \rangle$. If $\rho \models E$, then $\mathcal{D}'$ can be defined to be the single ground goal $\rho(G)$, and this completes the base case.

Now, suppose that the proposition holds for derivations of length n, and consider a derivation of the form

$$\langle E_0 : G_0 \rangle \xrightarrow[i_1,\,\alpha_1,\,\theta_1]{} \cdots \xrightarrow[i_n,\,\alpha_n,\,\theta_n]{} \langle E_n : G_n \rangle \xrightarrow[i,\,\alpha,\,\theta]{} \langle E : G \rangle.$$

Suppose that $\rho \models E$. This implies that $\rho \models E_n$, and by the induction hypothesis, it follows that there exists a ground derivation

$$\rho(G_0) \xrightarrow[i_1,\,\alpha_1,\,\rho_1]{} \cdots \xrightarrow[i_n,\,\alpha_n,\,\rho_n]{} \rho(G_n)$$

such that each ρ_i is $\rho \circ \theta_i$. It remains to show that there is a derivation step $\rho(G_n) \xrightarrow[i,\,\alpha,\,\rho \circ \theta]{} \rho(G)$. Let G_n be $A_1, \ldots, A_m$, and let the rule in P with label α be $B_0 \leftarrow B_1, \ldots, B_r$. Since $\langle E_n : G_n \rangle \xrightarrow[i,\,\alpha,\,\theta]{} \langle E : G \rangle$, we have

$$\begin{array}{rcl} G & = & A_1, \ldots, A_{i-1}, \theta(B_1), \ldots, \theta(B_r), A_{i+1}, \ldots, A_m \\ E & = & E_n \wedge (A_i = \theta(B_0)) \end{array}$$

Hence, the difference between $\rho(G_n)$ and $\rho(G)$ is that $\rho(A_i)$ in $\rho(G_n)$ is replaced by $\rho(\theta(B_1)), \ldots, \rho(\theta(B_n))$ in $\rho(G)$. Now, since $\rho \models E$, it follows that $\rho(A_i) = \rho(\theta(B_0))$. Hence $\rho(\theta(R))$ is equal to $\rho(A_i) \leftarrow \rho(\theta(B_1)), \ldots, \rho(\theta(B_n))$, and it follows that $\rho(G_n) \xrightarrow[i,\alpha,\rho\circ\theta]{} \rho(G)$. $\square$

Proposition 8 *Let D' be a ground derivation of length n starting from ground goal G'_0. If G_0 is a sequence of atoms such that G'_0 is a ground instance of G_0, then there exists a derivation $\mathcal{D}$ from $\langle true : G_0 \rangle$ to $\langle E_n : G_n \rangle$ and an environment $\rho_n \models E_n$ such that, for $j = 1..n$, the j^{th} steps of $\mathcal{D}$ and $\mathcal{D}'$ are respectively*

$$\langle E_a : G_a \rangle \xrightarrow[i,\alpha,\theta]{} \langle E_b : G_b \rangle \quad \textit{and} \quad \rho(G_a) \xrightarrow[i,\alpha,\rho]{} \rho(G_b)$$

such that $\rho =_{var(\alpha)} \rho_n \circ \theta$.

Proof: The proof is by induction on the length of $\mathcal{D}'$. If $\mathcal{D}'$ has length 0, then it consists of a single ground goal, call it G. If G_0 is a sequence of atoms such that G'_0 is a ground instance of G_0, then there exists an environment ρ such that G'_0 is $\rho(G_n)$. Clearly $\rho \models true$, and so assigning $\mathcal{D}$ to be derivation consisting of the single state $\langle true : G_0 \rangle$ completes the base case.

Now, suppose that the proposition holds for derivations of length n, and consider a ground derivation of the form

$$G'_0 \xrightarrow[i_1,\alpha_1,\rho_1]{} \cdots \xrightarrow[i_n,\alpha_n,\rho_n]{} G'_n \xrightarrow[i,\alpha,\rho]{} G'.$$

Suppose that G_0 is a sequence of atoms such that $\mathcal{G}'_0$ is $\rho(G_0)$. By the induction hypothesis, there exists an environment ρ' and a derivation

$$\langle E_0 : G_0 \rangle \xrightarrow[i_1,\alpha_1,\theta_1]{} \cdots \xrightarrow[i_n,\alpha_n,\theta_n]{} \langle E_n : G_n \rangle.$$

such that $\rho' \models E_n$, $G'_j = \rho'(G_j)$, and $\rho_j =_{var(\alpha_j)} \rho' \circ \theta_j$, where $j = 1..n$. It remains to show that there is an appropriate derivation step of the form $\langle E_n : G_n \rangle \xrightarrow[i,\alpha,\theta]{} \langle E : G \rangle$. Now, let G_n be $A_1, \ldots, A_m$, so that G'_n is $\rho'(A_1), \ldots, \rho'(A_m)$. Also let the rule in P with label α be $B_0 \leftarrow B_1, \ldots, B_r$. Since $G'_n \xrightarrow[i,\alpha,\rho]{} G'$,

$$G' = \rho'(A_1), \ldots, \rho'(A_{i-1}), \rho(B_1), \ldots, \rho(B_r), \rho'(A_{i+1}), \ldots, \rho'(A_m).$$

Now, let θ be a renaming substitution such that $var(\theta(R)) \cap var(E, G) = \{\}$, and define G and E as follows

$$\begin{array}{rcl} G & = & A_1, \ldots, A_{i-1}, \theta(B_1), \ldots, \theta(B_r), A_{i+1}, \ldots, A_m \\ E & = & E_n \wedge (A_i = \theta(B_0)). \end{array}$$

This defines the derivation $\mathcal{D}$, and it remains to show that there exists an appropriate environment ρ'' such that $\mathcal{D}$ and $\mathcal{D}'$ have the necessary relationship. Define ρ'' as follows:

$$\rho''(X) \;=\; \begin{cases} \rho'(X) & \text{if } X \in var(E_n, G_n) \\ \rho(\theta^{-1}(X)) & \text{otherwise.} \end{cases}$$

Since $var(E_0, G_0) \subseteq var(E_1, G_1) \subseteq \cdots \subseteq var(E_n, G_n)$, it follows that $\rho''(G_j) = \rho'(G_j)$, $j = 1..n$. Hence, $\rho''(G_j) = G'_j$, $j = 1..n$. Also, $\rho''(\theta(B_l)) = \rho(\theta^{-1}(\theta(B_l))) = \rho(B_l)$, $l = 1..r$, and so $\rho''(G)$ is

$$\rho'(A_1), \ldots, \rho'(A_{i-1}), \rho(B_1), \ldots, \rho(B_r), \rho'(A_{i+1}), \ldots, \rho'(A_m)$$

and hence $\rho''(G) = G'$. In summary, the j^{th} steps of $\mathcal{D}$ and $\mathcal{D}'$ have the respective forms

$$\langle E_a : G_a \rangle \xrightarrow[i_j, \alpha_j, \theta_j]{} \langle E_b : G_b \rangle \quad \text{and} \quad \rho''(G_b) \xrightarrow[i_j, \alpha_j, \rho_j]{} \rho''(G_b)$$

and so it only remains to show that $\rho_j =_{var(\alpha_j)} \rho'' \circ \theta_j$, $1 \leq j \leq n+1$. Now, for $j \leq n$, $X \in var(\alpha_j)$ implies that $\theta_j(X) \in var(E_n, G_n)$, and so $\rho'' \circ \theta_j(X) = \rho''(\theta_j(X)) = \rho'(\theta_j(X)) = \rho_j(X)$ (this last step follows from $\rho_j =_{var(\alpha_j)} \rho' \circ \theta_j$). Finally, in the case where $j = n+1$, $\rho_j =_{var(\alpha_j)} \rho'' \circ \theta_j$ reduces to $\rho =_{var(\alpha)} \rho'' \circ \theta$, which follows immediately from the definition of ρ''. $\square$

Using these propositions, the following correspondence between $\mathcal{CS}_P$ and $\mathcal{GCS}_P$ can be established.

Lemma 3 *Let $\mathcal{CS}_P$ and $\mathcal{GCS}_P$ be the collecting semantics and ground collecting semantics for an initial goal G_0. Then, for each program point α,*

$$\mathcal{GCS}_P(\alpha) =_{var(\alpha)} \{\rho : \rho \models E \textit{ for some } E \in \mathcal{CS}_P(\alpha)\} \qquad (4.3)$$

Proof: The proof proceeds in two parts, according to whether α is a rule label or body label. First consider the case where α is rule label, and let R be the rule in P with label α. If $\rho \in \mathcal{GCS}_P(\alpha)$ then there exists a ground instance G'_0 of the initial goal G_0, and a ground derivation $\mathcal{D}'$ from G'_0 that returns from R^α under ρ. Let n be the length of $\mathcal{D}$. Clearly there must be some k, $k \leq n$ such that the k^{th} step of $\mathcal{D}'$ is of the form $G'_a \xrightarrow[i,\alpha,\rho]{} G'_b$ such that each atom introduced during this step is solved in $\mathcal{D}'$ and one of the atoms introduced is minimally solved. Now, Proposition 8 implies that there exists a derivation $\mathcal{D}$ from $\langle true : G_0\rangle$ to $\langle E_n : G_n\rangle$ and an environment ρ' such that the k^{th} step of $\mathcal{D}$ is $\langle E_a : G_a\rangle \xrightarrow[i,\alpha,\theta]{} \langle E_b : G_b\rangle$, $\rho' \circ \theta =_{var(\alpha)} \rho$, and $\mathcal{D}$ solves the atoms introduced by the k^{th} step, and minimally solves one of them. Hence $\mathcal{D}$ returns from R^α under θ and so $\theta^{-1}(E_n) \in \mathcal{CS}_P(\alpha)$. Now, $\rho' \models E_n$, and so $\rho' \circ \theta \models \theta^{-1}(E)$. It follows that $\rho' \circ \theta$ is an element of the set on the right hand side of (4.3).

Conversely, if $\rho \models E$ for some $E \in \mathcal{CS}_P(\alpha)$ then there exists a derivation $\mathcal{D}$ from $\langle true : G_0\rangle$ to $\langle E_n : G_n\rangle$ that returns from R^α under θ and E is $\theta^{-1}(E_n)$. From $\rho \models E$ and the fact that E is $\theta^{-1}(E_n)$, it follows that $\rho \circ \theta^{-1} \models E_n$. Again, let n be the length of $\mathcal{D}$ and let k be such that the k^{th} step of $\mathcal{D}'$ is of the form $\langle E_a : G_a\rangle \xrightarrow[i,\alpha,\theta]{} \langle E_b : G_b\rangle$ and each body atom introduced during this step is solved in $\mathcal{D}'$, and one of them is minimally solved. Now, proposition 7 implies that there exists a derivation $\mathcal{D}'$ from $\rho(G_0)$ to $\rho(G_n)$ such that the k^{th} step of $\mathcal{D}$ is $\rho(G_a) \xrightarrow[i,\alpha,\rho']{} \rho(G_b)$ where $\rho' =_{var(\alpha)} \rho \circ \theta^{-1} \circ \theta$ and $\mathcal{D}'$ solves the atoms introduced by this step and minimally solves one of them. Hence $\mathcal{D}'$ returns from R^α under ρ and so $\rho \in \mathcal{CS}_P(\alpha)$.

The second part of the proof deals with the case where α is the label of a body atom in P; the proof here closely parallels that for the first part. If $\rho \in \mathcal{GCS}_P(\alpha)$ then there exists a ground instance G'_0 of the initial goal G_0, and a ground derivation $\mathcal{D}'$ from G'_0 to G'_n that selects A^α under ρ. That is, the atom selection function maps G'_n into i and the i^{th} element of G'_n is A^α such that A^α is introduced using ρ. Now, Proposition 8 implies that there exists a derivation $\mathcal{D}$ from $\langle true : G_0\rangle$ to $\langle E_n : G_n\rangle$ and an environment ρ' such that the i^{th} element of G_n is of the form B^α and is introduced using renaming θ where $\rho' \circ \theta =_{var(\alpha)} \rho$. By the assumption that substitutions do not affect the operation of the atom selection function, the i^{th} atom of G_n is selected. This implies that $\mathcal{D}$ selects B^α under θ and so $\theta^{-1}(E_n) \in \mathcal{CS}_P(\alpha)$. Since $\rho' \models E_n$, it follows that $\rho' \circ \theta \models \theta^{-1}(E_n)$. Hence $\rho' \circ \theta$ is an element of the set on the right hand side of (4.3).

Conversely, suppose that $\rho \models E$ for some $E \in \mathcal{CS}_P(\alpha)$. Then there exists a derivation $\mathcal{D}$ from $\langle E_0 : G_0\rangle$ to $\langle E_n : G_n\rangle$ that selects an atom of the form A^α under θ such that E is $\theta^{-1}(E_n)$. That is, the atom selection function maps G_n into i and the i^{th} element of G_n is A^α and this atom is introduced using θ. Since $\rho \models \theta^{-1}(E_n)$ it follows that $\rho \circ \theta^{-1} \models E_n$. Proposition 7 implies that there exists a derivation $\mathcal{D}'$ from $\rho \circ \theta^{-1}(G_0)$ to $\rho \circ \theta^{-1}(G_n)$ such that the i^{th} element of $\rho(G_n)$ is introduced using $\rho \circ \theta^{-1} \circ \theta$. By the assumption on the atom selection function, the i^{th} atom of $\rho(G_n)$ is selected. This implies that $\mathcal{D}$ selects $\rho(A^\alpha)$ under ρ and so $\rho \in \mathcal{CS}_P(\alpha)$. $\Box$

To complete the correctness proof of the environment constraints, we shall now relate $\mathcal{GCS}_P$ with the least model of the environment constraints. Specifically, where $\mathcal{I}_{gcs}$ denotes the interpretation that maps each Ψ^α into $\mathcal{GCS}_P(\alpha)$, we show that $\mathcal{I}_{gcs} = lm(\mathcal{EC}_P)$. The proof of this consists of two parts. The first part proves that $\mathcal{I}_{gcs}$ is a model of $\mathcal{EC}_P$. The second part proves that, for any model $\mathcal{I}$ of $\mathcal{EC}_P$, $\mathcal{I}_{gcs} \subseteq \mathcal{I}$.

We begin by proving the following proposition on combining parts of derivations. Note that this proposition only holds for atom selection rules satisfying the following criteria: if an atom A is selected from a ground goal G, then there exists a ground derivation starting from G that solves A before selecting any of the other atoms in G. In general, this condition may not be satisfied, and this will mean that the lemma will not hold. In this case, environment constraints can be used to obtain a conservative approximation of the collecting semantics, but cannot to be used to characterize it exactly. However in the case of the left-to-right and interleaving selection functions, the criteria is satisfied, and the environment constraints correspond exactly to the collecting semantics.

Proposition 9 *If $\mathcal{D}$ is a ground derivation from G to G' such that A is selected from G' and $\mathcal{D}'$ is a ground derivation that solves A, then there exists a ground derivation $\mathcal{D}''$ from G to G'' that minimally solves A where G'' is the result of deleting A from G'.*

Proof (for left-to-right semantics): Let G' be $A_1, A_2, \ldots, A_n$. The proof uses the ground derivation $\mathcal{D}'$ to construct a ground derivation $\mathcal{D}''$ from G to $A_2, \ldots, A_n$. In particular, we shall construct a derivation $\mathcal{D}''$ of the form

$$G_0 \xrightarrow[i_1, \alpha_1, \theta_1]{} G_1 \xrightarrow[i_2, \alpha_2, \theta_2]{} G_2 \xrightarrow[i_3, \alpha_3, \theta_3]{} \cdots$$

where each goal G_i in this derivation has the form $Seq_i, A_2, \ldots, A_n$ such that Seq_i contains exactly the set of descendants of A_1 that appear in G_i. That is, (a) if A_1 or any of its descendants appear in G_i, then they appear in Seq_i, and

(b) any atom in Seq_i is either A_1 or a descendant of A_1. The construction of $\mathcal{D}''$ proceeds as follows. The initial ground goal G_0 is just G and it is clear that if Seq_0 is set to be A_1 then conditions (a) and (b) are satisfied. Now, suppose that G_{i-1} can be written as $Seq_i, A_2, \ldots, A_n$ such that (a) and (b) are satisfied, and consider defining the i^{th} step of $\mathcal{D}''$. If Seq_{i-1} is empty then the construction of $\mathcal{D}''$ is complete. Otherwise, let Seq_{i-1} be of the form B, Seq'_{i-1}; that is, B is the first atom in the sequence Seq_{i-1} and Seq'_{i-1} contains all but the first atom in Seq_{i-1}. Since the selection function at hand is the leftmost selection function, B is selected from G_{i-1}. By part (b) of the invariant, B is either A_1 or a descendant of A_1. Now, since A_1 is solved in $\mathcal{D}'$, it must be the case that A_1 and all of its descendants are selected at some step in $\mathcal{D}'$. Hence $\mathcal{D}'$ must contain a step of the form $G_a \xrightarrow[1,\alpha,\rho]{} G_b$ such that the rule with label α has the form $C_0 \leftarrow C_1, \ldots, C_r$ where $\rho(C_0) = B$. Now, define the i^{th} of $\mathcal{D}''$ to be $G_{i-1} \xrightarrow[1,\alpha,\rho]{} G_i$ where G_i is $\rho(C_1), \ldots, \rho(C_r), Seq'_{i-1}, A_2, \ldots, A_n$. Clearly (a) and (b) are satisfied when Seq_i is $\rho(C_1), \ldots, \rho(C_r), Seq'_{i-1}$, and this completes the description of the procedure to construct $\mathcal{D}''$.

Eventually this procedure must reach a point such that G_i does not contain A_1 or any of the descendants of A_1. This is because the number of steps in the construction of $\mathcal{D}''$ is bounded by the number of steps in $\mathcal{D}'$. Hence eventually Seq_i is *empty*, and this yields a derivation $G \rightarrow^\star A_2, \ldots, A_n$, which is just G with the selected atom A_1 deleted. $\Box$

Proof (for interleaving semantics): Again let G' be $A_1, A_2, \ldots, A_n$. The main difference between this proof and the given above for the left-to-right semantics is that any of the atoms A_i may be selected. Suppose that the atom selection function selects A_i from G'. Then, we need to construct a ground derivation $\mathcal{D}''$ from G to $A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n$. Again, this construction is guided by $\mathcal{D}'$. The only difference is that this time the constructed ground derivation $\mathcal{D}''$ consists of ground goals of the form $A_1, \ldots, A_{i-1}, Seq_i, A_{i+1}, \ldots, A_n$. The construction process is a straightforward adaptation of that for the left-to-right case. $\Box$

Lemma 4 *$\mathcal{I}_{gcs}$ is a model of $\mathcal{EC}_P$.*

Proof (for left-to-right semantics): Let G_0 be the initial goal, and consider a constraint in $\mathcal{EC}_P$. Now, this constraint could either correspond to (a) a rule in P or (b) the initial goal G_0. We consider these cases in turn.

Suppose that the constraint corresponds to a rule R with label α_{n+1}, and let R be of the form $A_0 \leftarrow A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$. Then the constraint must be of the form

$$\Psi^{\alpha_{j+1}} \supseteq \Big[A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_j \in B_j.\Psi^{\beta_j}\Big] \tag{4.4}$$

where $0 \leq j \leq n$, β_0 is a body atom label such that $B_0^{\beta_0}$ is a body atom in P and A_0 and B_0 are compatible, and the β_i, $i = 1..j$, are rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible. Now, suppose that ρ is an element of $\mathcal{I}_{gcs}\Big([A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_j \in B_j.\Psi^{\beta_j}]\Big)$. This means that (a) $\rho(A_0) \in \mathcal{I}_{gcs}(B_0.\Psi^{\beta_0})$ and (b) $\rho(A_i) \in \mathcal{I}_{gcs}(B_i.\Psi^{\beta_i})$, $i = 1..j$.

From (a), there exists a $\rho' \in \mathcal{I}_{gcs}(\Psi^{\beta_0})$ such that $\rho(A_0) = \rho'(B_0)$. By definition of $\rho' \in \mathcal{I}_{gcs}(\Psi^{\beta_0})$, there exists a ground instance G_0' of G_0 and a ground derivation $\mathcal{D}$ from G_0' to G such that G is of the form $\rho'(B_0), C_1, \ldots, C_n$. Now, let $\mathcal{D}'$ be the ground derivation that combines $\mathcal{D}$ with the following addition ground derivation step

$$\rho'(B_0), A_1, \ldots, A_m \xrightarrow[1,\alpha_n,\rho]{} \rho(A_1), \ldots, \rho(A_n), C_1, \ldots, C_m.$$

Clearly $\mathcal{D}$ is a ground derivation from G_0' to $\rho(A_1), \ldots, \rho(A_n), C_1, \ldots, C_m$.

From (b), there exist $\rho_i \in \mathcal{I}_{gcs}(\Psi^{\beta_i})$ such that $\rho(A_i) = \rho_i(B_i)$, $i = 1..j$. By definition of $\rho_i \in \mathcal{I}_{gcs}(\Psi^{\beta_i})$, each ρ_i is such that there is a ground derivation $\mathcal{D}_i$ from some ground instance of G_0 such that $\mathcal{D}_i$ minimally solves $\rho_i(B_i)$.

In summary, there exists a ground derivation $\mathcal{D}'$ from G_0' (a ground instance of the initial goal G_0) to $\rho(A_1), \ldots, \rho(A_n), C_1, \ldots, C_m$, and, for each i, there exist derivations $\mathcal{D}_i$ that minimally solve each $\rho(A_i)$. Now, Proposition 9 can be applied to combine $\mathcal{D}'$ with $\mathcal{D}_1$ to produce a ground derivation $\mathcal{D}_1'$ from $\mathcal{G}_0$ to $\rho(A_2), \ldots, \rho(A_n), C_1, \ldots, C_m$. Proposition 9 can again be applied, this time to $\mathcal{D}_1'$ and $\mathcal{D}_2$ to produce a ground derivation from $\mathcal{G}_0$ to $\rho(A_3), \ldots, \rho(A_n), C_1, \ldots, C_m$. Repeating this process proves that there is a ground derivation $\mathcal{D}_j'$ from G_0' to $\rho(A_{j+1}), \ldots, \rho(A_n), C_1, \ldots, C_m$. Now, if $j < n$ then $\mathcal{D}_j'$ selects $\rho(A_{j+1})$ under ρ and so $\rho \in \mathcal{I}_{gcs}(\Psi^{\alpha_{j+1}})$. On the other hand, if $j = n$, then $\mathcal{D}_j'$ returns from $R^{\alpha_{n+1}}$ under ρ and so $\rho \in \mathcal{I}_{gcs}(\alpha_{n+1})$. Hence in either case the constraint is satisfied.

Now suppose that the constraint corresponds to the initial goal $\mathcal{G}_0$. The argument that such a constraint is satisfied by $\mathcal{I}_{gcs}$ is essentially a repeat of the argument for constraints corresponding to rules. Let G_0 be of the form $\leftarrow A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$, and let the label of this goal be α_{n+1}. Then the constraint must be of the form

$$\Psi^{\alpha_{j+1}} \supseteq \Big[A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_j \in B_j.\Psi^{\beta_j}\Big].$$

where $1 \leq j \leq n$, the β_i, $i = 1..j$, are rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible. Now, suppose that ρ is an element of $\mathcal{I}_{gcs}\Big([A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_j \in B_j.\Psi^{\beta_j}]\Big)$. This means that $\rho(A_i) \in \mathcal{I}_{gcs}(B_i.\Psi^{\beta_i})$, $i = 1..j$. Hence, there exist $\rho_i \in \mathcal{I}_{gcs}(\Psi^{\beta_i})$ such that $\rho(A_i) = \rho_i(B_i)$, $i = 1..j$. By definition of $\rho_i \in \mathcal{I}_{gcs}(\Psi^{\beta_i})$, each ρ_i is such that there is a ground derivation $\mathcal{D}_i$ from some ground instance of G_0 such that $\mathcal{D}_i$ minimally solves $\rho_i(B_i)$. Moreover, $\rho(A_1), \ldots, \rho(A_n)$ is an instance of G_0, and hence there is a derivation (of length 0) from an instance of G_0 to $\rho(A_1), \ldots, \rho(A_n)$.

In summary, there exists a ground derivation $\mathcal{D}'$ from some instance of G_0 to $\rho(A_1), \ldots, \rho(A_n)$, and, for $i = 1..j$, there exists a derivation $\mathcal{D}_i$ that minimally solves $\rho(A_i)$. Again, proposition 9 can be repeatedly applied to show that there is a ground derivation $\mathcal{D}'_j$ from G'_0 to $\rho(A_{j+1}), \ldots, \rho(A_n)$. Now, if $j < n$ then $\mathcal{D}'_j$ selects $\rho(A_{j+1})$ under ρ and so $\rho \in \mathcal{I}_{gcs}(\Psi^{\alpha_{j+1}})$. On the other hand, if $j = n$, then $\mathcal{D}'_j$ minimally solves G_0 under ρ and so $\rho \in \mathcal{I}_{gcs}(\alpha_{n+1})$. Hence in either case the constraint is satisfied. $\Box$

Proof (for interleaving semantics): The proof for interleaving semantics closely follows the structure of the proof for left-to-right semantics. Let G_0 be the initial goal, and consider a constraint in $\mathcal{EC}_P$. Now, this constraint could either correspond to (a) a rule in P or (b) the initial goal G_0. We consider these cases in turn. Suppose that the constraint corresponds to a rule R with label α_{n+1}, and let R be of the form $A_0 \leftarrow A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$. Then the constraint must have one of the following two forms

$$\begin{array}{lcl} \Psi^{\alpha_j} & \supseteq & \Big[A_0 \in B_0.\Psi^{\beta_0}\Big] \\ \Psi^{\alpha_{n+1}} & \supseteq & \Big[A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_n \in B_n.\Psi^{\beta_n}\Big] \end{array}$$

where $1 \leq j \leq n$, β_0 is a body atom label such that $B_0^{\beta_0}$ is a body atom in P and A_0 and B_0 are compatible, and the β_i, $i = 1..n$, are rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible.

Consider a constraint of the first form, and suppose that $\rho \in \mathcal{I}([A_0 \in B_0.\Psi^{\beta_0}])$. Then there exists an environment $\rho' \in \mathcal{I}_{gcs}(\Psi^{\beta_0})$ such that $\rho(A_0) = \rho'(B_0)$. By definition of $\rho' \in \mathcal{I}_{gcs}(\Psi^{\beta_0})$, there exists a ground instance G_0' of G_0 and a ground derivation $\mathcal{D}$ from G_0' to G such that G selects $\rho'(B_0^{\beta_0})$ via ρ. Let G be $C_1, \ldots, C_m$, and it must be the case that there exists an $i \leq m$ such that C_i is $\rho(A_0)$. Hence, $\mathcal{D}$ can be extended, using the rule $A_0 \leftarrow A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$ with environment ρ, into a derivation $\mathcal{D}'$ from G_0' to $C_1, \ldots, C_{i-1}, \rho(A_1^{\alpha_1}), \ldots, \rho(A_n^{\alpha_n}), C_{i+1}, \ldots, C_m$. Since the atom selection function of the interleaved semantics may select any of the atoms from this goal, it follows that $\rho \in \mathcal{I}(\Psi^{\alpha_j})$, $j = 1..n$.

Now consider a constraint of the second form, and suppose that $\rho \in \mathcal{I}([A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_n \in B_n.\Psi^{\beta_n}])$. This means that (a) $\rho(A_0) \in \mathcal{I}_{gcs}(B_0.\Psi^{\beta_0})$ and (b) $\rho(A_i) \in \mathcal{I}_{gcs}(B_i.\Psi^{\beta_i})$, $i = 1..n$. Reasoning as before, (a) implies that there exists a ground derivation $\mathcal{D}$ from G_0' to G such that G_0' is an instance of G_0 and G is of the form $C_1, \ldots, C_m$ where, for some i, $1 \leq i \leq m$, C_i is $\rho(A_0)$. Clearly $\mathcal{D}$ can be extended to give a ground derivation $\mathcal{D}'$ from G_0' to

$$C_1, \ldots, C_{i-1}, \rho(A_1^{\alpha_1}), \ldots, \rho(A_n^{\alpha_n}), C_{i+1}, \ldots, C_m.$$

Also, (b) implies that there exist $\rho_i \in \mathcal{I}_{gcs}(\Psi^{\beta_i})$ such that $\rho(A_i) = \rho_i(B_i)$, $i = 1..n$. By definition of $\rho_i \in \mathcal{I}_{gcs}(\Psi^{\beta_i})$, ρ_i is such that there is a ground derivation $\mathcal{D}_i$ from some ground instance of G_0 such that $\mathcal{D}_i$ minimally solves $\rho_i(B_i)$, and hence minimally solves $\rho(A_i)$. The derivations $\mathcal{D}'$ and $\mathcal{D}_j$ can be combined using proposition 9 (see the proof for the left-to-right semantics for more details) to obtain a derivation $\mathcal{D}''$ from G_0' to $C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_m$. Clearly $\mathcal{D}''$ returns from $R^{\alpha_{n+1}}$ under ρ and so $\rho \in \mathcal{I}_{gcs}(\alpha_{n+1})$.

Now suppose that the constraint corresponds to the initial goal $\mathcal{G}_0$. Let G_0 be of the form $\leftarrow A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$, and let the label of this goal be α_{n+1}. Then the constraint must have one of the following two forms

$$\begin{aligned} \Psi^{\alpha_j} &\supseteq [\,] \\ \Psi^{\alpha_{n+1}} &\supseteq \left[A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_n \in B_n.\Psi^{\beta_n}\right] \end{aligned}$$

where $1 \leq j \leq n$ and the β_i, $i = 1..n$, are rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible. The argument that such a constraint is satisfied by $\mathcal{I}_{gcs}$ is essentially a repeat of the argument for constraints corresponding to program rules. The main observation is that, for any environment ρ, there is a single step derivation $\mathcal{D}$ consisting of the goal $\rho(A_1^{\alpha_1}), \ldots, \rho(A_n^{\alpha_n})$. This implies that $\rho \in \mathcal{I}_{gcs}(\Psi^{\alpha_j})$ since any atom in the goal $\rho(A_1^{\alpha_1}), \ldots, \rho(A_n^{\alpha_n})$ may be selected in the interleaved semantics. Moreover, if $\rho \in [A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_n \in B_n.\Psi^{\beta_n}]$ then we can show that there exist derivations $\mathcal{D}_j$, $j = 1..n$, such that $\mathcal{D}_j$ solves $\rho(A_j)$, and then proposition 9 can be applied to combine $\mathcal{D}$ with the $\mathcal{D}_j$ to prove that $\rho \in \mathcal{I}_{gcs}(\Psi^{\alpha_{n+1}})$. $\Box$

Lemma 5 *If $\mathcal{I}$ is a model of $\mathcal{EC}_P$ then $\mathcal{I}_{gcs} \subseteq \mathcal{I}$.*

Proof (for left-to-right semantics): Let G_0 be the initial goal and let $\mathcal{I}$ be a model of $\mathcal{EC}_P$. To prove the lemma, we need to show that if $\rho \in \mathcal{I}_{gcs}(\Psi^\alpha)$ then $\rho \in \mathcal{I}(\Psi^\alpha)$ where α ranges over all program labels. Recalling the definition of $\mathcal{GCS}_P$, this can be reduced to the following property: if $\mathcal{D}$ is a ground derivation from some ground instance G_0' of G_0 then

(a) if $\mathcal{D}$ selects A^α under ρ then $\rho \in \mathcal{I}(\Psi^\alpha)$, and

(b) if $\mathcal{D}$ returns from R^α under ρ then $\rho \in \mathcal{I}(\Psi^\alpha)$.

We shall prove this using an induction argument whose hypothesis is:

Let $\mathcal{D}$ be a ground derivation and suppose that the sequence of goals in $\mathcal{D}$ is $G'_0, G'_1, \ldots, G'_N$ where G'_0 is a ground instance of G_0. If, for some n, $0 \leq n \leq N$, it is the case that

(0) all atoms A^α in G'_n that are solved by $\mathcal{D}$ are such that $A \in \mathcal{I}(B.\Psi^\beta)$ for some head atom B^β in P,

then the following conditions hold for all $k \leq n$:

(1) if $\mathcal{D}$ selects A^α and $\mathcal{D}$ introduces A^α at step k using ρ then $\rho \in \mathcal{I}(\Psi^\alpha)$, and

(2) if $\mathcal{D}$ uses rule R^α and environment ρ at step k and the atoms introduced by step k are solved in the subsequent steps of $\mathcal{D}$ then $\rho \in \mathcal{I}(\Psi^\alpha)$.

In essence, parts (1) and (2) of this hypothesis are a restricted form of (a) and (b). Note that in the case where $n = N$, part (0) of the induction hypothesis becomes vacuously true, and parts (1) and (2) are equivalent to (a) and (b). Hence, if $n = N$, then the induction hypothesis is equivalent to the lemma.

We prove the hypothesis by induction on n. In the base case $n = 0$ and G'_n is an instance of the initial goal G_0. Let G_0 have the form $A_1^{\alpha_1}, \ldots, A_r^{\alpha_r}$ and let the label of G_0 be α_{r+1}. Then G'_n has the form $\rho(A_1^{\alpha_1}), \ldots, \rho(A_r^{\alpha_r})$ where ρ is some environment. Assume that (0) holds, and consider (1) and (2). Since $k \leq 0$, the only possible value for k is 0.

To prove (1), suppose that $\mathcal{D}$ selects A^α and $\mathcal{D}$ introduces A^α at step 0 using ρ'. This means that A^α must appear in G'_0, and so there exists a j such that A^α is $\rho(A_j^{\alpha_j})$. Now, $\mathcal{D}$ follows a left-to-right execution strategy and so it is easy to verify that $\mathcal{D}$ must solve $A_1^{\alpha_1}, \ldots, A_{j-1}^{\alpha_{j-1}}$. This is because if G_N contains any descendants of $A_1^{\alpha_1}, \ldots, A_{j-1}^{\alpha_{j-1}}$ then these descendants must appear to the left of $\rho(A_j^{\alpha_j})$ and hence $\rho(A_j^{\alpha_j})$ could not be selected from G_N. Since $\mathcal{D}$ solves $A_1^{\alpha_1}, \ldots, A_{j-1}^{\alpha_{j-1}}$, assumption (0) can be applied to prove that for all i, $1 \leq i < j$, there exists a head atom $B_i^{\beta_i}$ in P such that $\rho(A_i^{\alpha_i}) \in \mathcal{I}(B_i.\Psi^{\beta_i})$. Now, corresponding to G_0, $\mathcal{EC}_P$ contains the constraint

$$\Psi^{\alpha_j} \quad \supseteq \quad \left[A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_{j-1} \in B_{j-1}.\Psi^{\beta_{j-1}}\right].$$

Clearly $\rho \in \mathcal{I}([A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_{j-1} \in B_{j-1}.\Psi^{\beta_{j-1}}])$, and since $\mathcal{I}$ is a model of $\mathcal{EC}_P$, it follows that $\rho \in \mathcal{I}(\Psi^{\alpha_j})$.

To prove (2), suppose that $\mathcal{D}$ uses rule R^α and environment ρ at step 0 and the atoms introduced by step 0 are solved in the subsequent steps of $\mathcal{D}$. This implies that $\mathcal{D}_N$ is *empty* and that all atoms in G_0 are solved. Arguing as above, assumption (0) can be used to show that for all i, $1 \leq i < m$, there exists a head atom $B_i^{\beta_i}$ in P such that $\rho(A_i^{\alpha_i}) \in \mathcal{I}(B_i.\Psi^{\beta_i})$. Since $\mathcal{EC}_P$ contains the constraint

$$\Psi^{\alpha_{r+1}} \quad \supseteq \quad \left[A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_r \in B_r.\Psi^{\beta_r}\right].$$

and it follows that $\rho \in \mathcal{I}(\Psi^{\alpha_j})$.

To prove the inductive case, suppose that for some n' the hypothesis holds for $n = n' - 1$, and we seek to prove that the hypothesis when $n = n'$. Assume that (0) holds. Let the step from G_{n-1} to G_n be of the form

$$C_1^{\gamma_1}, \ldots, C_n^{\gamma_n} \xrightarrow[1, \alpha_{r+1}, \rho]{} \rho(A_1^{\alpha_1}), \ldots, \rho(A_r^{\alpha_r}), C_2^{\gamma_2}, \ldots, C_n^{\gamma_n}$$

where the rule in P with label α_{r+1} is $A_0 \leftarrow A_1^{\alpha_1}, \ldots, A_r^{\alpha_r}$ such that $\rho(A_0) = C_1$. Now, consider the derivation $\mathcal{D}'$ consisting of the first $n-1$ steps of $\mathcal{D}$. Clearly $\mathcal{D}'$ is a derivation from G'_0 to G'_{n-1} that selects $C_1^{\gamma_1}$. Now, the hypothesis is assumed to hold for $n = n' - 1$, and when applied to derivation $\mathcal{D}'$, condition (0) is vacuous and so (1) implies that $\rho' \in \mathcal{I}(\Psi^{\gamma_1})$ where ρ' is the environment used to introduce $C_1^{\gamma_1}$. Let C'_1 be the body atom in P with label γ_1. Then $\rho(A_0) = C_1 = \rho'(C'_1)$ and hence

$$\rho(A_0) \in \mathcal{I}(C'_1.\Psi^{\gamma_1}) \tag{4.5}$$

Now, corresponding to the rule $A_0 \leftarrow A_1^{\alpha_1}, \ldots, A_r^{\alpha_r}$, $\mathcal{EC}_P$ contains constraints of the form $\Psi^{\alpha_{r'+1}} \supseteq [A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_{r'} \in B_{r'}.\Psi^{\beta_{r'}}]$ where r' ranges over $1..r$, β_0 ranges over body atom labels such that $B_0^{\beta_0}$ is a body atom in P and A_0 and B_0 are compatible, and the β_i, $i \geq 1$, range over rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible. Now (4.5) establishes that there is a body atom $B_0^{\beta_0}$ in P such that $\rho(A_0) \in \mathcal{I}(B_0^{\beta_0})$. Combining this with the fact that $\mathcal{I}$ satisfies all constraints in $\mathcal{EC}_P$ proves that, for $r' = 1..r$,

$$\text{if } \rho(A_1) \in \mathcal{I}(B_1^{\beta_1}), \ldots, \rho(A_{r'}) \in \mathcal{I}(B_{r'}^{\beta_{r'}}) \ \text{ then } \rho \in \mathcal{I}(\Psi^{\alpha_{r'+1}}) \tag{4.6}$$

where $\beta_1, \ldots, \beta_{r'}$ are body atom labels. Note that the condition that A_i and B_i are compatible, which is associated with the construction of an environment constraint, is subsumed by the condition $\rho(A_i) \in \mathcal{I}(B_i^{\beta_i})$ and hence does not appear in (4.6).

Before proving (1) and (2), we first establish that condition (0) holds for G'_{n-1}. This is necessary because the assumption that the hypothesis holds for $\mathcal{D}$ in the case where $n = n' - 1$ is a statement of the form: if (0) holds with $n = n' - 1$ then (1) and (2) hold with $n = n' - 1$. Hence, to make use of this assumption, we shall have to show that "(0) holds with $n = n' - 1$". Specifically, we need to show that

> all atoms A^α in $G_{n'-1}$ that are solved by $\mathcal{D}$ are such that
> $A^\alpha \in \mathcal{I}(B.\Psi^\gamma)$ for some head atom B^γ in P.

To prove this, let A^α be an atom in $C_1^{\gamma_1}, \ldots, C_n^{\gamma_n}$ that is solved in G. Now, if A^α is not $C_1^{\gamma_1}$ then A^α appears in G'_n and so (0) implies that $A^\alpha \in \mathcal{I}(B.\Psi^\gamma)$ for some head atom B^γ in P. On the other hand, if A^α is $C_1^{\gamma_1}$ then $\rho(A_1^{\alpha_1}), \ldots, \rho(A_r^{\alpha_r})$ must be solved in G. Now, the assumption (0) (for G_n) implies that, for $i = 1..r$, $\rho(A_i^{\alpha_i}) \in \mathcal{I}(B_i.\Psi^{\beta_i})$ for some head atom $B_i^{\beta_i}$ in P. Combining this with (4.6) proves that $\rho \in \mathcal{I}(\Psi^{\alpha_{r+1}})$. Since $A = C_1 = \rho(A_0)$, this implies that $A \in \mathcal{I}(A_0.\Psi^{\alpha_{r+1}})$.

Now, consider (1) and (2). If $k < n$, then (1) and (2) follow from the induction hypothesis and the fact that (0) holds for G'_{n-1} (which has just been proved). Now consider the case where $k = n$. To prove (1), suppose that $\mathcal{D}$ selects A^α and $\mathcal{D}$ introduces A^α at step k using ρ'. Since $k = n$, it must be the case that $r \geq 1$, $\rho' = \rho$ and A^α is $\rho(A_j^{\alpha_j})$ for some j in the range $1..r$. Since $\mathcal{D}$ follows a left-to-right execution strategy, it is easy to verify that $\mathcal{D}$ must solve $\rho(A_1^{\alpha_1}), \ldots, \rho(A_{j-1}^{\alpha_{j-1}})$. This is because if G_N contains any descendants of $\rho(A_1^{\alpha_1}), \ldots, \rho(A_{j-1}^{\alpha_{j-1}})$ then these descendants would appear to the left of $\rho(A_j^{\alpha_j})$ and hence $\rho(A_j^{\alpha_j})$ could not be selected from G_N. Since $\mathcal{D}$ solves $A_1^{\alpha_1}, \ldots, A_{j-1}^{\alpha_{j-1}}$, assumption (0) can be applied to prove that for all i, $1 \leq i < j$, there exists a head atom $B_i^{\beta_i}$ in P such that $\rho(A_i^{\alpha_i}) \in \mathcal{I}(B_i.\Psi^{\beta_i})$. Hence (4.6) implies that $\rho \in \mathcal{I}(\Psi^{\alpha_j})$.

To prove (2), suppose that $\mathcal{D}$ uses rule R^α and environment ρ' at step k and the atoms introduced by step k are solved in the subsequent steps of $\mathcal{D}$. Since $k = n$, it must be the case that α is α_{r+1}, $\rho = \rho'$ and the subsequent steps of $\mathcal{D}$ solve $\rho(A_1^{\alpha_1}), \ldots, \rho(A_r^{\alpha_r})$. From the assumption that (0) holds for G'_n, it follows that for each i, $i = 1..r$, $\rho(A_i) \in \mathcal{I}(B_i.\Psi^{\beta_i})$ for some head atom $B_i^{\beta_i}$ in P. Hence (4.6) implies that $\rho \in \mathcal{I}(\Psi^{\alpha_{r+1}})$. $\square$

Proof (for interleaving semantics): The proof for the top-down interleaving semantics is very similar to that for the top-down left-to-right case. The induction hypothesis used is identical and the base and inductive cases for (2) are also the same. The only difference is in the base and inductive cases for (1) where the proof is in fact simpler than in the left-to-right case.

Recall that in the base case $n = 0$ and G'_n is an instance of the initial goal G_0. Let G_0 have the form $A_1^{\alpha_1}, \ldots, A_r^{\alpha_r}$ and let the label of G_0 be α_{r+1}. Then G'_n has the form $\rho(A_1^{\alpha_1}), \ldots, \rho(A_r^{\alpha_r})$ where ρ is some environment. Assume that (0) holds, and consider (1). Since $k \leq 0$, the only possible value for k is 0. Suppose that $\mathcal{D}$ selects A^α and $\mathcal{D}$ introduces A^α at step 0 using ρ'. This means that A^α must appear in G'_0, and so there exists an i such that A^α is $\rho(A_i^{\alpha_i})$. Now, the environment constraints corresponding to $\mathcal{G}_0$ include the constraint $\Psi^{\alpha_i} \supseteq [\,]$. and it follows that $\rho \in \mathcal{I}(\Psi^{\alpha_i})$.

The proof of (2) when $n = 0$ given previously for the left-to-right semantics is in fact independent of the selection strategy at hand. Hence it can be used here without modification; we omit the repetition. This completes the base case.

Now suppose that for some n' the hypothesis holds for $n = n' - 1$, and we seek to prove that the hypothesis when n is n'. Assume that (0) holds. Let the step from G_{n-1} to G_n be of the form

$$C_1^{\gamma_1}, \ldots, C_n^{\gamma_n} \xrightarrow[i,\,\alpha_{r+1},\,\rho]{} C_1^{\gamma_1}, \ldots, C_{i-1}^{\gamma_{i-1}}, \rho(A_1^{\alpha_1}), \ldots, \rho(A_r^{\alpha_r}), C_{i+1}^{\gamma_{i+1}}, \ldots, C_n^{\gamma_n}$$

where the rule in P with label α_{r+1} is $A_0 \leftarrow A_1^{\alpha_1}, \ldots, A_r^{\alpha_r}$ such that $\rho(A_0) = C_1$. Now, consider the derivation $\mathcal{D}'$ consisting of the first $n - 1$ steps of $\mathcal{D}$. Clearly $\mathcal{D}'$ is a derivation from G'_0 to G'_{n-1} that selects $C_i^{\gamma_i}$. Now, the hypothesis is assumed to hold for $n = n' - 1$, and when applied to derivation $\mathcal{D}'$, condition (0) is vacuous and so (1) implies that $\rho' \in \mathcal{I}(\Psi^{\gamma_i})$ where ρ' is the environment used to introduce $C_i^{\gamma_i}$. Let C'_i be the body atom in P with label γ_i. Then $\rho(A_0) = C_i = \rho'(C'_i)$ and hence $\rho(A_0) \in \mathcal{I}(C'_i.\Psi^{\gamma_i})$. Now, corresponding to the rule $A_0 \leftarrow A_1^{\alpha_1}, \ldots, A_r^{\alpha_r}$, $\mathcal{EC}_P$ contains constraints of the following two forms:

$$\begin{array}{rcl} \Psi^{\alpha_j} & \supseteq & \left[A_0 \in B_0.\Psi^{\beta_0}\right] \\ \Psi^{\alpha_{r+1}} & \supseteq & \left[A_0 \in B_0.\Psi^{\beta_0}, A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_r \in B_r.\Psi^{\beta_r}\right] \end{array}$$

where j ranges over $1..r$, β_0 ranges over body atom labels such that $B_0^{\beta_0}$ is a body atom in P and A_0 and B_0 are compatible, and the β_i, $i \geq 1$, range over rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible. Now we have just established that there is a body atom $B_0^{\beta_0}$ in P such that $\rho(A_0) \in \mathcal{I}(B_0^{\beta_0})$. Combining this with the fact that $\mathcal{I}$ satisfies all constraints in $\mathcal{EC}_P$ proves that,

$$\begin{array}{l} \text{(a) } \rho \in \mathcal{I}(\Psi^{\alpha_j}),\ \ j = 1..r \\ \text{(b) if } \rho(A_1) \in \mathcal{I}(B_1^{\beta_1}), \ldots, \rho(A_r) \in \mathcal{I}(B_r^{\beta_r}) \ \text{ then } \rho \in \mathcal{I}(\Psi^{\alpha_{r+1}}) \end{array} \tag{4.7}$$

where $\beta_1, \ldots, \beta_r$ are body atom labels.

As in the left-to-right case, we begin by proving that condition (0) holds for G'_{n-1}. That is, we prove that all atoms A^α in $G_{n'-1}$ that are solved by $\mathcal{D}$ are such that $A^\alpha \in \mathcal{I}(B.\Psi^\gamma)$ for some head atom B^γ in P. To prove this, let A^α be an atom in $C_1^{\gamma_1}, \ldots, C_n^{\gamma_n}$ that is solved in G. Now, if A^α is not $C_i^{\gamma_i}$ then A^α appears in G'_n and so (0) implies that $A^\alpha \in \mathcal{I}(B.\Psi^\gamma)$ for some head atom B^γ in P. On the other hand, if A^α is $C_i^{\gamma_i}$ then $\rho(A_1^{\alpha_1}), \ldots, \rho(A_r^{\alpha_r})$ must be solved in G. Now, the assumption (0) (for G_n) implies that, for $j = 1..r$, $\rho(A_j^{\alpha_j}) \in \mathcal{I}(B_j.\Psi^{\beta_j})$ for some head atom $B_j^{\beta_j}$ in P. Combining this with part (b) of (4.7) proves that $\rho \in \mathcal{I}(\Psi^{\alpha_{r+1}})$. Since $A = C_1 = \rho(A_0)$, this implies that $A \in \mathcal{I}(A_0.\Psi^{\alpha_{r+1}})$.

Now, consider (1) and (2). If $k < n$, then (1) and (2) follow from the induction hypothesis and the fact that (0) holds for G'_{n-1} (which has just been proved). Now consider the case where $k = n$. To prove (1), suppose that $\mathcal{D}$ selects A^α and $\mathcal{D}$ introduces A^α at step k using ρ'. Since $k = n$, it must be the case that $r \geq 1$, $\rho' = \rho$ and A^α is $\rho(A_j^{\alpha_j})$ for some j in the range $1..r$. That $\rho \in \mathcal{I}(\Psi^{\alpha_j})$ follows immediately from part (a) of (4.7).

Again, the proof for the inductive case of (2) that was given for the left-to-right semantics is independent of the selection strategy at hand, and can be used here without modification. This completes the inductive case. $\square$

Theorem 4 (Correctness of Top-Down Constraints)
For all programs P and all labels α in P,

$$lm(\mathcal{EC}_P)(\Psi^\alpha) \ =_{var(\alpha)} \ \{\rho : \rho \models E \text{ and } E \in \mathcal{CS}_P(\alpha)\}.$$

Proof: Let α be a label in P. From Lemma 4 and Lemma 5 it follows that $lm(\mathcal{EC}_P)(\Psi^\alpha)$ is equal to $\mathcal{I}_{gcs}(\Psi^\alpha)$. By definition, $\mathcal{I}_{gcs}(\Psi^\alpha)$ is just $\mathcal{GCS}_P(\alpha)$. Finally, Lemma 3 proves that $\mathcal{GCS}_P(\alpha) =_{var(\alpha)} \{\rho : \rho \models E$ and $E \in \mathcal{CS}_P(\alpha)\}$, and this completes the proof. $\square$

4.6.2 Bottom-up Semantics

The proof of correctness for the bottom-up semantics is similar to that for the top-down semantics, although it is substantially simpler. Again we define a ground notion of derivability corresponding to the earlier definition of bottom-up derivability. Specifically, a ground atom $\leftarrow A$ is *ground bottom-up derivable* if there is a ground instance $A \leftarrow A_1, \ldots, A_n$ of a rule in P such that the ground atoms $A_1, \ldots, A_n$ are bottom-up derivable. As before, ground bottom-up derivability is closely linked to bottom-up derivability.

Proposition 10 *If $\langle E : A\rangle$ is bottom-up derivable and $\rho \models E$ then $\rho(A)$ is ground bottom-up derivable.*

Proof: The proof proceeds by induction on the definition of bottom-up derivable. Let $\langle E_i : B_i\rangle$, $i = 1..n$, be bottom-up derivable states that satisfy the proposition, let $A \leftarrow A_1, \ldots, A_n$ be a rule in P, and let $\theta_1, \ldots, \theta_n$ be renaming substitutions such that $var(A, A_1, \ldots, A_n)$, $var(\theta_1(E_1), \theta_1(B_1))$, $\ldots$, $var(\theta_n(E_n), \theta_n(B_n))$ are all disjoint sets. Under these assumptions, we need to show that $\langle E : A\rangle$ satisfies the proposition, where E is $(A_1 = \theta_1(B_1)) \wedge \theta_1(E_1) \wedge \cdots \wedge (A_n = \theta_n(B_n)) \wedge \theta_n(E_n)$. To this end, assume that $\rho \models E$. This implies that ρ is a model for each E_i and it follows that $\rho(B_i)$ is ground bottom-up derivable, $i = 1..n$, because each $\langle E_i : B_i\rangle$ is assumed to satisfy the proposition. $\rho \models E$ also implies that $\rho(A_i)$ is identical to $\rho(\theta_i(B_i))$. Hence $\rho(A) \leftarrow \rho(A_1), \ldots, \rho(A_n)$ is a ground instance of a rule in P, and it follows that $\rho(A)$ is ground bottom-up derivable. $\square$

Proposition 11 *If A is ground bottom-up derivable then there exists a bottom-up derivable state $\langle E : B\rangle$ and an environment ρ such that $\rho \models E$ and $\rho(B)$ is A.*

Proof: The proof proceeds by induction on the definition of ground bottom-up derivable. Let $A_1, \ldots, A_n$ be ground bottom-up derivable states that satisfy the proposition, let $A \leftarrow A_1, \ldots, A_n$ be a ground instance of a rule in P, and we seek to show that the proposition holds for A. Since $A_1, \ldots, A_n$ satisfy the proposition, it must be the case that for each i, $i = 1..n$, there exists a bottom-up derivable state $\langle E_i : B'_i\rangle$ and an environment ρ_i such that $\rho_i \models E$ and $\rho_i(B'_i)$ is A_i. Also, since $A \leftarrow A_1, \ldots, A_n$ is a ground instance of a rule in P, there exists a rule in P of the form $B \leftarrow B_1, \ldots, B_n$ and an environment ρ_R such that $A = \rho_R(B)$ and $A_i = \rho_R(B_i)$, $i = 1..n$. Let $\theta_1, \ldots, \theta_n$ be renaming substitutions such that $var(B, B_1, \ldots, B_n)$, $var(\theta_1(E_1), \theta_1(B'_1)), \ldots, var(\theta_n(E_n), \theta_n(B'_n))$ are disjoint sets, and define E to be $(B_1 = \theta_1(B'_1)) \wedge \theta_1(E_1) \wedge \cdots \wedge (B_n = \theta_n(B'_n)) \wedge \theta_n(E_n)$. Define an environment ρ as follows:

$$\rho(X) = \begin{cases} \rho_i(\theta_i^{-1}(X)) & \text{if } X \in var(\theta_i(E_i), \theta_i(B'_i)), \quad 1 \leq i \leq n \\ \rho_R(X) & \text{otherwise} \end{cases}$$

By definition, $\rho(\theta_i(X)) = \rho_i(\theta_i^{-1}(\theta_i(X))) = \rho_i(X)$ for each variable X appearing in E_i, and hence $\rho \models \theta_i(E_i)$ iff $\rho_i \models E_i$. It has already been established that $\rho_i \models E_i$ and so $\rho \models \theta_i(E_i)$, $i = 1..n$. Similarly, the definition of ρ implies that $\rho(\theta_i(B'_i)) = \rho_i(B'_i)$, $i = 1..n$. Combining this equality with previously established equalities proves that

$$\rho(\theta_i(B'_i)) = \rho_i(B'_i) = A_i = \rho_R(B_i) = \rho(B_i), \ \ i = 1..n$$

and it follows that ρ satisfies each equation $B_i = \theta_i(B'_i)$, and hence $\rho \models E$. Moreover, $\rho(B) = \rho_R(B) = A$. Hence $\langle E : B\rangle$ is a bottom-up derivable state and ρ is an environment such that $\rho \models E$ and $\rho(B)$ is A. $\square$

Using ground bottom-up derivable, a ground collecting semantics can now be defined.

Definition 9 *The ground bottom-up collecting semantics of a logic program P is the mapping $\mathcal{GCS}_P$ such that, for each rule label α,*

$$\mathcal{GCS}_P(\alpha) \quad \stackrel{\text{def}}{=} \quad \left\{\rho : \begin{array}{l} \textit{the rule in } P \textit{ with label } \alpha \textit{ is } A \leftarrow A_1, \ldots, A_n \textit{ and} \\ \rho(A_1), \ldots, \rho(A_n) \textit{ are ground bottom-up derivable} \end{array}\right\}.$$

□

Using the previous two propositions, which relate ground derivations with derivations, the following correspondence can be established.

Lemma 6 *For all rule labels α,*

$$\mathcal{GCS}_P(\alpha) \;\; =_{var(\alpha)} \;\; \{\rho : \rho \models E \textit{ for some } E \in \mathcal{CS}_P(\alpha)\}.$$

Proof: The first part of the proof shows that for any environment ρ in the set on the left hand side of this equation, there exists an environment ρ' in the right hand side such that $\rho =_{var(\alpha)} \rho'$. Suppose that $\rho \in \mathcal{GCS}_P(\alpha)$ and let the rule with label α be $A \leftarrow A_1, \ldots, A_n$. This implies that $\rho(A_1), \ldots, \rho(A_n)$ are ground bottom-up derivable. Hence, by proposition 11, there exist bottom-up derivable states $\langle E_i : B_i \rangle$ and environments ρ_i such that $\rho_i \models E_i$ and $\rho_i(B_i)$ is $\rho(A_i)$, $i = 1..n$. Now, let $\theta_1, \ldots, \theta_n$ be renaming substitutions such that $var(A, A_1, \ldots, A_n)$, $var(\theta_1(E_1), \theta_1(B_1))$, $\ldots$, $var(\theta_n(E_n), \theta_n(B_n))$ are all disjoint sets. By definition, $\langle E : A \rangle$ is bottom-up derivable where E is $(A_1 = \theta_1(B_1)) \wedge \theta_1(E_1) \wedge \cdots \wedge (A_n = \theta_n(B_n)) \wedge \theta_n(E_n)$. Now, define an environment ρ' as follows:

$$\rho'(X) = \begin{cases} \rho_i(\theta_i^{-1}(X)) & \text{if } X \in var(\theta_i(E_i), \theta_i(B_i')), \quad 1 \le i \le n \\ \rho(X) & \text{otherwise} \end{cases}$$

Using reasoning similar to that used in Proposition 11, it is easy to verify that ρ' is a model of each $\theta_i(E_i)$ and that $\rho'(A_i) = \rho'(\theta_i(B_i))$, $i = 1..n$. This implies that $\rho' \models E$, and so $\rho' \in \{\rho : \rho \models E$ for some $E \in \mathcal{CS}_P(\alpha)\}$. Since $\rho' =_{var(\alpha)} \rho$, the proof for this direction is complete.

Conversely, suppose that $\rho \models E$ such that $E \in \mathcal{CS}_P(\alpha)$. If the rule in P with label α is $A \leftarrow A_1, \ldots, A_n$, then by definition of $\mathcal{CS}_P$, there must exist bottom-up derivable states $\langle E_i : B_i \rangle$ and renamings θ_i, $1 \leq i \leq n$, such that $var(A, A_1, \ldots, A_n)$, $var(\theta_1(E_1), \theta_1(B_1))$, ..., $var(\theta_n(E_n), \theta_n(B_n))$ are disjoint sets, and such that E is $(A_1 = \theta_1(B_1)) \wedge \theta_1(E_1) \wedge \cdots \wedge (A_n = \theta_n(B_n)) \wedge \theta_n(E_n)$. Since $\rho \models E$, it follows that $\rho \models \theta_i(E_i)$, $i = 1..n$. This implies that $\rho \circ \theta_i \models E_i$. From Proposition 10 it follows that $\rho \circ \theta_1(B_1), \ldots, \rho \circ \theta_n(B_n)$ are all bottom-up derivable. Also, $\rho \models E$ implies that $\rho(A_i) = \rho \circ \theta_i(B_i)$, and hence $\rho(A_1), \ldots, \rho(A_n)$ are bottom-up derivable. Thus $\rho \in \mathcal{GCS}_P(\alpha)$, and the lemma is proved. $\square$

Note that the set of ground derivable goals corresponds to the usual bottom-up semantics of logic programs [7] defined using the T_p function (see section 4.2 on page 63). Specifically, $\leftarrow A$ is ground bottom-up derivable iff $A \in \mathit{lfp}(T_{\mathrm{P}})$. The above theorem can now be used to show that the bottom-up semantics defined in Section 4.2 is equivalent to the standard T_p bottom-up semantics in the following sense:

$$\mathit{lfp}(T_p) = \{\rho(A) : \langle E : A \rangle \text{ is bottom-up derivable and } \rho \models E\}$$

We now complete the proof of the correctness of the bottom-up semantics. Recall that the proof so far has connected bottom-up collecting semantics with bottom-up *ground* collecting semantics. The remaining part of the proof connects the ground collecting semantics with the environment constraints, and consists of two lemmas. Again $\mathcal{I}_{gcs}$ denotes the mapping such that $\mathcal{I}_{gcs}(\Psi^\alpha) = \mathcal{GCS}_P(\alpha)$. We begin with the following easy property.

Proposition 12 *If A_0^α is a head atom in P then each atom in $\mathcal{I}_{gcs}(A_0.\Psi^\alpha)$ is ground bottom-up derivable.*

Proof: Let $A_0 \leftarrow A_1, \ldots, A_n$ be the rule in P with label α and suppose that $A \in \mathcal{I}_{gcs}(A_0.\Psi^\alpha)$. This implies that there is an environment ρ such that $\rho \in \mathcal{I}_{gcs}(\Psi^\alpha)$ and $A = \rho(A_0)$. Now, $\rho \in \mathcal{I}_{gcs}(\Psi^\alpha)$ implies that $\rho(A_1), \ldots, \rho(A_n)$ are ground bottom-up derivable. It immediately follows that $\rho(A_0)$ is ground bottom-up derivable, and since $A = \rho(A_0)$, this completes the proof that A is ground bottom-up derivable. $\square$

Lemma 7 *$\mathcal{I}_{gcs}$ is a model of $\mathcal{EC}_P$.*

Proof: Each constraint in $\mathcal{EC}_P$ is of the form

$$\Psi^{\alpha} \supseteq \left[A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_n \in B_n.\Psi^{\beta_n}\right]$$

such that the rule in P with label α is $A_0 \leftarrow A_1, \ldots, A_n$ and each β_i is a rule label such that $B_i^{\beta_i}$ is a head atom in P and B_i and A_i are compatible. Now, suppose that $\rho \in \mathcal{I}_{gcs}([A_1 \in B_1.\Psi^{\beta_1}, \ldots, A_n \in B_n.\Psi^{\beta_n}]$. Hence $\rho(A_i) \in \mathcal{I}_{gcs}(B_i.\Psi^{\beta_i})$, $i = 1..n$. Proposition 12 implies that each $\rho(A_i)$ is bottom-up derivable and it is immediate that $\rho \in \mathcal{I}_{gcs}(\alpha)$. $\square$

Lemma 8 *If $\mathcal{I}$ is a model of $\mathcal{EC}_P$ then $\mathcal{I}_{gcs} \subseteq \mathcal{I}$.*

Proof: The proof uses induction on the definition of ground bottom-up derivability. To establish the basis for this induction, we first refine the definition of bottom-up derivability. Specifically, define that $\rho(A)$ is *ground bottom-up derivable with index k* if there is an environment ρ and a rule in P of the form $A \leftarrow A_1, \ldots, A_n$ such that each $\rho(A_i)$ is ground bottom-up derivable with index k_i and $k = 1 + k_1 + \cdots + k_n$. Clearly A is ground bottom-up derivable iff there exists a $k \geq 1$ such that A is ground bottom-up derivable with index k. The lemma is now established by proving the following hypothesis:

> For all rule labels α and environments ρ, if the rule in P with label α is $A \leftarrow A_1, \ldots, A_n$ and each $\rho(A_i)$ is bottom-up derivable with index $k_i < k$, then $\rho \in \mathcal{I}(\Psi^{\alpha})$.

It is easy to verify, using the definition of $\mathcal{GCS}_P$, that if this hypothesis holds for all k then $\mathcal{I}_{gcs} \subseteq \mathcal{I}$. The proof of the hypothesis proceeds by induction on k. Suppose that the hypothesis holds for all k less than k' where k' is some non-negative integer, and we seek to show the hypothesis when $k = k'$. Let α be a rule label, let $A \leftarrow A_1, \ldots, A_n$ be the rule in P with label α and let ρ be an environment such that each $\rho(A_i)$ is bottom-up derivable with index $k_i < k$. This means that for each i there is a rule label β and an environment ρ' such that the $B \leftarrow B_1, \ldots B_r$ is the rule in P with label β, $\rho'(B) = \rho(A_i)$ and, for each j, $j = 1..n$, $\rho'(B_j)$ is ground bottom-up derivable with index $l_j < k_i$. Since the hypothesis is assumed to hold for k_i, it follows that $\rho' \in \mathcal{I}(\Psi^{\beta})$ and so $\rho(A_i) \in \mathcal{I}(B.\Psi^{\beta})$. In summary then, for $i = 1..n$ there exists a head atom $C_i^{\gamma_i}$ in P such that $\rho(A_i) \in \mathcal{I}(C_i.\Psi^{\gamma_i})$. Now, by construction, $\mathcal{EC}_P$ contains the constraint

$$\Psi^{\alpha} \supseteq [A_1 \in C_1.\Psi^{\gamma_1}, \ldots, A_n \in C_n.\Psi^{\gamma_n}].$$

Since $\mathcal{I}$ is assumed to be a model of $\mathcal{EC}_P$, it immediately follows that $\rho \in \mathcal{I}(\Psi^{\alpha})$. □

Theorem 5 (Correctness of Bottom-Up Constraints)
For all programs P and all rule labels α in P,

$$lm(\mathcal{EC}_P)(\Psi^{\alpha}) \;\; =_{var(\alpha)} \;\; \{\rho : \rho \models E \textit{ and } E \in \mathcal{CS}_P(\alpha)\}.$$

Proof: This theorem follows immediately from lemmas 6, 7 and 8. □

Chapter 5

Set Based Approximation

Environment constraints provide a flexible framework for reasoning about programs. In particular, by re-interpreting the basic constructs of the constraints, we can define approximations of a program's collecting semantics. Such an approach is used in this chapter to define the set based approximation of a program. We begin by developing interpretations of the environment constraints that do not contain inter-variable dependencies. Central to these interpretations is the treatment of program variables as sets, and this is formalized using *set environments*, which are mappings from program variables into sets. The resulting interpretations are called *set based interpretations*. The *set based approximation* of a program is defined to be the smallest set based interpretation that is a model of the program's environment constraints. That is, the smallest (standard) interpretation of the environment constraints gives the collecting semantics of the program, and the smallest set based interpretation of the environment constraints gives the set based approximation of (the collecting semantics of) the program.

A key part of this chapter involves formalizing inter-variable dependencies. While this notion is fairly clear at an intuitive level, it has no *a priori* definition. In fact there are a number of potential definitions, and these are outlined during the development of set based interpretations. Importantly, there is one definition that is more natural than the others, and this definition is used as the basis for set based program approximation. This provides a key justification of our claim that set based approximation makes exactly one approximation: all inter-variable dependencies are ignored.

5.1 Summary of Environment Constraints

We begin by summarizing the form of the environment constraints used to characterize the collecting semantics of programs. We first consolidate some of the basic definitions for logic and imperative programs. Let Σ denote a common set of function symbols that subsumes the function symbols used in imperative programs as well as the function and predicate symbols used in logic programs. Let VAR denote a common set of program variables for both logic and imperative programs. Note that VAR is infinite. A *program term* is either a program variable or of the form $f(t_1, \ldots, t_n)$ or $f_{(j)}^{-1}(t_1)$ where each t_i is a program term, $f \in \Sigma$ and $1 \leq j \leq n$. Note that the definition of program term encompasses both logic and imperative program terms as well as logic program atoms. A *value* is a program term constructed only from symbols in Σ. An *environment* is a mapping from VAR into values. Again we shall write $[X_1 \mapsto v_1, \ldots, X_n \mapsto v_n]$ to denote an environment that maps X_i into v_i, $i = 1..n$. We shall frequently abuse this notation in examples and write expressions such as $\{[X \mapsto v_1, Y \mapsto v_2], [X \mapsto v_3, Y \mapsto v_4]\}$, denoting the infinite set of all environments that either map X to v_1 and Y to v_2 or else map X to v_3 and Y to v_4.

An *atomic program condition* is of the form $s = t$, $\neg(s = t)$, $match_f(t)$ or $\neg(match_f(t))$ where X is a program variable, f is a function symbol from Σ and s and t are program terms. A *program condition* is a disjunction of conjunctions of atomic program conditions. Note that this definition of program condition is, strictly speaking, a restriction of the previous definition since it only allows "disjunction normal form" program conditions to be written. This is done for convenience. Moreover, it is an inconsequential restriction since any program condition can be easily rewritten into an equivalent disjunction of conjunctions of basic program conditions.

An *environment variable* is a variable that ranges over sets of environments, and is denoted by the symbol Ψ. For each program point μ, there is a distinguished environment variable denoted Ψ^μ, whose purpose is to describe the environments corresponding to point μ. An *environment expression* is either

- an environment variable Ψ;
- the constant $\top$;

- $\Psi[X \mapsto t]$, where X is a program variable and t is a program term;
- $\Psi[cond]$, where $cond$ is a program condition;
- $[A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]$, where the A_i and B_i are program terms.

An *environment constraint* is of the form $\Psi \supseteq ee$ where ee is an environment expression.

We now review the meaning of environment constraints. First, the meaning of program terms is defined. Given an environment ρ, the meaning $\rho(t)$ of a term t is defined by extending ρ as follows:

- $\rho(f(t_1, \ldots, t_n)) \;=\; f(\rho(t_1), \ldots, \rho(t_n))$.
- $\rho(f^{-1}_{(i)}(t')) \;=\; v_i$ if $\rho(t') = f(v_1, \ldots, v_n)$ for some values $v_1, \ldots, v_n$.

Note that $\rho(f^{-1}_{(i)}(t'))$ is undefined if $\rho(t')$ is not of the form $f(v_1, \ldots, v_n)$. We write $\rho \rhd t$ if t is defined under ρ. Next, the meaning of program conditions is defined. As mentioned previously, each program condition is written as a disjunction of conjunctions of atomic program conditions. We write $\rho \rhd cond$ if $\rho \rhd t$ for each program term t appearing in $cond$. Now, where ρ is an environment such that $\rho \rhd cond$, the relation $\rho \models cond$ is defined as follows:

- $\rho \models s = t$ iff $\rho(s) = \rho(t)$;
- $\rho \models \neg(s = t)$ iff $\rho(s) \neq \rho(t)$;
- $\rho \models match_f(t)$ iff $\rho(t)$ is of the form $f(v_1, \ldots, v_n)$;
- $\rho \models \neg(match_f(t))$ iff $\rho(t)$ is of the form $g(v_1, \ldots, v_n)$ where $f \neq g$;
- $\rho \models cond_1 \wedge cond_2$ iff $\rho \models cond_1$ and $\rho \models cond_2$
- $\rho \models cond_1 \vee cond_2$ iff either $\rho \models cond_1$ or $\rho \models cond_2$.

If it is not the case that $\rho \rhd cond$ then $\rho \models cond$ is not defined. Finally, an *interpretation* of environment constraints is a mapping from each environment variable into a set of environments. Given such an interpretation $\mathcal{I}$, the meaning of an environment expression is defined as follows:

- $\mathcal{I}(\Psi)$ is already defined.
- $\mathcal{I}(\top) = \{\text{all environments}\}$.
- $\mathcal{I}(\Psi[X \mapsto t]) = \{\rho[X \mapsto \rho(t)] : \rho \in \Theta\}$ where Θ is $\{\rho \in \mathcal{I}(\Psi) : \rho \rhd t\}$.
- $\mathcal{I}(\Psi[cond]) = \{\rho \in \Theta : \rho \models cond\}$ where Θ is $\{\rho \in \mathcal{I}(\Psi) : \rho \rhd cond\}$.
- $\mathcal{I}([A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]) = \{\rho : \rho(A_i) \in \mathcal{I}(B_i.\Psi_i),\ i = 1..n\}$.

The expression $\mathcal{I}(B.\Psi)$ denotes the set of ground atoms $\{\rho(B) : \rho \in \mathcal{I}(\Psi)\}$. An interpretation is a *model* of a conjunction of environment constraints $\mathcal{EC}$ if $\mathcal{I}(\Psi) \supseteq \mathcal{I}(ee)$ for each constraint $\Psi \supseteq ee$ in $\mathcal{EC}$.

5.2 Inter-Variable Dependencies in $\mathcal{EC}_P$

What we ultimately desire is a simple, intuitive and decidable definition of an approximation to a program's (collecting) semantics. One natural way to obtain such an approximation is by developing a notion of *approximate* interpretation of the environment constraints. Given such a notion, an approximate semantics can be defined using the smallest approximate interpretation that is a model of the constraints. That is, the least (standard) model of the environment constraints defines the program's *exact* semantics, and the least approximate model defines the program's *approximate* semantics.

In essence, this is the approach used to define the set based approximation of a program. Specifically, to obtain a notion of program approximation that ignores inter-variable dependencies, we develop an interpretation of the environment constraints that is free of such dependencies. Now, recall that environment constraints are interpreted by starting with a mapping $\mathcal{I}$ from environment variables into arbitrary sets of environments, and then extending this mapping in an obvious way to map environment expressions into sets of environments. Inter-variable dependencies arise in two places in this interpretation. First, they may be present in the collections of environments specified by $\mathcal{I}$. Second, they may be present in the process of extending $\mathcal{I}$ to map from environment expressions into sets of environments. We consider these possibilities in turn.

Inter-Variable Dependencies in Sets of Environments

We begin our development of an interpretation of the environment constraints that ignores inter-variable dependencies, by considering the inter-variable dependencies that may be present in collections of environments. For example, suppose that one of the collections of environments specified by an interpretation is the following collection of environments

$$\{[X \mapsto a, Y \mapsto b], [X \mapsto c, Y \mapsto d]\},$$

where it is assumed that the only program variables of interest are X and Y. Inter-variable dependencies are present in this collection of environments in the sense that whenever X takes the value a, then it must be the case that Y takes the value b, and whenever X takes c, Y must take d.

Intuitively, a collection of environments is free of inter-variable dependencies if fixing the value of one or more variable *does not affect* the values that other variables may take. More concretely, let Θ be a collection of environments and suppose that we choose an environment ρ from Θ and modify ρ so that it maps X into v where v is one of the "possible values" for X. If Θ is free of inter-variable dependencies, then we expect that the modified environment $\rho[X \mapsto v]$ should also be an element of Θ. For example, consider once again the collection of environments $\{[X \mapsto a, Y \mapsto b], [X \mapsto c, Y \mapsto d]\}$; denote this collection by Θ. On choosing $[X \mapsto a, Y \mapsto b]$ from Θ and modifying this environment to map Y into d (one of the "possible values" for Y), we obtain the environment $[X \mapsto a, Y \mapsto d]$ and this is not contained in Θ. Hence, as expected, Θ is not free of inter-variable dependencies. However, a collecting of environments that is free of inter-variable dependencies can be obtained by augmenting Θ with the environments $[X \mapsto a, Y \mapsto d]$, $[X \mapsto c, Y \mapsto b]$.

More formally, let Θ be a collection of environments and define, in the context of Θ, that *v is a possible value for X* if there exists an environment ρ in Θ such that $\rho(X) = v$. Now, consider the environments $\hat{\rho}$ such that there exists an environment ρ in Θ and for each X either (a) $\hat{\rho}$ agrees with ρ on X or else (b) $\hat{\rho}(X)$ is a possible value for X. Define that Θ is *free of inter-variable dependencies* if Θ contains all such environments $\hat{\rho}$. This provides a fairly direct formalization of the above intuitions about inter-variable dependencies. However, observe that case (a) of the construction of $\hat{\rho}$ is redundant because if $\hat{\rho}(X) = \rho(X)$ then $\hat{\rho}(X)$ is a possible value for X. Hence, Θ is free of inter-variable dependencies if Θ contains all environments

$\hat{\rho}$ such that, for each X, $\hat{\rho}(X)$ is a possible value for X. More compactly, Θ is free of inter-variable dependencies if

$$\rho \in \Theta \quad \text{iff} \quad \text{for all } X,\ \rho(X) \text{ is a possible value for } X.$$

If a collection Θ satisfies this condition, then Θ can be completely described by the sets of possible values that it defines for each program variable. In other words, such a collection Θ can be viewed as a specification of a set of values for each program variable. Hence, collections Θ that are free of inter-variable dependencies can be characterized as *set based* collections of environments.

Definition 10 (Set Based Environment Collections) *A collection Θ of environments is set based if there exists a mapping F from program variables into sets of values such that*

$$\rho \in \Theta \quad \textit{iff} \quad \rho(X) \in F(X) \textit{ for all program variables } X. \qquad \Box$$

For example, $\{[X{\mapsto}a, Y{\mapsto}b],\ [X{\mapsto}a, Y{\mapsto}d],\ [X{\mapsto}c, Y{\mapsto}b],\ [X{\mapsto}c, Y{\mapsto}d]\}$ is set based because it can be represented by the function F that maps X into $\{a, c\}$ and Y into $\{b, d\}$.

The mapping F in Definition 10 can be thought of as a representation of the Θ. In most cases, there is only one set mapping F that represents a set based collection Θ. However, in the boundary case where Θ is the empty set, there are many possible choices for F since any F that maps at least one program variable into the empty set is a candidate. It is convenient to refine the notion of set mapping to obtain a unique representation of set based collections of environments as follows.

Definition 11 (Set Environments) *A set environment ϱ is a mapping from program variables into sets of values such that if ϱ maps some program variable into the empty set then it maps all program variables into the empty set.*

Clearly a collection Θ of environments is set based if there exists a set environment ϱ such that $\rho \in \Theta$ iff $\rho(X) \in \varrho(X)$ for all X. Moreover, there is a one-to-one correspondence between set based collections of environments and set environments. It is convenient to identify a set environment with the collection of environments it represents. To this end, we shall frequently treat a set environment ϱ as a set of environments and write $\rho \in \varrho$ to denote that $\rho(X) \in \varrho(X)$ for each program variable X.

Now, as a first step towards a definition of program approximation that ignores inter-variable dependencies, we define a notion of program approximation based on set environments. Specifically, where P is a program, let $approx_P$ denote the least interpretation that is both a model of $\mathcal{EC}_P$ and maps each environment variable into a set based collection of environments. Since $approx_P$ is a model of $\mathcal{EC}_P$, it follows that it must be larger than $lm(\mathcal{EC}_P)$. It follows that $lm(\mathcal{EC}_P) \subseteq approx_P$ and so $approx_P$ is a safe approximation of the collecting semantics of P.

The definition of $approx_P$ leads to a very simple definition of program approximation; however it is not decidable. The reason is that although $approx_P$ removes inter-variable dependencies in the collections of environments, it does not remove dependencies that may be introduced *through* the action of program variables. For example, consider the imperative program consisting of the single statement $X := pair(X, X)$. The environment constraints corresponding to this program are $\Psi^{\downarrow 1} \supseteq \Psi^{\uparrow 1}[X \mapsto pair(X, X)]$ and $\Psi^{\uparrow 1} \supseteq \top$. For this program, $lm(\mathcal{SC}_P)$ and $approx_P$ coincide and both map $\Psi^{\downarrow 1}$ into the set based collection of environments

$$[X \mapsto \{pair(v, v) : v \text{ is a value}\}].$$

Hence dependencies may be introduced by variables even though the collection of environments $\mathcal{I}(\Psi)$ does not contain inter-variable dependencies. It is these kinds of dependencies that lead to the undecidability of $approx_P$.

Dependencies Introduced By Program Variables

To describe the kinds of dependencies that may be introduced through the action of program variables, we must first consider dependencies in sets of values. A set of values contains dependencies if there are relationships between the components of each value. For example, consider the two sets of terms $\{f(a, b), f(c, d)\}$ and $\{f(g^n(a), g^n(b)) : n \geq 0\}$, where g^n is used to abbreviate n applications of g, so that $g^2(c)$ denotes $g(g(c))$. Both of these sets contain dependencies. Such dependencies may be present in the sets of values variables may be bound to at run-time. Some of these dependencies

logic program	imperative program
$p(pair(X, X)).$ $\leftarrow p(Y).$	$Y := pair(X, X);$
$q(pair(a, b)).$ $q(pair(c, d)).$ $\leftarrow q(Y).$	$Y := pair(a, b);$ **if** $match_{pair}(X)$ **then** $Y := pair(c, d);$

Figure 5.1: **Different Ways of Introducing Dependencies**

are explicitly introduced by the action of program variables, while others are introduced through the merging of different computation paths. To illustrate this, consider the four programs in Figure 5.1. In both of the top two programs, the set of values for Y after program execution is $\{pair(v, v) : v \text{ is a value}\}$; the dependencies in these two examples are introduced by the variable Y. In contrast, after execution of either of the bottom two programs in Figure 5.1, the value of Y is either $pair(a, b)$ or $pair(c, d)$; the dependencies here are introduced by merging computation paths. Note that the dependencies introduced by variables may be infinite in nature, whereas the dependencies introduced by the merging of computation paths are essentially finite. This has important decidability implications.

At the heart of set based analysis is the tradeoff between infinite sets of values, dependencies and decidability. First observe that since we do not employ an approximation of the underlying computation values, we must deal directly with infinite sets of values. Also note that the notion of intersection is inherently present in set based analysis because a conditional expression of the form $X = Y$ naturally leads to the intersection of the sets of values for X and Y. Now, as we have just noted above, the action of variables may introduce dependencies that are unbounded in nature. Such dependencies imply that sets of the form $\{f(g^n(a), g^n(b)) : n \geq 0\}$ may arise. This is suggestive of the expressive power of context free grammars. Since the intersection of two context free grammars is not recursive, it is not surprising that unbounded dependencies must be curtailed to obtain decidability.

A more concrete explanation of the undecidability of $approx_P$ is based on the observation that the semantic constructs of the language are sufficiently powerful that reasoning involving unbounded dependencies can still be carried out, even though inter-variable dependencies cannot appear in sets of environments. Specifically, let P be an imperative program, and consider coding P as an essentially equivalent imperative program that involves only one variable. To do this, let $X_1, \ldots, X_n$ be the variables in P, and let Env be a new variable that shall range over lists of length n, representing environments. Let Env^i denote $car(cdr^{i-1}(Env))$ (to access i^{th} element of list in Env) and let $[s_1, \ldots, s_n]$ denote the list of length n whose elements are, in order, $s_1, \ldots, s_n$. Now, construct a program P' from P by first replacing all assignment statements $X_i := t$ by $Env := [X_1, \ldots, X_{i-1}, t, X_{i+1}, \ldots, X_n]$, and then replacing all occurrences of X_i by Env^i, $1 \leq i \leq n$. Clearly the resulting P' is equivalent to P in the sense that if P starts execution from environment $[X_1 \mapsto u_1, \ldots, X_n \mapsto u_n]$ and reaches program point μ with environment $[X_1 \mapsto v_1, \ldots, X_n \mapsto v_n]$ then execution of P' starting from environment $[Env \mapsto [u_1, \ldots, u_n]]$ reaches point μ with environment $[Env \mapsto [v_1, \ldots, v_n]]$. Moreover, P' contains only one variable. It follows that $lm(\mathcal{EC}_{P'}) = approx_{P'}$. Hence, any oracle for deciding $\rho \in approx_{P'}(\Psi^\mu)$ can be used to decide $\rho \in lm(\mathcal{EC}_P)(\Psi^\mu)$.

A similar kind of construction is possible for logic programs. The important observation here is that $approx_P$ still allows an equality predicate to be defined. Specifically, let P be an arbitrary logic program. First rewrite P into an equivalent program that contains only unary predicate symbols (this can be easily done by introducing new function symbols f_n of every arity n, and then systematically rewriting each atom $p(s_1, \ldots, s_n)$ into $p(f_n(s_1, \ldots, s_n)))$. Second, rewrite each rule $p(s) \leftarrow B_1, \ldots, B_n$ into $p(X) \leftarrow eq(X, s), B_1, \ldots, B_n$ where X is a variable that does not already appear in the rule. Third, add the rule $eq(X, X)$. Call the resulting program P'. Clearly P' mimics P in a very straightforward manner. Moreover, $approx_{P'}$ and $lm(\mathcal{EC}_{P'})$ coincide in the following sense: for all head atoms A^μ in P'

$$approx_{P'}(\Psi^\mu) =_{var(A)} lm(\mathcal{EC}_{P'})(\Psi^\mu).$$

In both of these constructions, the key observation is that variable dependencies appear in $approx_P$ when an environment is applied to a term with multiple occurrences of a variable. Such dependencies must be eliminated if a decidable program approximation is to be obtained.

The approach for eliminating these dependencies is based on the use of set environments. Recall that set environments were introduced to represent set based collections of environments. Not only can set environments be treated as collections of environments, but they can also be treated as environment-like mappings. Whereas an environment is pointwise in the sense that it specifies a single value for each variable, a set environment is set-wise in the sense that it specifies a set of values for each variable. In analogy with environments, there is a natural notion of changing the binding of a variable in a set environment. Specifically, where ϱ is a set environment and S is a set of values, the notation $\varrho[X \mapsto S]$ denotes the set environment that maps all variables into the empty set if either S or $\varrho(X)$ is the empty set, and otherwise is the set environment that maps X into S and agrees with ϱ on program variables different from X.

Just as environments are used to assign meanings to program terms and conditions, we can use set environments to assign an alternative "set based" meaning to program terms and conditions. For example, suppose that X is the only variable of interest and consider an environment expression $\Psi[X \mapsto pair(X, X)]$ corresponding to an assignment statement. Suppose that $\mathcal{I}(\Psi)$ is the set environment $\{[X \mapsto \{a, b\}]\}$. Under the normal interpretation, the environments in $\mathcal{I}(\Psi)$ are considered one at a time and applied to the program term $pair(X, X)$ to obtain a binding for X. The result is that $\mathcal{I}(\Psi[X \mapsto pair(X, X)])$ yields the collection of environments $\{[X \mapsto pair(a, a)], [X \mapsto pair(b, b)]\}$. However, if $\mathcal{I}(\Psi)$ is applied to $pair(X, X)$ as a set environment, then each <u>occurrence</u> of X is treated as a set, resulting in $\{X \mapsto pair(a, a), X \mapsto pair(a, b), X \mapsto pair(b, a), X \mapsto pair(b, b)\}$. Hence, by using set environments to interpret program terms and conditions, we obtain a natural method for eliminating inter-variable dependencies.

We now outline how this approach can be generalized to interpret arbitrary environment expressions. This forms the core part of the definition of set based interpretation of environment constraints. In particular we shall develop interpretations of the expressions $\Psi[X \mapsto t]$, $\Psi[cond]$ and $[A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]$. In the remainder of this section, let $\mathcal{I}$ denote an interpretation that maps environment variables into set environments. Let $\mathcal{A}$ denote the operator that maps a collection Θ of environments into a set environment, denoted $\mathcal{A}(\Theta)$, as follows

$$\mathcal{A}(\Theta)(X) \stackrel{\text{def}}{=} \{\rho(X) : \rho \in \Theta\}.$$

Note that $\mathcal{A}(\Theta)$ is the smallest set environment containing Θ.

Consider first the set based interpretation of an environment expression of the form $\Psi[X \mapsto t]$, corresponding to an assignment statement. The standard interpretation of such an expression is

$$\mathcal{I}(\Psi[X \mapsto t]) \;=\; \{\rho[X \mapsto \rho(t)] : \rho \in \Theta\} \text{ where } \Theta \text{ is } \{\rho \in \mathcal{I}(\Psi) : \rho \rhd t\}.$$

Now, the set based interpretation of this expression is obtained by treating Θ as a set environment ϱ and applying ϱ directly to the program term t instead of applying Θ to t on an environment by environment basis. The resulting set of values is then used to appropriately update ϱ. Specifically, we define that the set based interpretation of $\Psi[X \mapsto t]$ under $\mathcal{I}$ is given by

$$\mathcal{I}\left(\Psi[X \mapsto t]\right) \stackrel{\text{def}}{=} \varrho[X \mapsto \varrho(t)] \text{ where } \varrho \text{ is } \mathcal{A}\left(\{\rho \in \mathcal{I}(\Psi) : \rho \rhd t\}\right)$$

Next consider an environment expression of the form $\Psi[cond]$ corresponding to imperative program conditions. Now, the effect of $[cond]$ is to restrict environments in $\mathcal{I}(\Psi)$. In the standard interpretation of conditions, environments are applied to program terms one at a time, and a notion of variable dependency can arise that is similar to the dependencies arising in the interpretation of $\Psi[X \mapsto t]$. For example, suppose that f is a binary function symbol and consider the program consisting of the single statement **if** $\left(f^{-1}_{(1)}(Y) = X \wedge f^{-1}_{(2)}(Y) = X\right)$ **then** *Seq*. Let the label of the first statement of *Seq* be β and let the last of the last be γ. The environment constraints for this program consist of the constraints

$$\begin{array}{lcl}
\Psi^{\uparrow 1} & \supseteq & \top \\
\Psi^{\uparrow \beta} & \supseteq & \Psi^{\uparrow 1}\left[f^{-1}_{(1)}(Y) = X \wedge f^{-1}_{(2)}(Y) = X\right] \\
\Psi^{\downarrow 1} & \supseteq & \Psi^{\uparrow 1}\left[\neg\left(f^{-1}_{(1)}(Y) = X\right) \vee \neg\left(f^{-1}_{(2)}(Y) = X\right)\right] \\
\Psi^{\downarrow 1} & \supseteq & \Psi^{\downarrow \gamma}
\end{array}$$

together with the appropriate constraints for *Seq*. For this program, $approx_P$ maps $\Psi^{\uparrow 1}$ into the set of all environment and maps $\Psi^{\uparrow \beta}$ into the collection of environments

$$\{[X \mapsto u, Y \mapsto f(v, v)] : \text{ for all } u \text{ and } v\}.$$

Again, dependencies are introduced through the treatment of distinct occurrences of variables.

To ignore such dependencies, we develop an interpretation of environment expressions $\Psi[cond]$ using set environments. First consider the standard interpretation of an expression of the form $\Psi[cond]$:

$$\mathcal{I}(\Psi[cond]) \;=\; \{\rho \in \Theta : \rho \models cond\} \text{ s.t. } \Theta \text{ is } \{\rho \in \mathcal{I}(\Psi) : \rho \rhd cond\}.$$

This involves three components: (i) the process of collecting environments ρ such that $\rho \in \mathcal{I}(\Psi)$ and $\rho \rhd cond$ into a set Θ, (ii) the relation $\rho \models cond$, and (iii) the process of collecting environments ρ such that $\rho \in \Theta$ and $\rho \models cond$. These three components are replaced by set environment counterparts as follows: (i) is replaced by the process of collecting environments ρ such that $\rho \in \mathcal{I}(\Psi)$ and $\rho \rhd cond$ into a set environment ϱ, (ii) is replaced by the relation $\rho \models_\varrho cond$ that defines the notion of an environment ρ satisfying *cond in the context* of set environment ϱ, and (iii) is replaced by the process of collecting environments ρ such that $\rho \in \Theta$ and $\rho \models_\varrho cond$ into a set environment. We now outline the definition of $\rho \models_\varrho cond$; the full details can be found in the next section.

Consider the atomic condition $f^{-1}_{(1)}(Y) = X$. Such a condition essentially represents two restrictions on an environment ρ. The left hand side of the equality restricts the values of Y and the right hand side restricts X. In other words, $\rho \models f^{-1}_{(1)}(Y) = X$ can be thought of as the combination of the two restrictions (i) $\rho(f^{-1}_{(1)}(Y)) \in \{\rho(X)\}$ and (ii) $\rho(X) \in \{\rho(f^{-1}_{(1)}(Y))\}$. In the set based interpretation of this basic condition, these two restrictions are explicitly separated, and the set environment ϱ is used to interpret X in (i) and $f^{-1}_{(1)}(Y)$ in (ii). Hence, $\rho \models_\varrho f^{-1}_{(1)}(Y) = X$ is satisfied if

$$\rho(f^{-1}_{(1)}(Y)) \in \varrho(X) \text{ and } \rho(X) \in \varrho(f^{-1}_{(1)}(Y))$$

Using this interpretation, consider again the environment expression

$$\Psi^{\uparrow 1}\left[f^{-1}_{(1)}(Y) = X \wedge f^{-1}_{(2)}(Y) = X\right],$$

and suppose that $\mathcal{I}(\Psi^{\uparrow 1})$ is the set of all environments. Instead of obtaining the set $\{[X{\mapsto}u, Y{\mapsto}f(v,v)] \;:\; \text{for all } u \text{ and } v\}$, we now obtain the set $\{[X{\mapsto}u, Y{\mapsto}f(v_1,v_2)] \;:\; \text{for all } u,\ v_1 \text{ and } v_2\}$. Hence dependencies are no longer introduced.

The interpretation of an atomic condition $\neg(s \neq t)$ can similarly be obtained. Again $\rho \models \neg(s \neq t)$ can be split into (i) $\exists v(\rho(s) \neq v \wedge v \in \{\rho(t)\}$ and (ii) $\exists v(\rho(t) \neq v \wedge v \in \{\rho(s)\}$. The set based interpretation of this basic condition uses ϱ to interpret t in (i) and s in (ii). Specifically, $\rho \models_\varrho \neg(s \neq t)$ holds if

$$\exists v\ (v \in \varrho(t) \wedge v \neq \rho(s)) \text{ and } \exists v\ (v \in \varrho(s) \wedge v \neq \rho(t)).$$

In other words, whenever a term is used in such a way that a new value is built up, then a set environment is used to interpret the term (since otherwise infinite dependencies may be introduced). In contrast, if a term is used for the purpose of *restricting* the values of program variables, then the term is interpreted using a (normal) environment (because no dependencies may be introduced by this process). Note that atomic conditions such as $match_f(s)$ and $\neg(match_f(s))$ serve only to restrict the values of the program variables contained in s, and so they cannot introduce dependencies. Hence set environments are not needed for their interpretation.

Finally, consider an expression $[A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]$ corresponding to a logic program rule. Again the standard interpretation of this expression may introduce dependencies through multiple occurrences of a variable. For example, consider the logic program consisting of the rules $eq(X, X)$ (labeled with 1) and the goal $\leftarrow eq(Y, pair(Z, Z))$ (labeled with 2). The bottom-up environment constraints for this program are

$$\begin{array}{lll} \Psi^1 & \supseteq & [\,] \\ \Psi^2 & \supseteq & [eq(Y, f(Z, Z)) \in eq(X, X).\Psi^1] \end{array}$$

and $approx_P$ maps Ψ^2 into the collection of environments

$$\{[X \mapsto u, Y \mapsto pair(v, v)] : \text{ for all } u \text{ and } v\}.$$

Dependencies are introduced here through the two occurrences of Z.

Set environments can be used to eliminate such dependencies in a manner similar to that used for environment expressions of the form $\Psi[X \mapsto t]$. First, recall that the standard interpretation of an expression of the form $\mathcal{I}([A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n])$ is

$$\{\rho : \text{ for each } i,\ \rho(A_i) \in \{\rho'(B_i) : \rho' \in \mathcal{I}(\Psi_i)\}\}.$$

This contains the two kinds of components: (i) the application of a set of environments $\mathcal{I}(\Psi_i)$ to the atom B_i to obtain a set of ground atoms, and (ii) the collection of environments ρ. In the set interpretation of this expression, components in (i) are replaced by the application of the set environment $\mathcal{I}(\Psi_i)$ to the atom B_i, and the component in (ii) is replaced by the collection of environments ρ into a set environment. For example, the constraint $\Psi^2 \supseteq [eq(Y, f(Z,Z)) \in eq(X,X).\Psi^1]$ is interpreted as

$$\Psi^2 \supseteq \mathcal{A}\left(\{\rho : \rho(eq(Y, f(Z,Z))) \in \varrho(eq(X,X)) \text{ where } \varrho \text{ is } \mathcal{I}(\Psi^1)\}\right).$$

More generally, the environment expression $[A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]$ is interpreted under $\mathcal{I}$ as the set

$$\mathcal{I}\left([A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]\right) = \mathcal{A}\left(\{\rho : \rho(A_i) \in \mathcal{I}(\Psi_i)(B_i)\}\right)$$

In summary, the use of set environments to interpret environment constraints corresponds to treating each variable as a set of values. Moreover, the use of sets in the interpretation of environment constraints provides a simple and natural way to ignore all dependencies that are introduced through the action of variables. For this reason we equate set based analysis with analysis in which all inter-variable dependencies are ignored (and no other approximations are made). Importantly, this uniform and intuitive reading of environment constraints leads to an accurate and decidable analysis.

5.3 Set Based Interpretation of $\mathcal{EC}_P$

We now present the complete details of the set based interpretation of the environment constraints. We begin by summarizing some definitions introduced in the previous section. A *set environment* ϱ is a mapping from program variables into sets of values such that if ϱ maps some program variable into the empty set then it maps all program variables into the empty set. We identify set based collections of environments with set environments, and write $\rho \in \varrho$ to denote that $\rho(X) \in \varrho(X)$ for each program variable X. If ϱ is a set environment and S is a set of values then $\varrho[X \mapsto S]$ denotes the set environment that maps all variables into the empty set if either S or $\varrho(X)$ is the empty set, and otherwise is the set environment that maps X into S and agrees with ϱ on program variables different from X. $\mathcal{A}$ denotes the operator that maps a collection Θ of environments into the smallest set environment containing Θ and is defined as follows:

$$\mathcal{A}(\Theta)(X) \stackrel{\text{def}}{=} \{\rho(X) : \rho \in \Theta\}.$$

Note that the fixed points of $\mathcal{A}$ are exactly the set based collections of environments.

Just as (pointwise) environments are extended to become partial functions from program terms into values, so set environments are extended to become functions from program terms into <u>sets</u> of values. Let ϱ be a set environment and let t be a program term. If ϱ is the set environment $\perp$ that maps all variables into the empty set then $\varrho(t)$ is the empty set, regardless of t. Otherwise $\varrho(t)$ is defined as follows:

- If t is a program variable, then $\varrho(t)$ is already defined.
- If t is $f(t_1, \ldots, t_n)$ then $\varrho(t)$ is $\{f(v_1, \ldots, v_n) : v_i \in \varrho(t_i)\}$.
- If t is $f_{(i)}^{-1}(s)$ then $\rho(t)$ is $\{v_i : f(v_1, \ldots, v_n) \in \varrho(s)\}$.

We now use set environments to interpret program conditions. We define a relation $\rho \models_\varrho cond$ to be read as ρ satisfies *cond in the context* of the set environment ϱ. As noted previously, it is assumed that program conditions are first written into disjunctive normal form. Let ρ be an environment, let ϱ be a set environment and define that:

- $\rho \models_\varrho s = t$ iff $\rho(s) \in \varrho(t)$ and $\rho(t) \in \varrho(s)$.
- $\rho \models_\varrho \neg(s = t)$ iff $\exists v\, (v \in \varrho(t) \wedge v \neq \rho(s))$ and $\exists v\, (v \in \varrho(s) \wedge v \neq \rho(t))$.
- $\rho \models_\varrho match_f(s)$ if $\rho(s)$ is of the form $f(v_1, \ldots, v_n)$.
- $\rho \models_\varrho \neg match_f(s)$ if $\rho(s)$ is not of the form $f(v_1, \ldots, v_n)$.
- $\rho \models_\varrho cond_1 \wedge cond_2$ iff $\rho \models_\varrho cond_1$ and $\rho \models_\varrho cond_2$.
- $\rho \models_\varrho cond_1 \vee cond_2$ iff either $\rho \models_\varrho cond_1$ or $\rho \models_\varrho cond_2$.

A *set based interpretation* $\mathcal{I}$ is a mapping from each environment variable into a set environment. Such a mapping can be extended to map from environment expressions *ee* into a set environment as follows.

- $\mathcal{I}(\Psi)$ is already defined.
- $\mathcal{I}(\top) \;=\; \{\text{all environments}\}$.
- $\mathcal{I}(\Psi[X \mapsto t]) \;=\; \varrho[X \mapsto \varrho(t)]$ where ϱ is $\mathcal{A}\left(\{\rho \in \mathcal{I}(\Psi) : \rho \rhd t\}\right)$
- $\mathcal{I}(\Psi[cond]) \;=\; \mathcal{A}\left(\left\{\rho : \begin{array}{c}\rho \in \varrho \\ \rho \models_{\varrho} cond\end{array}\right\}\right)$ where ϱ is $\mathcal{A}\left(\left\{\rho : \begin{array}{c}\rho \in \mathcal{I}(\Psi) \\ \rho \rhd cond\end{array}\right\}\right)$
- $\mathcal{I}([A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]) \;=\; \mathcal{A}\left(\left\{\rho : \begin{array}{ccc}\rho(A_1) & \in & \mathcal{I}(\Psi_1)(B_1) \\ \vdots & \vdots & \vdots \\ \rho(A_n) & \in & \mathcal{I}(\Psi_n)(B_n)\end{array}\right\}\right)$

Note that in the last part of this definition, each $\mathcal{I}(\Psi_i)$ is a set environment, and so the expression $\mathcal{I}(\Psi_i)(B_i)$ denotes the set of ground atoms resulting from applying $\mathcal{I}(\Psi_i)$ to B_i. A *set based model* of a collection of environment constraints is a set based environment that satisfies each constraint in the collection. Using set based models, we can now define the set based approximation of a program.

Definition 12 (Set Based Approximation) *Let P be a program and let $\mathcal{EC}_P$ be the environment constraints of P. Then the set based approximation of P, denoted sba_P, is the least set based model of $\mathcal{EC}_P$.* $\square$

Importantly, sba_P is a model of $\mathcal{EC}_P$.

Proposition 13 *For all programs P, sba_P is a model of $\mathcal{EC}_P$.*

Proof: The proposition is proved by showing that any set based model of $\mathcal{EC}_P$ is a model of $\mathcal{EC}_P$. To this end, let $\mathcal{I}$ be a set based interpretation. Now, when $\mathcal{I}$ is extended to map from environment expressions into sets of environments, either the set based interpretation rules can be used, or else $\mathcal{I}$ can be treated as a (normal) interpretation and the rules for (normal) interpretation used. Let $SET(\mathcal{I}, ee)$ denote the set of environments obtained when the environment expression ee is interpreted under $\mathcal{I}$ using the set based interpretation rules. Let $NML(\mathcal{I}, ee)$ denote the set of environments obtained when the environment expression ee is interpreted under $\mathcal{I}$ using the (normal) interpretation rules. To prove the proposition, if is sufficient to show that $SET(\mathcal{I}, ee) \supseteq NML(\mathcal{I}, ee)$.

Clearly if ee is either an environment variable or $\top$ then $SET(\mathcal{I}, ee) = NML(\mathcal{I}, ee)$. Before proving the remaining three cases, it is convenient to prove that

$$\text{if } \rho \in \varrho \text{ then } \rho(t) \in \varrho(t) \tag{5.8}$$

where t is a program term and ϱ is a set environment that is defined on t. This fact can be easily establish by structural induction on t.

Using (5.8), it is now straightforward to complete the proof. First suppose that ee is $\Psi[X \mapsto t]$. If $\rho \in NML(\mathcal{I}, \Psi[X \mapsto t])$ then there exists an environment ρ' such that $\rho' \rhd t$, $\rho' \in \mathcal{I}(\Psi)$ and $\rho = \rho'[X \mapsto \rho'(t)]$. Now, let ϱ be $\mathcal{A}(\{\rho \in \mathcal{I}(\Psi) : \rho \rhd t\})$. Clearly $\rho' \in \varrho$ and so $\rho'(t) \in \varrho(t)$ by (5.8). It follows from the definition of $\varrho[X \mapsto S]$ that $\rho \in \varrho[X \mapsto \varrho(t)]$.

Now consider an environment expression of the form $\Psi[cond]$. Using (5.8) it is easy to verify the following property by structural induction on *cond*:

$$\text{if } \rho \rhd cond,\ \rho \models cond, \text{ and } \rho \in \varrho \text{ then } \rho \models_{\varrho} cond \tag{5.9}$$

where *cond* is a program condition and ϱ is a set environment. To complete the proof for $\Psi[cond]$, suppose that $\rho \in NML(\mathcal{I}, \Psi[cond])$. This implies that $\rho \in \mathcal{I}(\Psi)$, $\rho \rhd cond$ and $\rho \models cond$. Let ϱ be $\mathcal{A}(\{\rho \in \mathcal{I}(\Psi) : \rho \rhd cond\})$. Clearly $\rho \in \varrho$ and so $\rho \models_{\varrho} cond$ by (5.9). Hence $\rho \in SET(\mathcal{I}, \Psi[cond])$.

Finally, consider $[A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]$ and suppose that $\rho \in NML(\mathcal{I}, [A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n])$. This implies that for $i = 1..n$, there exists an environment ρ_i such that $\rho(A_i) = \rho_i(B_i)$ and $\rho_i \in \mathcal{I}(\Psi_i)$. From (5.8) it follows that $\rho(A_i) \in \mathcal{I}(\Psi_i)(B_i)$. Hence $\rho \in SET(\mathcal{I}, \Psi[cond])$. $\square$

This proposition implies that $sba_P \supseteq lm(\mathcal{SC}_P)$. Since we have already proved that $lm(\mathcal{SC}_P)$ corresponds to $\mathcal{CS}_P$ (see Theorem 1 for imperative programs and Theorem 3 for logic programs) it follows that sba_P is a conservative approximation of the collecting semantics of a program in the following sense:

Theorem 6 (Correctness of sba_P)

- *For imperative programs P, $sba_P \supseteq \mathcal{CS}_P$.*
- *For logic programs P,*

$$sba_P(\Psi^\alpha) =_{var(\alpha)} \{\rho : \rho \models E \text{ and } E \in \mathcal{CS}_P(\alpha)\}, \text{ for all labels } \alpha.$$

$\square$

We conclude this section by noting that the uses of $\mathcal{A}$ in the definition of set based interpretation could have been removed without altering the definition of sba_P. They have been retained to emphasize that various objects in the definition are set environments. To see why they do not affect sba_P, first number the four occurrences of $\mathcal{A}$ in order so that the first occurrences appears in the interpretation of $\Psi[X \mapsto t]$, the second and third (numbered left-to-right) appear in the interpretation of $\Psi[cond]$ and the last appears in the interpretation of $[A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]$. Now, consider the second and last occurrences of $\mathcal{A}$. These occurrences ensure that the set based interpretation of an environment expression is always a set based collection of environments. However, when determining whether a set environment is a model of a constraint $\Psi \supseteq ee$, the difference between retaining these occurrences and omitting them reduces to the difference between the

$\mathcal{I}(\Psi) \supseteq \mathcal{A}(S_{ee,\mathcal{I}})$ and $\mathcal{I}(\Psi) \supseteq S_{ee,\mathcal{I}}$

where $\mathcal{S}_{ee,\mathcal{I}}$ is a set of environments dependent on ee and $\mathcal{I}$. Since $\mathcal{I}(\Psi)$ is required to be a set environment, these two formulas are equivalent.

Now consider the first and third occurrences of $\mathcal{A}$. These occurrences are applied to environment sets of the form $(\{\rho \in \mathcal{I}(\Psi) : \rho \rhd t\})$ or $(\{\rho \in \mathcal{I}(\Psi) : \rho \rhd cond\})$ to ensure that a set environment is obtained. However, the following proposition shows that if $\mathcal{I}(\Psi)$ is a set based collection of environments, then $(\{\rho \in \mathcal{I}(\Psi) : \rho \rhd t\})$ and $(\{\rho \in \mathcal{I}(\Psi) : \rho \rhd cond\})$ will also be set based collections of environments and so the application of $\mathcal{A}$ will not have any effect.

Proposition 14 *Let Θ be a set based collecting of environments and let $t_1, \ldots, t_r$ be program terms. Then $\{\rho \in \Theta : \wedge_{k=1..r} \rho \rhd t_k\}$ is a set based collection of environments.*

Proof: The proposition is established by showing that

$$\mathcal{A}(\{\rho \in \Theta : \wedge_{k=1..r}\rho \ \triangleright t_k\}) \subseteq \{\rho \in \Theta : \wedge_{k=1..r}\rho \ \triangleright t_k\}$$

Let $\rho \in \mathcal{A}(\{\rho \in \Theta : \wedge_{k=1..r}\rho \ \triangleright t_k\})$, and it remains to show that $\rho \in \{\rho \in \Theta : \wedge_{k=1..r}\rho \ \triangleright t_k\}$. From the definition of $\mathcal{A}$ it follows that there exists an environment ρ_X such that $\rho_X \in \{\rho \in \Theta : \wedge_{k=1..r}\rho \ \triangleright t_k\}$. Since each $\rho_X \in \Theta$ and Θ is set based, it follows that $\rho \in \Theta$. Hence to show that $\rho \in \{\rho \in \Theta : \wedge_{k=1..r}\rho \ \triangleright t_k\}$, it suffices to show that $\rho \ \triangleright t_k$, $k = 1..r$. The proof of this proceeds by structural induction on each t_k. The induction hypothesis is that $\rho \ \triangleright t_k$ and that either

(a) for some program variable X, $\rho_X \ \triangleright t_k$ and $\rho(t_k) = \rho_X(t_k)$, or

(b) there exist subterms $s_1, \ldots, s_n$ of t_k such that for any environment ρ', $\rho' \triangleright t_k$ implies that $\rho' \triangleright s_i$, $i = 1..n$, and $\rho'(t_k) = f(\rho'(s_1), \ldots, \rho'(s_n))$.

First suppose that t_k is a variable, say X. Then $\rho \triangleright t_k$ and $\rho(t_k) = \rho_X(t_k)$, and so the induction hypothesis holds with condition (a).

Now suppose that t_k is of the form $f(s_1, \ldots, s_n)$. Since each s_i satisfies the induction hypothesis, it follows that each $\rho(s_i)$ is defined and so $\rho \ \triangleright t_k$. Also, it is immediate that for any environment ρ', $\rho' \triangleright t_k$ implies that $\rho' \triangleright s_i$, $i = 1..n$, and $\rho'(t_k) = f(\rho'(s_1), \ldots, \rho'(s_n))$. Hence t_k satisfies condition (b).

The remaining case is where t_k is of the form $f^{-1}_{(j)}(s)$. Now, on applying the induction hypothesis to s, it follows that $\rho \triangleright s$ and s satisfies either (a) or (b). First suppose that s satisfies (a). Then $\rho(s) = \rho_X(s)$ for some program variable X, and since $\rho_X \ \triangleright f^{-1}_{(j)}(s)$, it follows that $\rho_X(s)$ must be of the form $f(\cdots)$. Hence $\rho(f^{-1}_{(j)}(s))$ is defined and is in fact equal to $\rho_X(f^{-1}_{(j)}(s))$, and so t_k satisfies case (a). Now suppose that s satisfies (b). Then there exist subterms $s_1, \ldots, s_n$ of s such that for any environment ρ', $\rho' \ \triangleright s$ implies that $\rho' \ \triangleright s_i$, $i = 1..n$, and $\rho'(s) = g(\rho'(s_1), \ldots, \rho'(s_n))$. This has a number of consequences. First, since $\rho_X \ \triangleright f^{-1}_{(j)}(s)$, it must be the case that $f = g$. Second, since $\rho \ \triangleright s$, it must be the case that $\rho \ \triangleright f^{-1}_{(j)}(s)$ is defined, and furthermore, that $\rho(f^{-1}_{(j)}(s)) = \rho(s_i)$. On applying the induction hypothesis to s_i, it is clear that t_k respectively satisfies (a) or (b) if s_i satisfies (a) or (b). $\square$

5.4 Alternative Definitions

The basic goal of set based analysis is to obtain a very simple definition of approximation based on the notion of ignoring dependencies arising from the behavior of variables. However this notion can potentially be realized in a number of different ways and the definition of set based program approximation presented in the previous section represents a choice among a number of possible definitions. We now outline the major alternatives and compare them with set based analysis. In particular, we shall argue that the set based analysis is the most natural choice, given the requirements of decidability, accuracy and simplicity.

Language Restrictions

Perhaps the simplest definition of approximation that employs the idea of ignoring inter-variable dependencies is approx_P. We have already shown that approx_P is not decidable and that it does not ignore all inter-variable dependencies. In essence, the language operations are sufficiently powerful that unbounded dependencies can be introduced even when all collections of environments are free from inter-variable dependencies. One way to address this problem is to restrict the language so that the language operations cannot by themselves introduce unbounded dependencies. This approach was used in an early version of set based analysis for imperative programs reported by Heintze and Jaffar in [23]. In essence, this paper obtains a decidable program approximation based on approx_P by restricting imperative programs in the following two ways:

(i) Assignment statements must have the form $X := f(X_1, \ldots, X_n)$ where the X_i were distinct;

(ii) Program conditions must have one of the following forms:

  (a) $X = Y$ where X and Y are program variables;
  (b) $\mathit{match}_f(X)$ where X is a program variable, or
  (c) a negation of (a) or (b).

Intuitively, these restrictions ensure that multiple occurrences of program variables cannot occur in a term. For example, a statement such as $Y :=$

$pair(X, X)$ cannot be written. This implies that, in isolation, conditions and assignments cannot introduce inter-variable dependencies. In other words, the only form of inter-variable dependency that arises is such programs is dependency between variable values in collections of environments. Since $approx_P$ ignores all such dependencies, it follows that for programs satisfying (i) and (ii), the approximation $approx_P$ is free of inter-variable dependencies. In fact, for such programs we can show that $approx_P = sba_P$. Hence, the set based approximation of imperative programs described in this thesis can be viewed as a conservative extension of the approximations defined in [23].

The main drawback of using this subclass of imperative programs defined by (i) and (ii) is that it is unreasonably restrictive. Although any imperative program P can be transformed (by "unfolding" complex assignments and conditions) into a semantically equivalent program P' that satisfies (i) and (ii), the transformation from P to P' forgets much of the structure of P and this has a detrimental effect on the accuracy of the analysis. In particular, we can show that $approx_{P'} \supseteq sba_P$. Despite this drawback, this approach does have one appealing property. In essence the approximation $approx_P$ corresponds to a Hoare-style reasoning about an imperative program using assertions that do not express information about inter-variable dependencies. Specifically, consider an assertion language consisting of formulas of the form $\Phi_1 \wedge \cdots \wedge \Phi_n$ where each Φ_i is a formula containing at most one free variable. Let Φ^μ be the strongest assertion that can be proved for the point μ and let $\rho \models \Psi$ denote that ρ satisfies the formula Φ. Then $\rho \models \Phi^\mu$ iff $\rho \in approx_P(\Psi^\mu)$.

We finally note the imperative language used in this thesis employs a moderate language restriction. Recall from Chapter 3 that atomic program conditions of the form $s = t$ are such that s and t are constructed from program variables and projection symbols. A more general language could be defined in which s and t are arbitrary program terms. However, sba_P for this language would not be decidable. Intuitively this is because the combination of function symbols and projection symbols allows a form of unbounded dependency to be introduced. To illustrate the reason for this, let f and g respectively be unary and binary function symbols, let $cond$ be the program condition

$$g(f^{-1}_{(1)}(g^{-1}_{(1)}(X)), f^{-1}_{(1)}(g^{-1}_{(2)}(X))) = Y$$

and consider the set based interpretation of the environment expression $\Psi[cond]$. Suppose that $\mathcal{I}(\Psi)$ is the set environment that maps X into the set of all values and maps Y into the singleton set $\{g(a,a)\}$. Then, using the definition of set based interpretation, $\mathcal{I}(\Psi[cond])$ is the set environment that maps X into $\{g(f(a),f(a))\}$. This example can be easily modified to show that sets of the form $\{g(f^n(a),f^n(a)) : n \geq 0\}$ can be formed in sba_P, and also to show that sba_P for this extended language is undecidable (for a related discussion, see Section 7.6 page 197).

Although our restriction on atomic program conditions of the form $s = t$ is very significant from a decidability point of view, it is inconsequential from a programming point of view because complex conditions such as the condition $g(f^{-1}_{(1)}(g^{-1}_{(1)}(X)), f^{-1}_{(1)}(g^{-1}_{(2)}(X))) = Y$ are rarely written in programs. We note that the restriction could be substantially relaxed to admit conditions $s = t$ where s and t do not contain combinations of function and projection symbols. Also note that it is easy to translate from an arbitrary condition $s = t$ into an equivalent condition that is in our language. For example $g(f^{-1}_{(1)}(g^{-1}_{(1)}(X)), f^{-1}_{(1)}(g^{-1}_{(2)}(X))) = Y$ could be translated into $f^{-1}_{(1)}(g^{-1}_{(1)}(X)) = g^{-1}_{(1)}(Y) \wedge f^{-1}_{(1)}(g^{-1}_{(2)}(X)) = g^{-1}_{(2)}(Y)$. Moreover, such a translation results in little loss of information in practice.

More Direct Use of Set Environments

We now present an alternative interpretation of environment constraints that employs set environments in a very direct manner. Consider an environment expression of the form $([A_1 \in B_1.\Psi_1, \ldots, A_n \in B_n.\Psi_n]$. The standard interpretation of such an expression under an interpretation $\mathcal{I}$ is:

$$\{\rho : \text{ for each } i,\ \rho(A_i) = \rho_i(B_i) \text{ for some } \rho_i \in \mathcal{I}(\Psi_i)\}.$$

A very natural way to modify this interpretation to use set environments is

$$\bigcup\{\varrho : \text{ for each } i,\ \varrho(A_i) = \varrho_i(B_i) \text{ for some } \varrho_i \subseteq \mathcal{I}(\Psi_i)\}.$$

where $\cup$ denotes the pointwise union of a set of set environments and $\subseteq$ denote subset on set environments (again defined pointwise). Such an approach can be extended in a straightforward manner to the other kinds of environment constraints. Moreover, it is arguably simpler than the definition of set based interpretation, and it is easy to verify that it is more accurate.

1. $p(f(a,a))$.
2. $p(V) \leftarrow q(V, f(W,W), f(s(W), s(W)))$.
3. $q(X,Y,X) \leftarrow p(Y)$.

$$\begin{array}{lcl}
\Psi_1 & \supseteq & [\,] \\
\Psi_2 & \supseteq & [q(V, f(W,W), f(s(W), s(W))) \in q(X,Y,X).\Psi^3] \\
\Psi_3 & \supseteq & [p(Y) \in p(f(a,a)).\Psi^1] \\
\Psi_3 & \supseteq & [p(Y) \in p(V).\Psi^2]
\end{array}$$

Figure 5.2: **Undecidability of Modified Set Based Interpretation**

Unfortunately it leads to an undecidable notion of program approximation. In essence, this is because inter-variable dependencies may arise and these lead to unbounded dependencies. To see this, consider Figure 5.2, which contains a logic program and its bottom-up environment constraints. Using the alternative interpretation just outlined, the least interpretation that is a model of these constraints is

$$\begin{array}{lcl}
\Psi^1 & \mapsto & \{\text{all environments}\} \\
\Psi^2 & \mapsto & \Big[V \mapsto \{f(s^n(a), s^n(a)) : n \geq 0\}, W \mapsto \{f(s^n(a), s^n(a)) : n \geq 0\}\Big] \\
\Psi^3 & \mapsto & \Big[X \mapsto \{\text{all values}\}, Y \mapsto \{f(s^n(a), s^n(a)) : n \geq 0\}\Big]
\end{array}$$

It is easy to modify this example to prove that the program approximation arising from this interpretation of environments expressions is undecidable.

Ignoring All Dependencies

Set based analysis ignores all inter-variable dependencies, but it does retain certain notions of dependency that are not related to the treatment of variables. For example, consider Figure 5.3, which shows an imperative program along with sba_P for this program at some selected program points. The set of values for X specified by sba_P at point $\uparrow 3$ exhibits inter-argument dependencies in the sense that whenever the first argument of *cons* is 1, the second argument is $cons(2, nil)$, and whenever the first argument is 2, the second argument is *nil*. If inter-argument dependencies are ignored, then this set would be enlarged to include $cons(2, cons(2, nil))$ and $cons(1, nil)$. An im-

1. $X := cons(1, cons(2, nil))$;
2. **while** $X \neq nil$ **do**
3. $\quad X := cdr(X)$;

point	sba_P at selected points
$\downarrow 1, \uparrow 2$	$X \mapsto \{cons(1, cons(2, nil))\}$
$\uparrow 3$	$X \mapsto \left\{ \begin{array}{c} cons(1, cons(2, nil)) \\ cons(2, nil) \end{array} \right\}$
$\downarrow 3$	$X \mapsto \left\{ \begin{array}{c} cons(2, nil) \\ nil \end{array} \right\}$

Figure 5.3: **Inter-Argument Dependency Example**

portant difference between these two kinds of dependency is that ignoring inter-variable dependencies is sufficient for obtaining decidable program approximations – it is not necessary to ignore inter-argument dependencies. Intuitively, this is because, given a program P, the inter-argument dependencies that are present in $approx_P$ are of a bounded nature in the sense that they are due to the (finite) collection of program terms that appear in P. In other words, no essentially new dependencies can be generated. In contrast, dependencies introduced through variables are potentially infinite, such as those introduced through a statement such as $X := pair(X, X)$.

Several approaches to program approximation based on ignoring inter-argument dependencies have been proposed in the literature (see for example [48, 68]) and we shall consider these in greater detail in Section 5.6. Since ignoring inter-argument dependencies implies that inter-variable dependencies are ignored, it follows that such approaches are strictly less accurate than set based approximation.

5.5 Examples

We now give some examples of the set based approximations. First we present some imperative program examples. Figure 5.4 contains the environment constraints and set based approximation of the program consisting of the single statement $X := pair(X, X)$. Figure 5.5 contains the constraints and set based approximation of an imperative program for computing the last element of a list. Note that in the set based approximation of this program, the set of values for Y at the end of the program is $\{a, b, c\}$. This is clearly an approximation of the run-time behavior of the program since the only possible value for Y at the end of program execution is c.

The remaining two figures give examples involving logic programs. Figure 5.6 presents the bottom-up set based approximations of two logic programs and illustrates that inter-variable dependencies are ignored in set based approximations, but inter-argument dependencies are not ignored. Finally, 5.7 presents the top-down set based approximation of a logic program that computes the last element of a list. Note that the set based approximation this program is exact in the sense that the set assigned to V at point 2 is $\{b\}$ and this is precisely the possible values for Y at this point. Intuitively, this is because the set of possible "calls" (this is given by the union of the sets $loop(a.b.nil, V).\Psi^1$ and $loop(L, Y).\Psi^4$) is computed exactly, and the only possible way that $loop(W.nil, W)$ can match this set is with $W = b$. Moreover the rule $loop(X.L, Y) \leftarrow loop(L, Y)$ cannot generate any new answers for the second argument to $loop$. Note that the bottom-up set based analysis of this program would not be exact.

$$\begin{array}{lll} \Psi^{\uparrow 1} & \supseteq & \top \\ \Psi^{\downarrow 1} & \supseteq & \Psi^{\uparrow 1}[X \mapsto pair(X,X)] \end{array} \qquad \begin{array}{lll} \Psi^{\uparrow 1} & \mapsto & \left[X \mapsto \{\text{all values}\}\right] \\ \Psi^{\downarrow 1} & \mapsto & \left[X \mapsto \left\{ pair(v_1, v_2) : \begin{array}{c} v_1 \text{ and } v_2 \\ \text{are values} \end{array} \right\}\right] \end{array}$$

Figure 5.4: **Set Based Approximation of** $X := pair(X, X)$

1. $L := cons(a, cons(b, nil));$
2. $X := c;$
3. **while** $(match_{cons}(L))$ **do**
4. $\quad X := car(L);$
5. $\quad L := cdr(L);$

$$\begin{array}{lll}
\Psi^{\uparrow 1} & \supseteq & \top \\
\Psi^{\downarrow 1} & \supseteq & \Psi^{\uparrow 1}[L \mapsto a.b.nil] \\
\Psi^{\uparrow 2} & \supseteq & \Psi^{\downarrow 1} \\
\Psi^{\downarrow 2} & \supseteq & \Psi^{\uparrow 2}[X \mapsto c] \\
\Psi^{\uparrow 3} & \supseteq & \Psi^{\downarrow 2} \\
\Psi^{\uparrow 3} & \supseteq & \Psi^{\downarrow 5} \\
\Psi^{\uparrow 4} & \supseteq & \Psi^{\uparrow 3}[match_{cons}(L)] \\
\Psi^{\downarrow 4} & \supseteq & \Psi^{\uparrow 4}[X \mapsto cons^{-1}_{(1)}(L)] \\
\Psi^{\uparrow 5} & \supseteq & \Psi^{\downarrow 4} \\
\Psi^{\downarrow 5} & \supseteq & \Psi^{\uparrow 5}[L \mapsto cons^{-1}_{(2)}(L)] \\
\Psi^{\downarrow 3} & \supseteq & \Psi^{\uparrow 3}[\neg match_{cons}(L)]
\end{array}$$

program point	set environment
$\uparrow 1$	$[L \mapsto \{\text{all values}\},\ X \mapsto \{\text{all values}\}]$
$\downarrow 1$, $\uparrow 2$	$[L \mapsto \{a.b.nil\},\ X \mapsto \{\text{all values}\}]$
$\downarrow 2$	$[L \mapsto \{a.b.nil\},\ X \mapsto \{c\}]$
$\uparrow 3$	$[L \mapsto \{a.b.nil, b.nil, nil\},\ X \mapsto \{a, b, c\}]$
$\uparrow 4$	$[L \mapsto \{a.b.nil, b.nil\},\ X \mapsto \{a, b, c\}]$
$\downarrow 4$, $\uparrow 5$	$[L \mapsto \{a.b.nil, b.nil\},\ X \mapsto \{b, c\}]$
$\downarrow 5$	$[L \mapsto \{b.nil, nil\},\ X \mapsto \{b, c\}]$
$\downarrow 3$	$[L \mapsto \{nil\},\ X \mapsto \{a, b, c\}]$

Figure 5.5: **Set Based Approximation of Program 2**

1. $\leftarrow q(X).$
2 $q(Y) \leftarrow p(Y).$
3. $p(f(a,b)).$
4. $p(f(c,d)).$

$$\Psi^1 \supseteq [q(X) \in q(Y).\Psi^2]$$
$$\Psi^2 \supseteq [p(Y) \in p(f(a,b)).\Psi^3]$$
$$\Psi^2 \supseteq [p(Y) \in p(f(c,d)).\Psi^4]$$
$$\Psi^3 \supseteq [\,]$$
$$\Psi^4 \supseteq [\,]$$

$$\Psi^1 \mapsto [X \mapsto \{f(a,b), f(c,d)\}]$$
$$\Psi^2 \mapsto [Y \mapsto \{f(a,b), f(c,d)\}]$$
$$\Psi^3 \mapsto \{\text{all environments}\}$$
$$\Psi^4 \mapsto \{\text{all environments}\}$$

1. $\leftarrow q(X).$
2 $q(f(Y_1,Y_2)) \leftarrow p(f(Y_1,Y_2)).$
3. $p(f(a,b)).$
4. $p(f(c,d)).$

$$\Psi^1 \supseteq [q(X) \in q(f(Y_1,Y_2)).\Psi^2]$$
$$\Psi^2 \supseteq [p(f(Y_1,Y_2)) \in p(f(a,b)).\Psi^3]$$
$$\Psi^2 \supseteq [p(f(Y_1,Y_2)) \in p(f(c,d)).\Psi^4]$$
$$\Psi^3 \supseteq [\,]$$
$$\Psi^4 \supseteq [\,]$$

$$\Psi^1 \mapsto \left[X \mapsto \left\{ \begin{array}{c} f(a,b), f(c,d), \\ f(a,d), f(c,b) \end{array} \right\}\right]$$
$$\Psi^2 \mapsto [Y_1 \mapsto \{a,c\},\ Y_2 \mapsto \{b,d\}]$$
$$\Psi^3 \mapsto \{\text{all environments}\}$$
$$\Psi^4 \mapsto \{\text{all environments}\}$$

Figure 5.6: **Bottom-Up Set Based Approx. of Two Logic Programs**

2. $\leftarrow loop(a.b.nil, V)^1.$
3. $loop(W.nil, W).$
5. $loop(X.L, Y) \leftarrow loop(L, Y)^4.$

$$
\begin{aligned}
\Psi^1 &\supseteq [\,] \\
\Psi^2 &\supseteq [loop(a.b.nil, V) \in loop(W.nil, W).\Psi^3] \\
\Psi^2 &\supseteq [loop(a.b.nil, V) \in loop(X.L, Y).\Psi^5] \\
\Psi^3 &\supseteq [loop(W.nil, W) \in loop(a.b.nil, V).\Psi^1] \\
\Psi^3 &\supseteq [loop(W.nil, W) \in loop(L, Y).\Psi^4] \\
\Psi^4 &\supseteq [loop(X.L, Y) \in loop(a.b.nil, V).\Psi^1] \\
\Psi^4 &\supseteq [loop(X.L, Y) \in loop(L, Y).\Psi^4] \\
\Psi^5 &\supseteq [loop(X.L, Y) \in loop(a.b.nil, V).\Psi^1,\ loop(L, Y) \in loop(W.nil, W).\Psi^3] \\
\Psi^5 &\supseteq [loop(X.L, Y) \in loop(a.b.nil, V).\Psi^1,\ loop(L, Y) \in loop(X.L, Y).\Psi^5] \\
\Psi^5 &\supseteq [loop(X.L, Y) \in loop(L, Y).\Psi^4,\ loop(L, Y) \in loop(W.nil, W).\Psi^3] \\
\Psi^5 &\supseteq [loop(X.L, Y) \in loop(L, Y).\Psi^4,\ loop(L, Y) \in loop(X.L, Y).\Psi^5]
\end{aligned}
$$

$$
\begin{aligned}
&\Psi^1 \mapsto [V \mapsto \{\text{all values}\}] \\
&\Psi^2 \mapsto [V \mapsto \{b\}] \\
&\Psi^3 \mapsto [W \mapsto \{b\}] \\
&\Psi^4 \mapsto [X \mapsto \{a, b\},\ Y \mapsto \{\text{all values}\},\ L \mapsto \{b.nil, nil\}] \\
&\Psi^5 \mapsto [X \mapsto \{a, b\},\ Y \mapsto \{b\},\ L \mapsto \{b.nil, nil\}]
\end{aligned}
$$

Figure 5.7: **Top-Down Set Based Approximation of Program 10**

5.6 Related Work

In the literature, the notion of program approximation appears both in the areas of types and program analysis. For the purposes of the following discussion, the distinction between these two areas is somewhat artificial, and it is more useful to classify these works according to the underlying approach used to obtain program approximation. Broadly speaking, two approaches have been used, one based on abstract interpretation and the other based on the use of closure operations and constraints. We shall refer to these as abstract interpretation and non-abstract interpretation approaches, respectively. We note that this terminology is somewhat loose because if one takes a very broad view of abstract interpretation, then many of the non-abstract interpretation approaches can be viewed as abstract interpretation. The essential difference between the two approaches is that the abstract interpretation approach employs (a variant of) an iterative fixed point computation to compute the program approximation, whereas in the non-abstract interpretation approach the program approximation cannot be computed by an iterative fixed point computation (even with the aid of widening and narrowing), and so very different computation techniques must be employed.

Abstract Interpretation Approaches

In these approaches, program approximation is defined by specifying a collection of approximate values in the place of the exact values. This induces an approximate semantic function, and the program approximation is typically the least fixed point of this function. Algorithms for computing such approximations usually take the form of some kind of iterative fixed point computation. Importantly, the approximate values are chosen in such a way that such an iterative fixed point computation is guaranteed to terminate.

In logic programs, this approach has been widely used in type inference [38, 40, 67], sharing analysis [27, 51], instantiation analysis [45] and in various combinations of these analyses [11, 15]. General frameworks for abstract interpretation of logic programs have also been developed by Bruynooghe [10] and also by Marriot, Sondergaard and Jones [44, 60].

Similarly, the idea of using a collection of approximate values to reason about programs appears, often somewhat implicitly, in most of the work on analysis of imperative programs. The use of approximate values is made more explicit in a number of the more formal accounts of this approach. Early works in this area include papers by Sintzoff [58], Kildall [39] and Wegbreit [66]. These ideas were further developed by Cousot and Cousot [13, 14].

In functional programming, abstract interpretation has been used for a variety of analyses including strictness analysis, sharing analysis, and binding time analysis (see, for example, the collection of papers [1]). There are also connections with type systems, particularly those involving subtypes. In such systems, the starting point is some given finite set of base types. This set is then closed under a small finite number of type constructors, usually including the arrow type constructor. The use of a finite predefined set of base types, and the need to avoid infinite ascending chains of types, is in spirit similar to the use of approximate values in abstract interpretation. In fact a number of type inference problems can be usefully viewed as abstract interpretation (for example, the refinement types of Freeman and Pfenning [17]).

Although there is much diversity in the above works for logic, imperative and functional languages, one unifying factor is the use of a collection of approximate values that is finite or, more generally, satisfies some kind of finite ascending chain condition. Moreover, this condition is used to guarantee termination of the analysis.

In terms of accuracy, the program approximations defined by this kind of approach are not directly comparable to set based approximations. For some programs, abstract interpretation is more accurate. As an example, if an imperative or logic program contains only constants (function symbols of arity 0), then an abstract interpretation approach can be used to compute the exact collecting semantics. This is because the domain of values is finite, and so the collection of abstract value can be chosen to be all possible sets of program values. However, the set based approximation of such a program is, in general, an approximation to its meaning. Conversely, there exist classes of programs for which sba_P and $\mathcal{CS}_P$ coincide, but such that no abstract interpretation algorithm can compute $\mathcal{CS}_P$ for each program in the class. For example, consider a program in which all function symbols (and if applicable, predicate symbols) have arity 0 or 1. For such programs $sba_P = \mathcal{CS}_P$. However no abstract interpretation based approach can be used to compute exactly $\mathcal{CS}_P$ for each program in this class. This is essentially because any regular language is definable by a monadic program. Hence, if an abstract interpretation is to be exact on all monadic programs, the collection of abstract values used must be expressive enough to represent all regular languages, and this leads to termination problems. A more detailed account of this argument is presented in Appendix II.

In summary, the program approximations that arise in abstract interpretation work can capture some information about inter-variable dependencies, but they must embody other forms of approximation (to ensure that the iterative fixed point computation terminates). On the other hand, set based approximations ignore all information about inter-variable dependencies, but make no other approximations.

For efficiency reasons, many abstract interpretation approaches ignore all inter-variable dependencies. In this case, set based analysis is more accurate than abstract interpretation. Moreover, abstract interpretation that ignores inter-variable dependencies can be used to provide an alternative definition of set based analysis as follows. Consider the class of all abstract interpretations that ignore inter-variable dependencies. In essence, each abstract interpretation in this class is defined by an abstract domain that consists of a collection of descriptions for sets of program values. In some sense, the choice of abstract domain in each case is somewhat *ad hoc*, since the accuracy of the analysis can always be improved by adding more descriptions to the abstract domain. In contrast, set based analysis is *optimal* in the sense that it corresponds to an inter-variable dependencies free abstract interpretation over the abstract domain consisting of *all* possible sets of program values. Clearly, traditional iterative fixed point techniques cannot be applied to compute over this domain, and hence the set based analysis algorithm must use very different techniques. We note that, given a program P, the output of the set based analysis algorithm essentially defines a finite domain that is optimal for the inter-variable dependency free abstract interpretation of P. In other words, for every program P, the set based analysis algorithm synthesizes a finite abstract domain $\mathcal{D}_P$ that is at least as good as any (finite or infinite) abstract domain for analyzing P, and moreover corresponds to the set based analysis of P. The optimal domain $\mathcal{D}_P$ is clearly different for different programs.

$p(f(a,b)).$	$\Upsilon_p \subseteq p(f(a,b)) \uplus p(f(b,a))$
$p(f(b,a)).$	$\Upsilon_q \subseteq q(X)$
$q(X) \leftarrow p(f(X,X)).$	$p(X) \subseteq \Upsilon_p$

Figure 5.8: **Program 11 and Its Set Formula**

Non Abstract Interpretation Approaches

Instead of using a collection of approximate values to obtain a decidable program approximation, these approaches approximate inter-variable or inter-argument dependencies. They are usually based on the use of set formulas (constructed from a program) or closure operators. Since there are close connections between these approaches and set based analysis, we shall consider this part of the literature in considerable detail.

We begin by discussing approaches to the approximation of logic programs. One of the early works in this area was by Mishra [48] and involved the use of set formulas to approximate a program's success set. Specifically, a set formula was constructed from a given program P, and it was shown that the greatest model of this formula was a superset of the success set of P. We now illustrate the construction of these formulas. Figure 5.8 contains Program 11 and its corresponding set formula. In this formula, the variables Υ_p and Υ_q are intended to be the subsets of atoms in the success set of Program 11 that involve the predicates p and q respectively. The variable X denotes a *set* of values and is intended to capture the values of the program variable X. The operator $\uplus$ is similar to set union except that it performs a tuple closure and this serves to ignore inter-argument dependencies. For example, $\{f(a,b)\} \uplus \{f(b,a)\}$ is defined to be $\{f(a,a), f(a,b), f(b,a), f(b,b)\}$. More formally, if S_1 and S_2 are two sets of values, then $S_1 \uplus S_2$ is $(S_1 \cup S_2)^\star$ where $\star$ is defined to be the following closure operator:

$$S^\star \stackrel{\text{def}}{=} \{c : c \text{ is a constant in } S\} \cup \bigcup_{f\in\Sigma} f\left((f^{-1}_{(1)}(S))^\star, \ldots, (f^{-1}_{(arity(f))}(S))^\star\right)$$

where $f(S_1, \ldots, S_n)$ denotes the set $\{f(s_1, \ldots, s_n) : s_i \in S_i\}$ and $f^{-1}_{(i)}(S)$ denotes the set $\{s_i : f(s_1, \ldots, s_n) \in S\}$. An interpretation $\mathcal{I}$ of a set formula is defined by specifying a set of ground atoms for each variable of the form Υ_p, and a set of values for each program variable. Interpretations are ordered pointwise as follow: $\mathcal{I} \supseteq \mathcal{I}'$ if, for each variable, the set specified by $\mathcal{I}$ is larger than or equal to the one specified by $\mathcal{I}'$. An interpretation that satisfies the formula is called a model. For example, the greatest model of the set formula in Figure 5.8 is

$$\begin{array}{ccl} \Upsilon_p & \mapsto & \{p(f(a,a)), p(f(a,b)), p(f(b,a)), p(f(b,b))\} \\ \Upsilon_q & \mapsto & \{q(a), q(b)\} \\ X & \mapsto & \{a,b\} \end{array}$$

Using the same kind of approximation, Yardeni and Shapiro [68] defined a notion of program approximation based on the immediate consequence operator T_{P}. Recall from Section 4.2, page 63, that the immediate consequence operator T_{P} is defined by

$$T_{\mathrm{P}}(S) = \left\{ \rho(A_0) \ : \ \begin{array}{c} A_0 \leftarrow A_1, \ldots, A_n \text{ is a rule in } P \\ \text{and } \rho(A_i) \in S, \ i = 1..n \end{array} \right\}$$

and that the least fixed point of this function, denoted $\mathit{lfp}(T_{\mathrm{P}})$ can be used to defined a semantics of logic programs. Yardeni and Shapiro modified this definition using the $\star$ operator. Specifically, an approximate immediate consequence operator Y_{P} was defined by $Y_{\mathrm{P}}(S) = (T_{\mathrm{P}}(S))^{\star}$. This gives rise to an approximate program semantics $\mathit{lfp}(Y_{\mathrm{P}})$. Clearly $Y_{\mathrm{P}}(S) \supseteq T_{\mathrm{P}}(S)$ for each S, and it follows that $\mathit{lfp}(Y_{\mathrm{P}})$ is a conservative approximation of P in the sense that $\mathit{lfp}(Y_{\mathrm{P}}) \supseteq \mathit{lfp}(T_{\mathrm{P}})$.

In [25], Heintze and Jaffar showed that the approximations defined by the set formulas of Mishra and the Y_{P} operator of Yardeni and Shapiro are very closely related. In essence, the greatest model of the set formulas corresponds to the greatest fixed point of Y_{P}. [25] also shows how the set formulas can be re-engineered so that the correspondence becomes exact. Specifically, it is shown how, from a program P, set formulas can be constructed such that a model of the set formulas is a fixed point of Y_{P} and vice-versa. (Strictly speaking, this correspondence may not be exact for a small class of degenerate programs.)

In summary, the set formulas and the Y_{P} operator both approximate programs by ignoring dependencies between arguments of function symbols. The advantage of using set formulas is that they provide a natural starting point for development of algorithms. Partial algorithms were reported in [48]. The advantage of using T_{P}-like operators is that they provide a closer connection with standard notions of logic program semantics, and hence give better insight into the nature of the approximation.

The notion of closure embodied in $\star$ is in some sense very extreme: it forces all dependencies to be ignored. Moreover, it behaves in an unbounded manner (see the recursive case in the definition of $\star$), and this appears to introduce substantial complexity. It has yet to be shown that approximations using $\star$, such as $\mathit{lfp}(Y_{\mathrm{P}})$, are decidable. One reason for the extra complexity is that the usual distributive laws do not hold when $\cup$ is replaced by $\uplus$. That is, $(S_1 \uplus S_2) \cap S_3$ does not in general equal $(S_1 \cap S_3) \uplus (S_2 \cap S_3)$.

In [21], Heintze and Jaffar proposed a more accurate approximate consequence operator, $\mathcal{T}_{\mathrm{P}}$. Instead of ignoring all dependencies, $\mathcal{T}_{\mathrm{P}}$ approximates programs by ignoring inter-variable dependencies. In essence, this work was an early version of the bottom-up set based analysis of logic programs. To define $\mathcal{T}_{\mathrm{P}}$, recall the definition of $\mathcal{A}$, which maps a collection of environments Θ into the smallest set environment that contains Θ. Specifically, $\mathcal{A}(\Theta)$ is the set environment that maps each variable X into $\{\theta(X) : \theta \in \Theta\}$. Now, we have already observed how set environments can be treated as mappings from terms into sets of values. For example, if ϱ is the set environment $\{[X \mapsto \{a, c\}, Y \mapsto \{b, d\}]\}$ then $\varrho(f(X, Y))$ denotes the set $\{f(a, b),\ f(a, d),\ f(c, b),\ f(c, d)\}$. Using these definitions, $\mathcal{T}_{\mathrm{P}}$ can be defined as follows:

$$\mathcal{T}_{\mathrm{P}}(I) \stackrel{\mathrm{def}}{=} \left\{ a \in \varrho(A_0) : \begin{array}{l} A_0 \leftarrow A_1, \ldots, A_n \in P, \text{ and} \\ \varrho = \mathcal{A}\left(\{\rho : \rho(A_1) \in I, \ldots, \rho(A_n) \in I\} \right) \end{array} \right\}.$$

where I is a set of ground atoms. Intuitively, $\mathcal{T}_{\mathrm{P}}$ first collects together the environments for a rule that instantiate the body atoms into elements in I. Using this set of environments, it collects together the possible values that each variable may be instantiated to, ignoring relationships between these variables. A set environment is then defined as the mapping from each variable into the corresponding collected set of values, and finally, this set environment is applied to the head of the rule.

To illustrate the difference between Y_{P} and $\mathcal{T}_{\mathrm{P}}$, consider Figure 5.9, which gives an example logic program and the corresponding least fixed points of Y_{P}, $\mathcal{T}_{\mathrm{P}}$ and T_{P}. For simplicity, only the subsets of atoms with predicate q are given; we abbreviate these sets by $\mathit{lfp}(Y_{\mathrm{P}})|_q$, $\mathit{lfp}(\mathcal{T}_{\mathrm{P}})|_q$ and $\mathit{lfp}(T_{\mathrm{P}})|_q$ respectively. For Y_{P}, the approximation is at the level of arguments, and this is reflected by the presence of subterms such as $f(a, d)$ and $f(c, b)$ in $\mathit{lfp}(Y_{\mathrm{P}})|_q$. In contrast, $\mathit{lfp}(\mathcal{T}_{\mathrm{P}})$ does not contain such elements and is strictly smaller than $\mathit{lfp}(Y_{\mathrm{P}})$. $\mathit{lfp}(\mathcal{T}_{\mathrm{P}})$ does however contain elements that do not appear in $\mathit{lfp}(T_{\mathrm{P}})$. This relationship holds in general. That is, for all programs P,

$p(f(a, b), a).$
$p(f(c, d), b).$
$q(X, Y) \leftarrow p(X, Y).$

$lfp(Y_{\mathrm{P}})\vert_q$	$lfp(\mathcal{T}_{\mathrm{P}})\vert_q$	$lfp(T_{\mathrm{P}})\vert_q$
q(f(a, b), a)	q(f(a, b), a)	q(f(a, b), a)
q(f(a, d), a)		
q(f(c, d), a)	q(f(c, d), a)	
q(f(c, b), a)		
q(f(c, d), b)	q(f(c, d), b)	q(f(c, d), b)
q(f(a, d), b)		
q(f(a, b), b)	q(f(a, b), b)	
q(f(c, b), b)		

Figure 5.9: **Differences between T_{P}, $\mathcal{T}_{\mathrm{P}}$ and Y_{P}**

$lfp(T_P) \subseteq lfp(\mathcal{T}_{\mathrm{P}}) \subseteq lfp(Y_P)$. This just reflects the fact that approximations defined by ignoring inter-variable dependencies are more accurate than those defined by ignoring all dependencies.

In [25], Heintze and Jaffar defined a further operator whose accuracy is between that of $\mathcal{T}_{\mathrm{P}}$ and Y_{P}. This operator, called Z_{P}, differs from $\mathcal{T}_{\mathrm{P}}$ in that it treats each variable occurrence separately. Specifically, an *occurrence based environment* is a mapping from pairs (X, i) into values, where X is a variable and i is an integer indicating a variable occurrence. We adopt the convention that occurrences of variables are labeled left to right. For example, applying $\{(X,1)\mapsto a,\ (X,2)\mapsto b,\ (X,3)\mapsto c,\ (Y,1)\mapsto d,\ (Y,2)\mapsto e\}$, to the sequence of atoms $p(X),\ q(X,Y)$ yields $p(a),\ q(b,d)$. The $\mathcal{A}$ approximation operator can now be adapted to map collections of occurrence based environments into a set environment as follows: $\mathcal{A}(\Theta)$ maps each variable X into

$$\{v : \text{ for all } i, \text{ there exists } \rho \in \Theta \text{ s.t. } \rho(X, i) = v\}$$

The operator Z_P can now be defined similarly to $\mathcal{T}_{\mathrm{P}}$, except that environments are replaced by occurrence-based environments.

$$Z_P(I) \stackrel{\text{def}}{=} \left\{ a \in \varrho(A_0) : \begin{array}{r} A_0 \leftarrow A_1, \ldots, A_n \in P, \text{ and} \\ \varrho = \mathcal{A}(\{\rho : \rho(A_i) \in I, 1 \leq i \leq n\}) \end{array} \right\}$$

The variable ϱ ranges over set environments and ρ ranges over occurrence based environments. In general Z_{P} is less accurate than $\mathcal{T}_{\mathrm{P}}$ (because it ignores dependencies between different occurrences of the same variable) but more accurate than Y_{P} (because it does not ignore all inter-argument

$$
\begin{array}{l}
p(f(a,b)).\\
p(f(b,a)).\\
r(X) \leftarrow p(f(X,X)).\\
s(f(Y,Z)) \leftarrow p(f(Y,Z)).
\end{array}
$$

$\mathit{lfp}(T_{\mathrm{P}})$	$\left\{ \begin{array}{c} p(f(a,b)),\ p(f(b,a)), \\ s(f(a,b)),\ s(f(b,a)) \end{array} \right\}$
$\mathit{lfp}(\mathcal{T}_{\mathrm{P}})$	$\left\{ \begin{array}{c} p(f(a,b)),\ p(f(b,a)), \\ s(f(a,a)),\ s(f(a,b)),\ s(f(b,a)),\ s(f(b,b)) \end{array} \right\}$
$\mathit{lfp}(Z_{\mathrm{P}})$	$\left\{ \begin{array}{c} p(f(a,a)),\ p(f(a,b)), \\ r(a),\ r(b) \\ s(f(a,a)),\ s(f(a,b)),\ s(f(b,a)),\ s(f(b,b)) \end{array} \right\}$
$\mathit{lfp}(Y_{\mathrm{P}})$	$\left\{ \begin{array}{c} p(f(a,a)),\ p(f(a,b)),\ p(f(b,a)),\ p(f(b,b)) \\ r(a),\ r(b) \\ s(f(a,a)),\ s(f(a,b)),\ s(f(b,a)),\ s(f(b,b)) \end{array} \right\}$

Figure 5.10: **Differences between** T_{P}, $\mathcal{T}_{\mathrm{P}}$, Z_{P} and Y_{P}

dependencies). Figure 5.10 illustrates the difference between T_{P}, $\mathcal{T}_{\mathrm{P}}$, Z_{P} and Y_{P}.

We now compare Y_{P}, Z_{P} and $\mathcal{T}_{\mathrm{P}}$ to set based analysis. As has already been mentioned, the work on $\mathcal{T}_{\mathrm{P}}$ was essentially an early version of bottom-up set based analysis for logic programs. Specifically, given a logic program P, $\mathit{lfp}(\mathcal{T}_{\mathrm{P}})$ corresponds exactly to the least set based model of the bottom-up environment constraints of P. It follows that set based analysis is strictly more accurate than the Y_{P} and Z_{P} approaches to program approximation.

We now sketch the proof of the equivalence of $\mathit{lfp}(\mathcal{T}_{\mathrm{P}})$ and the least set based model of bottom-up $\mathcal{EC}_P$. First, the two approximations must be put into the same form, since $\mathcal{T}_{\mathrm{P}}$ defines an approximation to the set of successful ground goals for P, whereas the least set model of bottom-up $\mathcal{EC}_P$ associates a set environment to each program rule. However, $\mathcal{T}_{\mathrm{P}}$ can be thought of as associating collections of environments to each program rule in a natural way. To see how this may be done, first recall the definition of $\mathcal{T}_{\mathrm{P}}$.

$$\mathcal{T}_{\mathrm{P}}(I) \stackrel{\text{def}}{=} \left\{ a \in \varrho(A_0) : \begin{array}{l} A_0 \leftarrow A_1, \ldots, A_n \in P, \text{ and} \\ \varrho = \mathcal{A}(\{\rho : \rho(A_i) \in I, \quad 1 \leq i \leq n\}) \end{array} \right\}.$$

Implicit in this definition is the computation of a set environment corresponding to each program rule. To make this explicit, consider replacing the set I of ground atoms, with a set of pairs (B, ϱ), where B is the head of a program rule and ϱ is a set environment. Then $\mathcal{T}_{\mathrm{P}}$ can be defined by

$$\mathcal{T}_{\mathrm{P}}(I) \stackrel{\text{def}}{=} \left\{ (A_0, \mathcal{A}(\Theta)) : \begin{array}{c} A_0 \leftarrow A_1, \ldots, A_n \in P \quad \text{and} \\ \Theta = \left\{ \rho : \rho(A_i) \in \bigcup\limits_{(B,\varrho) \in I} \varrho(B), \quad 1 \leq i \leq n \right\} \end{array} \right\}$$

where the least fixed point of this alternative function defines a collection of pairs (B, ϱ) such that taking the union of the sets $\varrho(B)$ over all these pairs recovers the least fixed point of the original $\mathcal{T}_{\mathrm{P}}$ definition.

The least fixed point of this definition is just the least solution of the equation $\mathcal{T}_{\mathrm{P}}(I) = I$, and so the least fixed point can be characterized as the least solution of

$$I \;=\; \left\{ (A_0, \mathcal{A}(\Theta)) : \begin{array}{c} A_0 \leftarrow A_1, \ldots, A_n \in P \quad \text{and} \\ \Theta = \left\{ \rho : \rho(A_i) \in \bigcup\limits_{(B,\varrho) \in I} \varrho(B), \quad 1 \leq i \leq n \right\} \end{array} \right\}$$

where solutions to this equation are ordered as follows: $I_1 \leq I_2$ if, for all head atoms B, $(B, \varrho_1) \in I_1$ and $(B, \varrho_2) \in I_2$ implies that $\varrho_1 \subseteq \varrho_2$.

Now, a solution I of this equation is a specification of a set environment to each rule in P. Using the fact that each rule in a logic program is assumed to have a unique label, I can be viewed as a specification of a set of elements of the form ϱ^α, one for each rule R^α in P. Using this notion, the single equation above can be decomposed into a collection of equations, one for each rule label α as follows:

$$\varrho^\alpha \;=\; \mathcal{A}\left(\left\{ \rho : \rho(A_i) \in \bigcup_{R^\beta \in P} \varrho^\beta(\mathit{head}(R)), \quad 1 \leq i \leq n \right\} \right) \tag{5.10}$$

where $\mathit{head}(R)$ denotes the head of the rule R. Now, consider replacing equality in these constraint by inequality to obtain the constraints:

$$\varrho^{\alpha} \;\supseteq\; \mathcal{A}\left(\left\{\ \rho : \rho(A_i) \in \bigcup_{R^{\beta}\in P} \varrho^{\beta}(head(R)),\quad 1 \leq i \leq n\ \right\}\right) \tag{5.11}$$

where α ranges over rule labels in P. Importantly, it can be shown that the least model of (5.11) is the same as the least model of (5.10).

Now, consider the expressions $\rho(A_i) \in \bigcup_{R^{\beta}\in P} \varrho^{\beta}(head(R))$. The only components of $\bigcup_{R^{\beta}\in P} \varrho^{\beta}(head(R))$ that may contain a term of the form $\rho(A_i)$ are those terms that match A_i. Hence

$$\rho(A_i) \in \bigcup_{R^{\beta}\in P} \varrho^{\beta}(B^{\beta}) \quad \text{iff} \quad \rho(A_i) \in \bigcup \left\{ \varrho^{\beta}(head(R)) : \begin{array}{r} R^{\beta} \in P \text{ and} \\ head(R) \text{ and } A_i \\ \text{are compatible} \end{array} \right\}$$

It follows that the constraints (5.11) are satisfied if and only if the following constraints are satisfied:

$$\varrho^{\alpha} \;\supseteq\; \left\{\rho : \rho(A_1) \in \varrho^{\beta_1}(B_1)\ldots, \rho(A_n) \in \varrho^{\beta_n}(B_n)\right\} \tag{5.12}$$

where the β_i range over rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible.

Observe that the constraints (5.12) are virtually identical in meaning to the set based interpretation of the bottom-up environment constraints. In fact the only essential difference is that variables ϱ^{β} are used instead of Ψ^{β}. Recalling that $R^{\beta} \in P$ indicates that the rule with label β in P is R, it is now easy to see that bottom-up set based analysis of logic programs corresponds to $lfp(\mathcal{T}_{\mathrm{P}})$ in the following sense:

Proposition 15 *Let P be a logic program, and let $\mathcal{EC}_P$ be the bottom-up environment constraints for P. Then*

$$lfp(\mathcal{T}_{\mathrm{P}}) \quad = \quad \bigcup_{R^{\beta}\in P} sba_P(\Psi^{\beta})(head(R)) \qquad \Box$$

We conclude by discussing the use of non abstract interpretation approaches to the analysis of imperative and functional programs. The two main works here are by Jones and Muchnick [32, 34] and Reynolds [63]. Instead of developing an approximate program semantics in the style of Y_{P} or $\mathcal{T}_{\mathrm{P}}$, these works focus on the use set constraints to obtain information about the possible run-time values of program variables.

In [32], an analysis is described for an imperative language with LISP-like data structures (this work was later generalized in [34]). The essence of this approach is the construction of set constraints corresponding to a program to capture the flow of values from one variable to another as the program is executed. Underlying this work is the intuition of treating program variables as sets of values and this is inherited by set based analysis. However, the set constraints are constructed in such a manner that they can be solved by a fairly straightforward algorithm. In particular the set constraints do not contain a notion of intersection, and their only operation is projection (corresponding to decomposition of data structures). Hence they are not expressive enough to capture a number of important components of programs. For example, all information about the conditions in conditional statements is completely omitted. Also information relating to well definedness of expressions is ignored (for example, after a statement $X = car(Y)$, it must be the case that Y is of the form $cons(\cdots)$ because otherwise the program would have terminated with an error).

In contrast, the earlier paper [63] uses set constraints to compute data type definitions for program variables in a first order functional language. The constraints used are similar to those used in [32]. Again the only set operation of the constraints is projection, and so the program approximations obtained are considerably less accurate than set based approximations.

In summary, the set constraints used in [32, 63] are substantially simpler than those used in set based analysis. This has the advantage that they are much easier to solve. However, the program approximations that they define are significantly less accurate than set based approximations. Another major difference is that these approaches have viewed constraints as a tool for obtaining information about the program, and the constraints themselves incorporate a number of *ad hoc* approximations in addition to ignoring inter-variable dependencies.

The general approach of [32, 63] has been extended by Jones [31] to deal with higher order functions. This approach has been further developed for binding time analysis [50], garbage collection [30] and globalization of function parameters [56]. One presentational difference in these works is the use of various extensions of regular grammars instead of constraints. However there is a strong duality between such grammars and set constraints since there is a natural way to view set constraints as grammars and vice-versa. In particular, the technical details are broadly similar to [31, 32, 63].

Chapter 6

Set Constraints

Previous chapters have described how environment constraints can be used to characterize the run-time behavior of programs. Subsequently, set based program approximation was defined by treating program variables as sets of values. As a first step towards computing set based approximations, we now show how environment constraints may be translated into set constraints such that the least set based model of the environment constraints corresponds to the least model of the set constraints. This translation uses the fact that, when interpreted using set based interpretations, certain aspects of environment constraints may be significantly simplified.

We first describe the general form of the set constraints used and prove some basic properties. Then, for each kind of environment constraint, we give the translation into set constraints and show that is correct. In effect, this translation reduces the problem of computing the set based approximation of a program into the problem of computing the least model of a collection of set constraints.

6.1 Set Constraints

We define a scheme for set based calculi. This scheme is defined in the context of some alphabet Σ of function symbols where each function symbol f comes equipped with a unique arity denoted $arity(f)$. The letters f, g and h shall be used to denote function symbols. A function symbol of arity 0 is called a *constant*. A *value* is an expression constructed from the function symbols in Σ, viz: $f(v_1, \ldots, v_{arity(f)})$ is a value if each v_i is a value. We shall assume a countably infinite collection $\mathcal{VAR}$ of set variables. Set variables shall be denoted $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$, etc., and shall be interpreted as sets of values.

The main parameter of the calculus scheme is a collection of operations for combining sets of values. Specifically let OP be a collection of *set operations*, where each operation $op \in OP$ has an associated arity denoted $arity(op)$, as well as a meaning function $[\![op]\!]$, which maps any sequence of sets of values $(S_1, \ldots, S_{arity(op)})$ into a set of values. In the context of such a collection of operations, define that a *set expressions se* is either:

- a set variable;
- one of the special constants $\top$ or $\bot$;
- $se_1 \cup se_2$;
- $f(se_1, \ldots, se_n)$ where f is an n-ary symbol from Σ, or
- $op(se_1, \ldots, se_n)$ where op is an n-ary operation from OP,

where the se_i are set expressions. Note that the constant $\top$ also appears in environment constraints. However, it will always be clear from context whether $\top$ represents an environment expression or a set expression. A *set constraint* is of the form $se_1 \supseteq se_2$ or $se_1 = se_2$. Collections of set constraints shall be denoted by the symbol $\mathcal{C}$. Where $exp_1, \ldots, exp_n$, $n \geq 1$, is a sequence of set expressions or set constraints, $var(exp_1, \ldots, exp_n)$ denotes the collection of all set variables that appear in $exp_1, \ldots, exp_n$.

To define the meaning of set expressions, let $\mathcal{I}$ be a mapping from each set variable into sets of values. Then $\mathcal{I}(se)$ is defined to be:

- $\mathcal{I}(\mathcal{X})$ if $\mathcal{X}$ is a set variable;

- the set of all values, if se is $\top$;
- the empty set $\{\}$, if se is $\perp$;
- $\mathcal{I}(se_1) \cup \mathcal{I}(se_2)$, if se is $se_1 \cup se_2$;
- $\{f(v_1, \ldots, v_n) : v_i \in \mathcal{I}(se_i)\}$ if se is $f(se_1, \ldots, se_n)$, or
- $[\![op]\!](\mathcal{I}(se_1), \ldots, \mathcal{I}(se_n))$ if se is $op(se_1, \ldots, se_n)$.

$\mathcal{I}$ is a *model* of a constraint $se_1 \supseteq se_2$ or $se_1 = se_2$ if $\mathcal{I}(se_1) \supseteq \mathcal{I}(se_2)$ or $\mathcal{I}(se_1) = \mathcal{I}(se_2)$ respectively. $\mathcal{I}$ is a model of a collection $\mathcal{C}$ of constraints, denoted $\mathcal{I} \models \mathcal{C}$, if $\mathcal{I}$ is a model of each constraint in the collection. Models shall be ordered componentwise: $\mathcal{I}_1 \supseteq \mathcal{I}_2$ iff $\mathcal{I}_1(\mathcal{X}) \supseteq \mathcal{I}_2(\mathcal{X})$ for each set variable $\mathcal{X}$. We write $lm(\mathcal{C})$ to denote the least model of $\mathcal{C}$ if it exists.

Note that the meaning $\mathcal{I}(ee)$ of a set expression ee is defined in terms of the meanings of the immediate subexpressions of ee, and it follows that "equal" terms can be replaced in all contexts. Specifically,

Proposition 16 *Let se be a set expression that contains se_1 as a subexpression. Let se' be the result of replacing se_1 in se by the set expression se_2. If $\mathcal{I}(se_1) = \mathcal{I}(se_2)$ then $\mathcal{I}(se) = \mathcal{I}(se')$.*

Proof: By structural induction on se. $\square$

We now give some example constraints. Let $\mathcal{C}$ denote the single constraint $\mathcal{X} \supseteq c \ \cup \ f(f(\mathcal{X}))$, where c is a constant and f is a unary symbol. $\mathcal{C}$ has many models, including the interpretation that maps all set variables into the set $\{c, f(c), f(f(c)), \ldots\}$. Another model of $\mathcal{C}$ is the interpretation $\mathcal{I}$ defined by

$$\mathcal{I}(\mathcal{Y}) = \begin{cases} \{c, f^2(c), f^4(c), \ldots\} & \text{if } \mathcal{Y} \text{ is } \mathcal{X} \\ \{\} & \text{if } \mathcal{Y} \text{ is different from } \mathcal{X} \end{cases}$$

where f^n abbreviates n applications of f. This model is smaller than the first, and is in fact the least model of $\mathcal{C}$.

Without using operators from $\mathcal{OP}$, the constraints that can be formed have a simple structure. In particular, it is fairly easy to reason about their least model because the structure of the least model is readily apparent from the constraints. Specifically, if $\mathcal{C}$ is a collection of constraints involving only set variables, $\perp$, $\top$, union and function symbols, then there is a simple polynomial time algorithm for determining $v \in lm(\mathcal{C})(\mathcal{X})$ where v is a value and $\mathcal{X}$ is a set variable. We shall discuss this further in the next chapter, and also show that such collections $\mathcal{C}$ correspond to regular tree grammars, or alternatively, regular tree automaton.

The most interesting aspect of set constraints lies in the set operators that make up the parameter $\mathcal{OP}$. One of the simplest set operators is intersection, which is given its usual set theoretic interpretation so that if $\mathcal{I}(\mathcal{X}) = \{a, b, c\}$ and $\mathcal{I}(\mathcal{Y}) = \{b, c, d\}$ then $\mathcal{I}(\mathcal{X} \cap \mathcal{Y}) = \{b, c\}$. As an example of the use of intersection, consider the following constraints:

$$\begin{array}{lcl} \mathcal{X} & \supseteq & a \cup f^3(\mathcal{X}) \\ \mathcal{Y} & \supseteq & a \cup f^2(\mathcal{Y}) \\ \mathcal{Z} & \supseteq & \mathcal{X} \cap \mathcal{Y} \end{array}$$

The least model of these constraints maps $\mathcal{X}$ into $\{a, f^3(a), f^6(a), \ldots\}$, maps $\mathcal{Y}$ into $\{a, f^2(a), f^4(a), \ldots\}$, maps $\mathcal{Z}$ into $\{a, f^6(a), f^{12}(a), \ldots\}$, and maps all other variables into $\{\}$. For convenience, we shall consider n-ary intersections, written $\cap_n$ where $n \geq 2$. The subscript n shall usually be omitted and an expression $\cap_n(se_1, \ldots, se_n)$ shall be written as $se_1 \cap \cdots \cap se_n$.

Another kind of operator is projection. Specifically, for each n-ary symbol $f \in \Sigma$, there are n projection operators, $f^{-1}_{(1)}$, ..., $f^{-1}_{(n)}$. The meaning of each operator $f^{-1}_{(i)}$ is defined by the function $[\![f^{-1}_{(i)}]\!]$, which maps a set S of value into $\{v_i : f(v_1, \ldots, v_n) \in S\}$. For example, consider the following constraint:

$$\mathcal{X} \;\supseteq\; f(f(\mathcal{X})) \;\cup\; a \;\cup\; f^{-1}_{(1)}(\mathcal{X})$$

The least model of this constraint maps $\mathcal{X}$ into $\{a, f(a), f(f(a)), \ldots\}$ and maps all other variables into the empty set.

The next class of operators have arity 0. Their purpose is to allow a very limited form of complementation to be expressed. Specifically, where se is a ground set expression, define that $\overline{se}$ is a *complement constant* that represents the complement of se. That is, for any interpretation $\mathcal{I}$, $\mathcal{I}(\overline{se}) = \{v : v \notin \mathcal{I}(se)\}$. Since se is ground, complement constants $\overline{se}$ have a fixed meaning over all interpretations. We choose not to introduce complementation in its full generality because it is not monotonic. The limited form of complementation embodied in the complement constants proves to be useful for reasoning about certain aspects of imperative programs such as inequality and negated match conditions.

Although the ground set expression se in a complement constant $\overline{se}$ may be arbitrary, we shall not use this generality in the constraints employed to compute sba_P. Specifically, we shall only use complement constants that have the form $\overline{a_1 \cup \cdots \cup a_n}$ where each a_i is either of the form $f_i(\top, \ldots, \top)$ or contains only function symbols. For notational convenience, we shall frequently write complement constants in the form $\overline{S}$ where S is a set of ground set expressions, so that if S is $\{a_1, \ldots, a_n\}$ then $\overline{S}$ denotes $\overline{a_1 \cup \cdots \cup a_n}$. For example, $\overline{\{nil, cons(\top, \top)\}}$ denotes $\overline{nil \cup cons(\top, \top)}$, which describes the set of values whose top-most symbol is not nil or $cons$. We shall identify the constant $\top$ with $\overline{\{\ \}}$.

Note that if Σ is finite, then complement constraints $\overline{S}$ can be considered to be a notation for a somewhat unwieldy ground set expression. For example, if Σ is $\{f, g, a\}$, then $\overline{f(a)}$ can be identified with the expression $g(\top) \cup a \cup f(\overline{a})$ where $\overline{a}$ denotes $f(\top) \cup g(\top)$. We choose to use explicit complement constants because it gives a slightly more general treatment, and leads to more efficient algorithms.

The final class of set operators used in this thesis are quantified operators. In essence, a quantified operator of arity n is a formulas with n holes. We begin by defining these formulas. A *quantified set expression* is of the form $\{X : conj\}$ where X is a program variable and $conj$ is a conjunction of quantified conditions of the form $s \in se$ or $s \dagger se$ where s is a program term and se is a set expression. If $conj$ is the empty conjunction, then $\{X : conj\}$ is identified with $\top$. Now, a quantified operator is essentially a quantified set expression with the set expressions missing. Specifically, a quantified operator op consists of a program variable X and a sequence of $m \geq 0$ formulas, each of the form $(s \in \cdot)$ or $(s \dagger \cdot)$. The result of applying op to a sequence of set expressions $se_1, \ldots, se_m$ is

$$\{X : conj_1 \wedge \cdots \wedge conj_m\}$$

where, for $i = 1..m$, $conj_i$ is $(s \in se_i)$ if the i^{th} formula in op is $(s \in \cdot)$, and $conj_i$ is $(s \dagger se_i)$ if the i^{th} formula in op is $(s \dagger \cdot)$. For example, if op consists of the program variable X and the sequence of formulas $(f(X) \in \cdot), (X \dagger \cdot)$ then $op(f(\mathcal{Z}), \mathcal{Y})$ is $\{X : f(X) \in f(\mathcal{Z}) \wedge X \in \mathcal{Y}\}$.

The meaning of quantified operators is defined by giving a meaning to quantified set expressions. Let $\mathcal{I}$ be an interpretation that maps each set variable into a set of values. Then $\mathcal{I}(\{X : conj\})$ is defined to be $\{\rho(X) : \rho \in \mathcal{I}(conj)\}$ where $\rho \in \mathcal{I}(conj)$ if

$$\begin{array}{lllll} & \rho \rhd s & \wedge & \rho(s) \in \mathcal{I}(se) & \text{for all } s \in se \text{ in } conj, \\ \text{and} & \rho \rhd s & \wedge & \exists v\,(v \neq \rho(s) \;\wedge\; v \in \mathcal{I}(se)) & \text{for all } s \dagger se \text{ in } conj. \end{array}$$

Intuitively, $\rho \in \mathcal{I}(s \in se)$ if $\rho(s)$ is contained in the set of values that $\mathcal{I}$ assigns se, and $\rho \in \mathcal{I}(s \dagger se)$ if $\mathcal{I}(se)$ contains a value different from $\rho(s)$. For example, consider the following constraints:

$$\begin{array}{lll} \mathcal{X} & \supseteq & \left\{X : X \dagger \mathcal{Z} \;\wedge\; X \in \overline{f(\top)} \;\wedge\; X \in \mathcal{Y}\right\} \\ \mathcal{Y} & \supseteq & a \cup b \cup c \cup f(a) \\ \mathcal{Z} & \supseteq & c \end{array}$$

The least model of these constraints maps $\mathcal{Z}$ into $\{c\}$, $\mathcal{Y}$ into $\{a, b, c, f(a)\}$ and $\mathcal{X}$ into $\{a, b\}$. Note that if the constraint $\mathcal{Z} \supseteq a$ is added, then the least model changes and $\mathcal{X}$ is now assigned $\{a, b, c\}$. This is because the least model now maps $\mathcal{Z}$ into $\{a, c\}$, and so the disjointness condition $X \dagger \mathcal{Z}$ is becomes vacuous.

Intuitively, quantified conditions of the form $s \in se$ are elementhood relationships, and are generated from atomic program conditions of the form $s = t$, $match_f(s)$ and $\neg match_f(s)$. Quantified conditions of the form $s \dagger se$ are "apartness" relationships and are generated from atomic program conditions of the form $\neg(s = t)$. For example, consider the program condition $X \neq Y$. In essence, this shall be translated to

$$(X \in \mathcal{X}) \wedge (X \dagger \mathcal{Y}) \wedge (Y \in \mathcal{Y}) \wedge (Y \dagger \mathcal{X})$$

where $\mathcal{X}$ and $\mathcal{Y}$ respectively denote the sets of values for X and Y at the point just before execution of the conditional statement. The idea is that the two apartness conditions capture the requirement that X and Y assume distinct values. For example, if $\mathcal{X}$ is $\{a, b\}$ and $\mathcal{Y}$ is $\{b\}$, then the only possible values for X and Y after the conditional are a and b respectively.

To summarize, the set operators used to compute sba_P include intersection, projections, complement constants and quantified operators. We note that quantified operators may be viewed as a generalization of projections because any expression $f_{(i)}^{-1}(se)$ can be translated into the quantified expression $\{X_i : f(X_1, \ldots, X_n) \in se\}$ where f is n-ary and $X_1, \ldots, X_n$ are distinct program variables.

Central to our work on set constraints is the notion of least model. In general, these models do not exist. For example, the constraint $\mathcal{X} \cup \mathcal{Y} = a$ has two minimal models as follows

$$\mathcal{I}_1(\mathcal{Z}) = \begin{cases} \{a\} & \text{if } \mathcal{Z} \text{ is } \mathcal{X} \\ \{\} & \text{otherwise} \end{cases} \qquad \mathcal{I}_2(\mathcal{Z}) = \begin{cases} \{a\} & \text{if } \mathcal{Z} \text{ is } \mathcal{Y} \\ \{\} & \text{otherwise} \end{cases}$$

but does not have a least model. However, in certain circumstances, a least model is guaranteed to exist. Specifically, define that a constraint is in *variable-expression form* if is has the form $\mathcal{X} \supseteq se$ where $\mathcal{X}$ is a variable and se is a set expression. All of the constraints used in this thesis shall have this form. Now, Corollary 4 of the Appendix I shows that the least model of a collection of variable-expression form constraints exists if the set operators appearing in the constraints are monotonic in all arguments. It is easy to verify that all of the operators we have introduced are monotonic, and hence the collections of constraints we shall consider will always have least models.

Any constraint of the form $\mathcal{X} \supseteq se$ shall be referred to as a *lower bound for* the set variable $\mathcal{X}$ because in any model of this constraint $\mathcal{X}$ contains at least se. The following two propositions establish some useful properties about lower bounds; both propositions are instances of more general propositions that can be found in Appendix I (see propositions 48 and 49 respectively).

Proposition 17 *Let $\mathcal{C}$ be a collection of constraints in variable-expression form. If $v \in lm(\mathcal{C})(\mathcal{X})$ then $\mathcal{C}$ contains a lower bound on $\mathcal{X}$ of the form $\mathcal{X} \supseteq se$ such that $v \in lm(\mathcal{C})(se)$.* $\square$

Proposition 18 *Let $\mathcal{C}$ be a collection of constraints in variable-expression form in which all set operators are monotonic. If $\mathcal{X} \supseteq se$ is the only lower bound for $\mathcal{X}$ then $lm(\mathcal{C})(\mathcal{X}) = lm(\mathcal{C})(se)$.* $\square$

This last proposition implies that if $\mathcal{X}_1, \ldots, \mathcal{X}_n$ is a sequence of distinct variables, then any collection of constraints of the form

$$\mathcal{X}_1 \supseteq se_1, \ldots, \mathcal{X}_n \supseteq se_n$$

has the same least model as the constraints $\mathcal{X}_1 = se_1, \ldots, \mathcal{X}_n = se_n$. Now, consider an arbitrary collection of variable-expression form constraints and suppose that the following step is repeatedly applied to the collection: replacing two constraints $\mathcal{X} \supseteq se$ and $\mathcal{X} \supseteq se'$ by the single constraint $\mathcal{X} \supseteq se \cup se'$. The result is an equivalent collection of constraints of the form $\mathcal{X}_1 \supseteq se_1, \ldots, \mathcal{X}_n \supseteq se_n$ where $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are distinct variables. It follows that variable-expression form collections of constraints are completely interchangeable (w.r.t. least models) with constraints of the form $\mathcal{X}_1 = se_1, \ldots, \mathcal{X}_n = se_n$ where $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are distinct. Hence, we can choose between these two forms of constraints. In two previous papers on set constraints [21, 24], we chose to use the equational form. However, in this thesis we use the $\supseteq$ form because it simplifies some of the presentation.

We also note that when the set operators in $\mathcal{OP}$ are monotonic, proposition 16 can be strengthened as follows.

Proposition 19 *Let se be a set expression that contains se_1 as a subexpression. Let se' be the result of replacing se_1 in se by the set expression se_2. If $\mathcal{I}(se_1) \subseteq \mathcal{I}(se_2)$ then $\mathcal{I}(se) \subseteq \mathcal{I}(se')$.* $\square$

We conclude this section by relating variable-expression form constraints and the *definite* set constraints considered by Heintze and Jaffar in [22]. Whereas variable-expression form constraints are of the form $\mathcal{X} \supseteq se$, definite set constraints are of the form $ae \supseteq se$ such that ae is a set expression that does not contain any set operators. Clearly definite set constraints are a strict generalization of variable-expression form constraints. A collection of definite constraints may not have any models, but if there is a model then there is a least model. In essence, [22] proceeds by reducing such constraints into constraints of the form $\mathcal{X} \supseteq se$, and the core algorithm of the paper then solves these reduced constraints. The core algorithm of [22] is essentially an early version of the set constraint algorithm described in the next chapter.

6.2 Environment Constraints and Set Constraints

We now translate environment constraints into set constraints in such a way that the least set based interpretation of the environment constraint corresponds to the least model of the set constraints. Let P be a program and let $X_1, \ldots, X_m$ denote the program variables in P. Now, for each program point μ, introduce set variables $\mathcal{X}_1^\mu, \ldots, \mathcal{X}_m^\mu$. Intuitively, the set variable $\mathcal{X}_i^\mu$ shall denote the set of values for the program variable X_i at the point μ. In other words, the set environment Ψ^μ shall be represented by the tuple of set variables $(\mathcal{X}_1^\mu, \ldots, \mathcal{X}_m^\mu)$.

The construction of set constraints from environment constraints requires the systematic replacement of program variables with set variables. For example, consider the environment constraint $\Psi^\mu \supseteq \Psi^\lambda$, which states that each environment in Ψ^λ must appear in Ψ^μ. If Ψ^λ is represented by $(\mathcal{X}_1^\lambda, \ldots, \mathcal{X}_m^\lambda)$ and Ψ^λ by $(\mathcal{X}_1^\mu, \ldots, \mathcal{X}_m^\mu)$, then appropriate set constraints for $\Psi^\mu \supseteq \Psi^\lambda$ are:

$$\mathcal{X}_i^\mu \supseteq \mathcal{X}_i^\lambda \quad \text{for } i = 1..n$$

Now consider the environment constraint $\Psi^\mu \supseteq \Psi^\lambda[X_3 \mapsto cons(X_1, X_2)]$, corresponding to an assignment statement $X_3 := cons(X_1, X_2)$. The values for the variables different from X_3 remain unchanged. The value for X_3 is given by treating $\mathcal{X}_m^\lambda$ as a set mapping and applying it to $cons(X_1, X_2)$. This can be expressed using set constraints as follows:

$$\begin{aligned} \mathcal{X}_3^\mu &\supseteq cons(\mathcal{X}_1^\lambda, \mathcal{X}_2^\lambda) \\ \mathcal{X}_i^\mu &\supseteq \mathcal{X}_i^\lambda \quad \text{for } i \neq 3 \end{aligned}$$

A more complicated example is $\Psi^\mu \supseteq \Psi^\lambda[X_2 \mapsto car(X_3)]$, corresponding to an assignment statement $X_2 := car(X_3)$. In essence, we wish to construct the following set constraints:

$$\begin{aligned} \mathcal{X}_2^\mu &\supseteq car(\mathcal{X}_3^\lambda) \\ \mathcal{X}_i^\mu &\supseteq \mathcal{X}_i^\lambda \quad \text{for } i \neq 2 \end{aligned}$$

However these set constraints are not faithful to the meaning of $\Psi^\mu \supseteq \Psi^\lambda[X_2 \mapsto car(X_3)]$. This is because $car(\mathcal{X}_3^\lambda)$ provides an implicit restriction on the values for X_3 after the assignment statement: they must all be of the form $cons(\cdots)$. To appropriately modify the constraints so that they reflect this condition, recall that the definition of the set based interpretation of an environment expression $\Psi[X \mapsto t]$ is:

$$\mathcal{I}\left(\Psi[X \mapsto t]\right) = \varrho[X \mapsto \varrho(t)] \quad \text{where} \quad \varrho \text{ is } \mathcal{A}\left(\{\rho \in \mathcal{I}(\Psi) : \rho \rhd t\}\right).$$

That is, the definition first constructs a set environment $\mathcal{A}(\varrho = \{\rho \in$

$\mathcal{I}(\Psi) : \rho \rhd t\})$ and then modifies this set environment to reflect the assignment $X := t$. This indicates a two stage process, and the set constraints corresponding to $\Psi^\mu \supseteq \Psi^\lambda[X_2 \mapsto car(X_3)]$ are similarly constructed in two stages. Specifically, let $\mathcal{X}_1, \ldots, \mathcal{X}_n$ be new set variables, and the intention is that these variables will be used to capture the "temporary" set environment $\varrho = \mathcal{A}\left(\{\rho \in \mathcal{I}(\Psi) : \rho \rhd car(X_3)\}\right)$. Now, corresponding to $\Psi^\mu \supseteq \Psi^\lambda[X_2 \mapsto car(X_3)]$, construct the following set constraints:

$$\left.\begin{array}{rcl} \mathcal{X}_3 & \supseteq & \{X_3 : X_3 \in cons(\top, \top) \wedge X_3 \in \mathcal{X}_3^\lambda\} \\ \mathcal{X}_i & \supseteq & \mathcal{X}_i^\lambda \quad \text{for } i \neq 3 \end{array}\right\} (1)$$

$$\left.\begin{array}{rcl} \mathcal{X}_2^\mu & \supseteq & car(\mathcal{X}_3) \\ \mathcal{X}_i^\mu & \supseteq & \mathcal{X}_i \quad \text{for } i \neq 2 \end{array}\right\} (2)$$

The first group of set constraints, labeled (1), corresponds to the construction of set environment ϱ, and in essence restricts the values of X_3 so that $car(X_3)$ is defined. The second group of set constraints, labeled (2), updates set for X_2 to $car(\mathcal{X}_3)$, and retains the sets for the other variables.

The actual constraints used are slightly more complex than these examples suggest. The main reason for this is that if $i \neq j$ then the set variables $\mathcal{X}_i^\mu$ and $\mathcal{X}_j^\mu$ are essentially independent of each other – there is no *a priori* reason why one cannot be empty and the other non-empty. However, recall that, by definition, a set environment ϱ is subject to the following requirement: if ϱ maps some program variable into the empty set, then it must map all program variables into the empty set. Hence, if the set variables $(\mathcal{X}_1^\mu, \ldots, \mathcal{X}_m^\mu)$ are used to represent a set environment Ψ^μ, then care must be taken to ensure that if $\mathcal{X}_i^\mu$ is empty, for some i, then $\mathcal{X}_i^\mu$ is empty for all i.

We now give the details of the set constraints. Since the treatment of program terms will require the replacement of program variables by set variables, first define that $[X_1, \ldots, X_m \mapsto \mathcal{X}_1, \ldots, \mathcal{X}_m]$ denotes the renaming substitution that maps X_i into $\mathcal{X}_i$, so that $t[X_1, \ldots, X_m \mapsto \mathcal{X}_1, \ldots, \mathcal{X}_m]$ is the result of replacing each program variable X_i by the set variable $\mathcal{X}_i$. The set constraints for a program P can now be defined by translating the environment constraints of P as follows.

Definition 13 (Set Constraints) *The* <u>*set constraints*</u> $\mathcal{SC}_P$ *for a program* P *are constructed as follows:*

(i) For each constraint $\Psi^\mu \supseteq \Psi^\lambda$, $\mathcal{SC}_P$ contains[1]*:*

$$\mathcal{X}_i^\mu \supseteq \{X_i : \textstyle\bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda\},\ i = 1..m.$$

(ii) For each constraint $\Psi^\mu \supseteq \top$, $\mathcal{SC}_P$ contains:

$$\mathcal{X}_i^\mu \supseteq \top, i = 1..m.$$

(iii) For each constraint $\Psi^\mu \supseteq \Psi^\lambda[X_l \mapsto t]$, $\mathcal{SC}_P$ contains:

$$\begin{aligned} \mathcal{X}_i &\supseteq \{X_i : \mathit{defined}(t) \ \wedge\ \textstyle\bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda\}, \quad i = 1..m \\ \mathcal{X}_i^\mu &\supseteq \left\{ X_i : \begin{array}{l} X_l \in t[X_1, \ldots, X_m \mapsto \mathcal{X}_1, \ldots, \mathcal{X}_m] \\ X_j \in \mathcal{X}_j, \quad 1 \le j \ne l \le m \end{array} \right\}, \quad i = 1..m \end{aligned}$$

where $\mathcal{X}_1, \ldots, \mathcal{X}_m$ are distinct new set variables and defined(t) denotes the conjunction of all quantified conditions $s \in f(\top, \ldots, \top)$ such that t has a subterm of the form $f_{(j)}^{-1}(s)$.

(iv) For each constraint $\Psi^\mu \supseteq \Psi^\lambda[cond]$, let cond be $conj_1 \vee \cdots \vee conj_n$ and $\mathcal{SC}_P$ contains:

$$\begin{aligned} \mathcal{X}_i &\supseteq \{X_i : \mathit{defined}(cond) \ \wedge\ \textstyle\bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda\}, \quad i = 1..m \\ \mathcal{X}_i^\mu &\supseteq \{X_i : \mathit{translate}(conj_k) \ \wedge\ \textstyle\bigwedge_{j=1..m} X_j \in \mathcal{X}_j\},\ i = 1..m,\ k = 1..n \end{aligned}$$

where $\mathcal{X}_1, \ldots, \mathcal{X}_m$ are distinct new set variables, defined$(cond)$ denotes the conjunction of all quantified conditions $s \in f(\top, \ldots, \top)$ such that cond has a subterm of the form $f_{(j)}^{-1}(s)$, and translate$(conj_k)$ is defined to be conjunction consisting of the following quantified conditions:

- $\left(\bigwedge \begin{array}{l} s \in t[X_1, \ldots, X_m \mapsto \mathcal{X}_1^\lambda, \ldots, \mathcal{X}_m^\lambda] \\ t \in s[X_1, \ldots, X_m \mapsto \mathcal{X}_1^\lambda, \ldots, \mathcal{X}_m^\lambda] \end{array} \right)$ *for each condition of the form $s = t$ in $conj_k$.*
- $\left(\bigwedge \begin{array}{l} s \dagger t[X_1, \ldots, X_m \mapsto \mathcal{X}_1^\lambda, \ldots, \mathcal{X}_m^\lambda] \\ t \dagger s[X_1, \ldots, X_m \mapsto \mathcal{X}_1^\lambda, \ldots, \mathcal{X}_m^\lambda] \end{array} \right)$ *for each condition of the form $\neg(s = t)$ in $conj_k$.*
- $t \in f(\top, \ldots, \top)$ *for each condition of the form $match_f(t)$ in $conj_k$.*

[1]We note that $\Psi^\mu \supseteq \Psi^\lambda$ could alternatively be translated into $\mathcal{X}_i^\mu \supseteq \mathcal{X}_i^\lambda$, $i = 1..m$. Although these alternative constraints are simpler and clearly desirable in practice, we use the more complex constraints for presentational reasons because they clarify relationships between the emptiness of the variables $\mathcal{X}_1^\mu, \ldots, \mathcal{X}_m^\mu$.

- $t \in \overline{f(\top, \ldots, \top)}$ *for each condition of the form $\neg(match_f(t))$ in $conj_k$.*

(v) For each constraint $\Psi^\mu \supseteq [A_1 \in B_1.\Psi^{\lambda_1}, \ldots, A_n \in B_n.\Psi^{\lambda_n}]$, $\mathcal{SC}_P$ contains:

$$\mathcal{X}_i^\mu \supseteq \left\{ X_i : \bigwedge_{k=1..n} A_k \in B_k[X_1, \ldots, X_m \mapsto \mathcal{X}_1^{\lambda_k}, \ldots, \mathcal{X}_m^{\lambda_k}] \right\}, \ i = 1..m$$

□

The set constraints $\mathcal{SC}_P$ correctly characterize $approx_P$ in the following sense.

Proposition 20 *Let P be a program. Then, for all program points μ,*

$$sba_P(\Psi^\mu)(X_i) = (lm(\mathcal{SC}_P))\,(\mathcal{X}_i^\mu), \quad i = 1..m.$$

Proof: The first step of the proof establishes a strong correspondence between models of $\mathcal{EC}_P$ and models of $\mathcal{SC}_P$. Specifically, if $\mathcal{I}_{ec}$ is an interpretation of $\mathcal{EC}_P$ and $\mathcal{I}_{sc}$ is an interpretation of $\mathcal{SC}_P$ then

$$\begin{array}{l} \text{If (a) } \mathcal{I}_{sc}(\mathcal{X}_i^\mu) = \mathcal{I}_{ec}(\Psi^\mu)(X_i) \text{ for all } \mu \text{ and } i, \text{ and} \\ \quad \text{(b) } \mathcal{I}_{sc}(\mathcal{X}) = \mathcal{I}_{sc}(se) \text{ for all constraints } \mathcal{X} \supseteq se \text{ in } \mathcal{SC}_P \\ \qquad \text{where the variable } \mathcal{X} \text{ is not of the form } \mathcal{X}_i^\mu \\ \text{then } \mathcal{I}_{ec} \models \mathcal{EC}_P \text{ iff } \mathcal{I}_{sc} \models \mathcal{SC}_P. \end{array} \tag{6.13}$$

To prove property (6.13) let $\mathcal{I}_{ec}$ be a model of $\mathcal{EC}_P$ and let $\mathcal{I}_{sc}$ be a model of $\mathcal{SC}_P$, and suppose that conditions (a) and (b) are satisfied. Now, corresponding to each constraint in $\mathcal{EC}_P$, a collection of constraints is introduced into $\mathcal{SC}_P$. It therefore suffices to show that for each of the cases (i–v) in the construction of $\mathcal{SC}_P$, the environment constraint considered is satisfied by $\mathcal{I}_{ec}$ iff the set constraints constructed are satisfied by $\mathcal{I}_{sc}$. Consider each case in turn.

In case (i), an environment constraint $\Psi^\mu \supseteq \Psi^\lambda$ is considered and set constraints $\mathcal{X}_i^\mu \supseteq \{X_i : \bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda\}$, $i = 1..m$ are constructed. To prove this case, consider the following chain of reasoning:

$$\begin{array}{ll}
& \mathcal{I}_{ec}(\Psi^\mu) \;\supseteq\; \mathcal{I}_{ec}(\Psi^\lambda) \\
\text{iff} & \mathcal{I}_{ec}(\Psi^\mu)(X_i) \;\supseteq\; \mathcal{I}_{ec}(\Psi^\lambda)(X_i) \\
\text{iff} & \mathcal{I}_{ec}(\Psi^\mu)(X_i) \;\supseteq\; \{\rho(X_i) : \rho \in \mathcal{I}_{ec}(\Psi^\lambda)\} \\
\text{iff} & \mathcal{I}_{ec}(\Psi^\mu)(X_i) \;\supseteq\; \{\rho(X_i) : \rho(X_j) \in \mathcal{I}_{ec}(\Psi^\lambda)(X_j), j = 1..m\} \\
\text{iff} & \mathcal{I}_{ec}(\Psi^\mu)(X_i) \;\supseteq\; \{\rho(X_i) : \rho(X_j) \in \mathcal{I}_{sc}(\mathcal{X}_j^\lambda), j = 1..m\} \\
\text{iff} & \mathcal{I}_{sc}(\mathcal{X}_i^\mu) \;\supseteq\; \mathcal{I}_{sc}(\bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda).
\end{array}$$

The first step in this chain follows from the pointwise ordering of the comparison of set environments. The second and third steps follow from the definition of set environments. The fourth step follows from the equality $\mathcal{I}_{sc}(\mathcal{X}_i^\mu) = \mathcal{I}_{ec}(\Psi^\mu)(X_i)$. The last step just follows from the definition of $\mathcal{I}_{sc}(\bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda)$.

In case (ii), an environment constraint $\Psi^\mu \supseteq \top$ is considered and set constraints $\mathcal{X}_i^\mu \supseteq \top$, $i = 1..m$ are constructed. Clearly $\mathcal{I}_{ec}(\Psi^\mu) \supseteq \mathcal{I}_{ec}(\top)$ iff $\mathcal{I}_{ec}(\Psi^\mu)(X_i) = \{\textit{all values}\}$ iff $\mathcal{I}_{sc}(\mathcal{X}_i^\mu) \supseteq \mathcal{I}_{sc}(\top)$.

Now consider case (iii), and let ϱ be the set environment $\mathcal{A}(\{\rho \in \mathcal{I}_{ec}(\Psi^\mu) : \rho \rhd t\})$. Since condition (b) is satisfied, the set constraints $\mathcal{X}_i \supseteq \{X_i : \textit{defined}(t) \wedge \bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda\}, i = 1..m$ are guaranteed to be satisfied by $\mathcal{I}_{sc}$. Condition (b) also implies that $\varrho(X_i) = \mathcal{I}_{sc}(\mathcal{X}_i)$ for $i = 1..m$, as the following chain of reasoning demonstrates:

$$\begin{array}{rcl}
\varrho(X_i) & = & \{\rho(X_i) : \rho \rhd t \text{ and } \rho \in \mathcal{I}_{ec}(\Psi^\lambda)\} \\
& = & \{\rho(X_i) : \rho \rhd t \text{ and } \rho(X_j) \in \mathcal{I}_{ec}(\Psi^\lambda)(X_j),\ j = 1..m\} \\
& = & \{\rho(X_i) : \rho \rhd t \text{ and } \rho(X_j) \in \mathcal{I}_{sc}(\mathcal{X}_j^\lambda),\ j = 1..m\} \\
& = & \left\{\rho(X_i) : \begin{array}{ll} \rho(s) \text{ has form } f(\cdots) & \text{for each subterm } f_{(k)}^{-1}(s) \text{ of } t \\ \rho(X_j) \in \mathcal{I}_{sc}(\mathcal{X}_j^\lambda) & \text{for } j = 1..m \end{array}\right\} \\
& = & \left\{\rho(X_i) : \begin{array}{ll} \rho(s) \in \mathcal{I}(se) & \text{for each } s \in se \text{ in } \textit{defined}(t) \\ \rho(X_j) \in \mathcal{I}_{sc}(\mathcal{X}_j^\lambda) & \text{for } j = 1..m \end{array}\right\} \\
& = & \mathcal{I}_{sc}\left(\{X_i : \textit{defined}(t) \wedge \bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda\}\right) \\
& = & \mathcal{I}_{sc}(\mathcal{X}_i).
\end{array}$$

The first equality is just the definition of ϱ. The second follows from the fact that $\rho \in \mathcal{I}_{ec}(\Psi^\mu)$ iff $\bigwedge_{j=1..m} \rho(X_j) \in \mathcal{I}_{ec}(\Psi^\mu)(X_j)$. The third follows from hypothesis (b) of (6.13). The fourth follows from the observation that the place in the definition of $\rho(t)$ where undefinedness can be introduced is at the evaluation of projections. In particular, it is easy to verify that $\rho(t)$ is defined iff for all subterms of t that are of the form $f^{-1}_{(i)}(s)$, it is the case that $\rho(s)$ is $f(v_1, \ldots, v_n)$ for some values v_i. The fifth equality follows from the definition of *defined*. The sixth equality follows from the definition of $\mathcal{I}_{sc}$. Finally, the seventh equality follows from condition (b).

Having established that $\varrho(X_i) = \mathcal{I}_{sc}(\mathcal{X}_i)$, it follows from a simple structural induction argument on any program term t, that

$$\varrho(t) = \mathcal{I}_{sc}(t[X_1, \ldots, X_m \mapsto \mathcal{X}_1, \ldots, \mathcal{X}_m]).$$

The proof for case (iii) can now be completed by the following chain of reasoning:

$$\mathcal{I}_{ec}(\Psi^\mu) \;\supseteq\; \mathcal{I}_{ec}(\Psi^\lambda[X_l \mapsto t])$$

$$\mathcal{I}_{ec}(\Psi^\mu) \;\supseteq\; \varrho[X_l \mapsto \varrho(t)]$$

$$\text{iff} \quad \begin{pmatrix} \mathcal{I}_{ec}(\Psi^\mu)(X_l) \;\supseteq\; \varrho(t), \quad \text{and} \\ \mathcal{I}_{ec}(\Psi^\mu)(X_i) \;\supseteq\; \varrho(X_i), \; i \neq l \end{pmatrix}$$

$$\text{iff} \quad \begin{pmatrix} \mathcal{I}_{sc}(\mathcal{X}^\mu_l) \;\supseteq\; \mathcal{I}_{sc}(t[X_1, \ldots, X_m \mapsto \mathcal{X}_1, \ldots, \mathcal{X}_m]), \quad \text{and} \\ \mathcal{I}_{sc}(\mathcal{X}^\mu_i) \;\supseteq\; \mathcal{I}_{sc}(\mathcal{X}_i), \; i \neq l \end{pmatrix}$$

Now consider case (iv), let ϱ be the set environment $\mathcal{A}(\{\rho \in \mathcal{I}_{ec}(\Psi^\mu) : \rho \rhd \mathit{cond}\})$, and write cond in the form $\mathit{conj}_1 \vee \cdots \vee \mathit{conj}_n$ where each conj_k is a conjunction of atomic program conditions. Again condition (b) implies that the constraints $\mathcal{X}_i \supseteq \{X_i : \mathit{defined}(\mathit{cond}) \wedge \bigwedge_{j=1..m} X_j \in \mathcal{X}^\lambda_j\}, i = 1..m$ are satisfied by $\mathcal{I}_{sc}$. Using reasoning identical to that in case (iii), it is easy to verify that condition (b) also implies that $\varrho(X_i) = \mathcal{I}_{sc}(\mathcal{X}_i)$, and hence

$$\varrho(t) = \mathcal{I}_{sc}(t[X_1, \ldots, X_n \mapsto \mathcal{X}_1, \ldots, \mathcal{X}_n])$$

for any program term t. Case (iv) can now be established using the following chain of reasoning.

$$
\begin{aligned}
&\mathcal{I}_{ec}(\Psi^\mu) \;\supseteq\; \mathcal{A}(\{\rho \in \varrho : \rho \models_\varrho \mathit{conj}_1 \vee \cdots \vee \mathit{conj}_n\}) \\
&\text{iff } \bigwedge_{i=1..m} \mathcal{I}_{ec}(\Psi^\mu)(X_i) \;\supseteq\; \{\rho(X_i) : \rho \in \varrho \wedge \rho \models_\varrho \mathit{conj}_1 \vee \cdots \vee \mathit{conj}_n\} \\
&\text{iff } \bigwedge_{i=1..m} \bigwedge_{k=1..n} \mathcal{I}_{ec}(\Psi^\mu)(X_i) \;\supseteq\; \{\rho(X_i) : \rho \in \varrho \wedge \rho \models_\varrho \mathit{conj}_k\} \\
&\text{iff } \bigwedge_{i=1..m} \bigwedge_{k=1..n} \mathcal{I}_{ec}(\Psi^\mu)(X_i) \;\supseteq\; \left\{\rho(X_i) : \rho \models_\varrho \mathit{conj}_k \;\wedge \bigwedge_{j=1..r} \rho(X_j) \in \varrho(X_j)\right\} \\
&\text{iff } \bigwedge_{i=1..m} \bigwedge_{k=1..n} \mathcal{I}_{sc}(\mathcal{X}_i^\mu) \;\supseteq\; \left\{\rho(X_i) : \rho \models_\varrho \mathit{conj}_k \;\wedge \bigwedge_{j=1..r} \rho(X_j) \in \mathcal{I}_{sc}(\mathcal{X}_j)\right\} \\
&\text{iff } \bigwedge_{i=1..m} \bigwedge_{k=1..n} \mathcal{I}_{sc}(\mathcal{X}_i^\mu) \;\supseteq\; \mathcal{I}_{sc}\left(\left\{X_i : \mathit{translate}(\mathit{conj}_k) \;\wedge \bigwedge_{j=1..r} X_j \in \mathcal{X}_j\right\}\right)
\end{aligned}
$$

The first four steps are straightforward: the first follows from the definition of $\mathcal{A}$, the second from the fact that $\rho \models_\varrho \mathit{cond}_1 \vee \cdots \vee \mathit{conj}_n$ iff $\rho \models_\varrho \mathit{cond}_k$ for some k, the third from the definition of $\rho \in \varrho$, and the fourth from $\mathcal{I}_{ec}(\Psi^\mu)(X_i) = \mathcal{I}_{sc}(\mathcal{X}_i^\mu)$ and $\varrho(X_i) = \mathcal{I}_{sc}(\mathcal{X}_i)$. The core part of the proof is the last step. To prove this, let σ abbreviate the renaming $[X_1, \ldots, X_m \mapsto \mathcal{X}_1, \ldots, \mathcal{X}_m]$, and consider the following chain of equalities:

$$\left\{\rho(X_i) : \begin{array}{ll} \rho \models_{\varrho} conj_k & \text{and} \\ \rho(X_j) \in \mathcal{I}_{sc}(\mathcal{X}_j) & \text{for } j = 1..r \end{array}\right\}$$

$$= \left\{\rho(X_i) : \begin{array}{ll} \rho(s) \in \varrho(t) \wedge \rho(t) \in \varrho(s) & \text{for all } s = t \text{ in } conj_k \\ \exists v\, (v \in \varrho(t) \wedge v \neq \varrho(s)) & \text{for all } \neg(s = t) \text{ in } conj_k \\ \exists v\, (v \in \varrho(s) \wedge v \neq \varrho(t)) & \text{for all } \neg(s = t) \text{ in } conj_k \\ \rho(s) \text{ has form } f(\cdots) & \text{for all } match_f(s) \text{ in } conj_k \\ \rho(s) \text{ is not of form } f(\cdots) & \text{for all } \neg(match_f(s)) \text{ in } conj_k \\ \rho(X_j) \in \mathcal{I}_{sc}(\mathcal{X}_j) & \text{for } j = 1..r \end{array}\right\}$$

$$= \left\{\rho(X_i) : \begin{array}{ll} \rho(s) \in \mathcal{I}_{sc}(\sigma(t)) \wedge \rho(t) \in \varrho(\sigma(s)) & \text{for all } s = t \text{ in } conj_k \\ \exists v\, (v \in \mathcal{I}_{sc}(\sigma(t)) \wedge v \neq \rho(s)) & \text{for all } \neg(s = t) \text{ in } conj_k \\ \exists v\, (v \in \mathcal{I}_{sc}(\sigma(s)) \wedge v \neq \rho(t)) & \text{for all } \neg(s = t) \text{ in } conj_k \\ \rho(s) \in \mathcal{I}_{sc}(f(\top, \ldots, \top)) & \text{for all } match_f(s) \text{ in } conj_k \\ \rho(s) \in \mathcal{I}_{sc}(\overline{f(\top, \ldots, \top)}) & \text{for all } \neg(match_f(s)) \text{ in } conj_k \\ \rho(X_j) \in \mathcal{I}_{sc}(\mathcal{X}_j) & \text{for } j = 1..r \end{array}\right\}$$

$$= \left\{\rho(X_i) : \begin{array}{ll} \rho(s) \in \mathcal{I}(se) & \text{for all } s \in se \text{ in } translate(conj_k) \\ \exists v \begin{pmatrix} v \neq \rho(s) \\ v \in \mathcal{I}(se) \end{pmatrix} & \text{for all } s \dagger se \text{ in } translate(conj_k) \\ \rho(X_j) \in \mathcal{I}_{sc}(\mathcal{X}_j) & \text{for } j = 1..r \end{array}\right\}$$

$$= \quad \mathcal{I}_{sc}\left(\left\{X_i : translate(conj_k) \ \wedge \bigwedge_{j=1..r} X_j \in \mathcal{X}_j\right\}\right)$$

The first equality in this chain is just the expansion of the definition of $\rho \models_{\varrho} conj_k$, noting that $cond_k$ is just a conjunction of atomic program conditions. The second equality holds because:

- it has previously been established that $\varrho(t) = \mathcal{I}_{sc}(\sigma(t))$ for any program term t (recall the at σ abbreviates $[X_1, \ldots, X_m \mapsto \mathcal{X}_1, \ldots, \mathcal{X}_m]$),
- $\mathcal{I}_{sc}(f(\top, \ldots, \top))$ is the set of all values of the form $f(\cdots)$, and so the condition $\rho(s) \in \mathcal{I}_{sc}(f(\top, \ldots, \top))$ is equivalent to the condition "$\rho(s)$ has form $f(\cdots)$", and
- $\mathcal{I}_{sc}(\overline{f(\top, \ldots, \top)})$ is the set of all values not of the form $f(\cdots)$, and so

the condition $\rho(s) \in \mathcal{I}_{sc}(\overline{f(\top,\ldots,\top)})$ is equivalent to the condition "$\rho(s)$ is not of form $f(\cdots)$".

The third equality follows immediately from the definition of *translate*, and finally, the fourth equality is just the definition of $\mathcal{I}_{sc}$.

This completes the proof of (6.13). The remainder of the proof uses (6.13) to show the following two properties:

(1) There is a model $\mathcal{I}_{sc}$ of $\mathcal{SC}_P$ such that $lm(\mathcal{EC}_P)(\Psi^\mu)(X_i) = \mathcal{I}_{sc}(\mathcal{X}_i^\mu)$.

(2) There is a model $\mathcal{I}_{ec}$ of $\mathcal{EC}_P$ such that $\mathcal{I}_{ec}(\Psi^\mu)(X_i) = lm(\mathcal{SC}_P)(\mathcal{X}_i^\mu)$.

These two properties imply the proposition because from the first it follows that

$$lm(\mathcal{EC}_P)(\Psi^\mu)(X_i) = \mathcal{I}_{sc}(\mathcal{X}_i^\mu) \supseteq lm(\mathcal{SC}_P)(\mathcal{X}_i^\mu),$$

and from the second it follows that

$$lm(\mathcal{EC}_P)(\Psi^\mu)(X_i) \subseteq \mathcal{I}_{ec}(\Psi^\mu)(X_i) = lm(\mathcal{SC}_P)(\mathcal{X}_i^\mu).$$

and together these imply that $lm(\mathcal{EC}_P)(\Psi^\mu)(X_i) = lm(\mathcal{SC}_P)(\mathcal{X}_i^\mu)$.

To prove property (1), define an interpretation $\mathcal{I}_{sc}$ of $\mathcal{SC}_P$ by

$$\mathcal{I}_{sc}(\mathcal{X}) \stackrel{\text{def}}{=} \begin{cases} lm(\mathcal{EC}_P)(\Psi^\mu)(X_i) & \text{if } \mathcal{X} \text{ is } \mathcal{X}_i^\mu \\ \mathcal{I}_{sc}(se) & \text{if } \mathcal{X} \text{ is not of form } \mathcal{X}_i^\mu \text{ and } \mathcal{X} \supseteq se \text{ is in } \mathcal{SC}_P \end{cases}$$

This is well defined because each set variable $\mathcal{X}$ appearing in $\mathcal{SC}_P$ is either of the form $\mathcal{X}_i^\mu$ or else it is one of the extra variables introduced in the translation of environment expressions $\Psi[cond]$ or $\Psi[X \mapsto t]$. In the latter case there is only one constraint of the form $\mathcal{X} \supseteq se$, and se contains only variables of the form $\mathcal{X}_i^\mu$. Hence this definition does yield an interpretation $\mathcal{I}_{sc}$ of $\mathcal{SC}_P$, and it only remains to verify that $\mathcal{I}_{sc}$ is a model of $\mathcal{SC}_P$. From the $\mathcal{I}_{sc}$, it is clear that the pair of models $lm(\mathcal{EC}_P)$ and $\mathcal{I}_{sc}$ satisfy conditions (a) and (b) of (6.13), and so (6.13) implies that $\mathcal{I}_{sc} \models \mathcal{SC}_P$.

To prove property (2), define an interpretation $\mathcal{I}_{ec}$ of $\mathcal{EC}_P$ by

$$\mathcal{I}_{ec}(\Psi^\mu)(X_i) \stackrel{\text{def}}{=} lm(\mathcal{SC}_P)(\mathcal{X}_i^\mu)$$

The fact that this does define an interpretation $\mathcal{I}_{ec}$ of $\mathcal{EC}_P$ is not obvious because an interpretation $\mathcal{I}_{ec}$ of $\mathcal{EC}_P$ must satisfy the following property for all μ:

$$\begin{array}{l} \text{if } \exists i(\mathcal{I}_{ec}(\Psi^\mu)(X_i) = \{\}) \\ \text{then } \forall i(\mathcal{I}_{ec}(\Psi^\mu)(X_i) = \{\}) \end{array}$$

where i ranges over $1..m$. This requires that $lm(\mathcal{SC}_P)$ satisfy:

$$\begin{array}{l} \text{if } \exists i(lm(\mathcal{SC}_P)(\mathcal{X}_i^\mu) = \{\}) \\ \text{then } \forall i(lm(\mathcal{SC}_P)(\mathcal{X}_i^\mu) = \{\}) \end{array} \tag{6.14}$$

Note that this property does not hold for an arbitrary model of $\mathcal{SC}_P$; its proof uses special properties of least models. Specifically, Proposition 17 proves that if $v \in lm(\mathcal{SC}_P)(\mathcal{X})$ then $\mathcal{SC}_P$ contains a constraint $\mathcal{X} \supseteq se$ such that $v \in lm(\mathcal{EC}_P)(se)$.

Now, to prove (6.14), suppose that $lm(\mathcal{SC}_P)(\mathcal{X}_r^\mu) \neq \{\}$ for some r, $1 \leq r \leq m$. Then there exists some $v \in lm(\mathcal{SC}_P)(\mathcal{X}_r^\mu)$, and so by Proposition 17 there must be some constraint $\mathcal{X}_r^\mu \supseteq se$ in $\mathcal{SC}_P$ such that $v \in lm(\mathcal{SC}_P)(se)$. Now, this constraint $\mathcal{X}_r^\mu \supseteq se$ could have been introduced via any of steps (i–v) of the construction of $\mathcal{SC}_P$. These possibilities are split into two cases.

In the first case, se is of the form $\{X_r : conj\}$ and $\mathcal{SC}_P$ contains the constraints $\mathcal{X}_i^\mu \supseteq \{X_i : conj\}$ for all $i = 1..m$ (this case includes all constraints introduced in steps (i), (iii), (iv) and (v)). If $v \in lm(\mathcal{SC}_P)(se)$, then by definition there exists an environment ρ such that $\rho(X_r) = v$ and $\rho \in lm(\mathcal{SC}_P)(conj)$, and it immediately follows that each $lm(\mathcal{SC}_P)(\{X_i : conj\})$ is non-empty, and so $lm(\mathcal{SC}_P)(\mathcal{X}_j^\mu) \neq \{\}$, $j = 1..m$.

In the second case, the constraint is introduced by step (ii), and $\mathcal{SC}_P$ contains the constraints $\mathcal{X}_j^\mu \supseteq \top$ for all $j = 1..m$. Clearly $lm(\mathcal{SC}_P)(\mathcal{X}_i^\mu)$ is equal to the set of all values, $j = 1..m$.

This completes the proof of (6.14), and so the mapping $\mathcal{I}_{ec}$ defined using $lm(\mathcal{SC}_P)$ is in fact an interpretation of $\mathcal{EC}_P$. It remains to show that $\mathcal{I}_{ec}$ is a model of $\mathcal{EC}_P$. Now, consider the constraints of the form $\mathcal{X} \supseteq se$ in $\mathcal{SC}_P$ where the variable $\mathcal{X}$ is not of the form $\mathcal{X}_i^\mu$. By the construction of $\mathcal{SC}_P$, if such a constraint is present, then it is the only lower bound for the variable $\mathcal{X}$. Hence it follows from Proposition 18 that $lm(\mathcal{SC}_P)(\mathcal{X}) = lm(\mathcal{SC}_P)(se)$, and so part (b) of (6.13) holds. Furthermore, part (a) holds because of the definition of $\mathcal{I}_{ec}$. Thus from (6.13), $\mathcal{I}_{ec} \models \mathcal{EC}_P$. This completes the proof of property (2), and so the proposition if proved. $\square$

Part II

Set Based Analysis

Having defined set based program approximation, we now show how this approximation may be computed. We begin by translating the environment constraints into set constraints such that the least set based model of the environment constraints corresponds to the least model of the set constraints. We then present an algorithm for solving these set constraints. The output of the algorithm is a representation of the least model of the input constraints that is *explicit* in the sense that structural properties of the model can be easily inferred. We conclude by describing a prototype implementation of the set constraint algorithm.

Chapter 7

Solving Set Constraints

In the previous chapter, the problem of computing the set based approximation of a program was reduced to the computing the least model of a collection of set constraints. This chapter presents algorithms for constructing an explicit representation of the least model of such set constraints. We first address the issue of explicit representation using regular term grammars. Then, we present a high level description of the set constraint algorithm in the form of a generic algorithm. The remainder of the chapter presents two progressively more complex instances of this algorithm. The second of these algorithms proves the main result of this thesis: set based program approximations are decidable and can be represented using regular term grammars.

7.1 Explicit Representation of $lm(\mathcal{C})$

We first address the issue of what kind of object is output by the set constraint algorithms and why this is an appropriate explicit representation. Recall that a model of a collection of set constraints is a mapping that associates a set of values to each set variable. Now, clearly such sets of values may be infinite. Therefore, to provide a description of the least model of a given collection of set constraints, the set constraint algorithm must output descriptions of sets of values, one for each set variable. The descriptions output by our algorithm are essentially regular term grammars.

Regular Term Grammars

A *regular term grammar* $\mathcal{G}$ consists of a set $NT_{\mathcal{G}}$ of non-terminals, a set $\Sigma_{\mathcal{G}}$ of function symbols, each with a unique arity, and a finite set $\mathcal{P}_{\mathcal{G}}$ of productions. To define productions, first define that a *term* is either a non-terminal or of the form $f(t_1, \ldots, t_n)$ where f is an n-ary symbol from Σ and each t_i is a term. Now, a *production* is of the form $nt \Rightarrow t$ such that nt is a non-terminal from $NT_{\mathcal{G}}$ and t is a term. Using the productions in $\mathcal{P}_{\mathcal{G}}$, a derivability relation on terms can be defined in the obvious way: $t_1 \Rightarrow t_2$ if there is a production $nt \Rightarrow t$ and t_2 is obtained from t_1 by replacing an occurrence of nt in t_1 by t. Let $\Rightarrow^\star$ denote the transitive reflexive closure of $\Rightarrow$. The language corresponding to a non-terminal nt, denoted $\mathcal{L}(nt)$, is defined as follows:

$$\mathcal{L}(nt) = \{t : \ nt \Rightarrow^\star t \text{ and } t \text{ does not contain non-terminal symbols}\}$$

For example, consider the grammar where non-terminals are *list* and *int*, the set of function symbols is $\{cons, nil, succ, zero\}$, and $\mathcal{P}$ consists of the productions

$$\begin{array}{rcl}
int & \Rightarrow & succ(int) \\
int & \Rightarrow & zero \\
list & \Rightarrow & cons(int, list) \\
list & \Rightarrow & nil
\end{array}$$

This grammar describes integers in successor-zero notation, and lists of integers. If S is a set of terms such that there is some regular term grammar $\mathcal{G}$ and non-terminal $nt \in NT_{\mathcal{G}}$ such that $S = \mathcal{L}(nt)$, then S is *regular set of* *terms* and $(\mathcal{G}, nt)$ is called a *description* of S. As another example, consider the productions

$$\begin{array}{lcl}
\mathcal{X} & \Rightarrow & cons(c, d) \\
\mathcal{X} & \Rightarrow & cons(d, c) \\
\mathcal{X} & \Rightarrow & cons(\mathcal{X}, \mathcal{X}) \\
\mathcal{X} & \Rightarrow & cons(cons(c, d), cons(d, c))
\end{array}$$

where $NT_{\mathcal{G}} = \{\mathcal{X}\}$ and $\Sigma_{\mathcal{G}} = \{cons, c, d\}$. Here $\mathcal{L}(\mathcal{X})$ consists of elements such as $cons(c, d)$ and $cons(d, c)$, as well as $cons(cons(c, d), cons(c, d))$ and $cons(cons(c, d), cons(d, c))$. Note that the last production is redundant.

Regular term grammars are essentially equivalent to tree[1] automata (the natural generalization of finite state automaton to terms). Tree automata can be divided into four classes, according to whether (a) they are deterministic or non-deterministic, and (b) whether they are root-to-frontier or frontier-to-root (that is, whether they start from the root and work towards the leaves of the tree, or vice-versa). The languages definable by non-deterministic root-to-frontier tree automata, non-deterministic frontier-to-root tree automata, deterministic frontier-to-root tree automata and regular term grammars are all equivalent. See [19] for further details. (Note that deterministic root-to-frontier tree automaton are strictly less powerful. In particular they correspond to regular term grammars where the productions for each non-terminal involve distinct outermost function symbols. Specifically, if $nt \Rightarrow t$ and $nt \Rightarrow t'$ are distinct productions, then t must be of the form $f(\cdots)$ and t' must be of the form $f'(\cdots)$ such that $f \neq f'$.)

Regular term grammars provide a representation of a set of terms that is explicit in the sense that there are straightforward polynomial-time algorithms to determine membership and emptiness. Furthermore, there are standard algorithms to compute the intersection, union, complementation and containment of regular term grammars. In short, a regular term grammar description of a set of terms provides a presentation of the set that exposes much of the internal structure of the set.

Regular term grammars can also be used explicitly represent a set constraint interpretation. For this purpose, it is convenient to identify non-terminals with set variables. Then, a grammar $\mathcal{G}$ *represents* an interpre-

[1]The notions of "term" and "tree" are completely interchangeable in this context. Terms are simply labeled trees and vice versa. The use of "terms" in term grammars and "trees" in tree automata is largely historical.

tation $\mathcal{I}$ if $\mathcal{L}(\mathcal{X}) = \mathcal{I}(\mathcal{X})$ for each set variable $\mathcal{X}$. Clearly, only certain interpretations can be represented in this way. Now, the essence of the set constraint algorithm is to input a collection $\mathcal{C}$ of set constraints and output a regular term grammar description of $lm(\mathcal{C})$. Note that there is no *a priori* reason why the least model of $\mathcal{C}$ should be representable by a regular term grammar. The set constraint algorithm in fact provides a constructive proof that this is always the case.

Explicit Form Constraints

Strictly speaking, the output of the set constraint algorithm is not a regular term grammar, but rather a restrictive class of set constraints that essentially corresponds to a regular term grammar. To define this class of constraints, we define the *atomic* set expressions, which are essentially set expressions that do not contain set operators of arity $n \geq 1$.

Definition 14 (Atomic Set Expressions) *A set expression is* <u>*atomic*</u> *if it is constructed from set variables, function symbols, the special constants* $\top$ *and* $\bot$*, and set operators of arity* 0. $\square$

For example, $f(f_{(1)}^{-1}(\mathcal{X}))$ and $\mathcal{X} \cap \mathcal{Y}$ are not atomic, but $\overline{c}, \mathcal{X}$ and $f(c, \mathcal{X})$ are. In what follows, we shall reserve the letter a for atomic set expressions. *Explicit form* constraints can now be defined as follows.

Definition 15 (Explicit Form Constraints) *A constraint* $\mathcal{X} \supseteq a$ *is in* <u>*explicit form*</u> *if* a *is an atomic set expression that is not a set variable. A collection* $\mathcal{C}$ *of constraints is in explicit form if each constraint in* $\mathcal{C}$ *is in explicit form.* $\square$

The output of the algorithm is a collection of explicit form constraints. Such constraints $\mathcal{C}$ can be viewed as a regular term grammar $\mathcal{G}_\mathcal{C}$ whose non-terminals are set variables and whose productions are $\{\mathcal{X} \Rightarrow t : (X \supseteq t) \in \mathcal{C}\}$. In general, this grammar $\mathcal{G}_\mathcal{C}$ is not a regular term grammar because of presence of constants such as $\top$, $\bot$ and $\overline{S}$. However $\mathcal{G}_\mathcal{C}$ is a straightforward extension of the notion of regular term grammar in which the constants $\top$, $\bot$ and $\overline{S}$ denote their usual sets. Specifically, extend the definition $\mathcal{L}$ to be

$$\mathcal{L}(\mathcal{X}) = \{v \in \mathcal{I}(t) : \quad \mathcal{X} \Rightarrow^{\star} t \text{ and } t \text{ does not contain non-terminals}\}$$

where $\mathcal{I}$ is some interpretation. Note the choice of $\mathcal{I}$ is immaterial since t does not contain non-terminals (set variables) and so $\mathcal{I}(t)$ is independent of $\mathcal{I}$. Given such a definition, $\mathcal{G}_{\mathcal{C}}$ and $lm(\mathcal{C})$ are equivalent in the following sense:

$$lm(\mathcal{C})(\mathcal{X}) = \mathcal{L}(\mathcal{X}) \text{ for each set variable } \mathcal{X} \text{ appearing in } \mathcal{C}.$$

Furthermore, even though $\mathcal{G}_{\mathcal{C}}$ is not quite a regular term grammar, it is still an explicit form in the sense that questions about membership, non-emptiness, etc., can easily be answered. Hence $\mathcal{C}$ can be viewed as explicit representation of its own least model because it is essentially a regular term grammar description of $lm(\mathcal{C})$. Constraints of the form $\mathcal{X} \supseteq a$ where a is a non-variable set expression containing only constants, function symbols and set variables are therefore call *explicit form constraints.* Such constraints play a key role in the set constraint algorithm.

Note that if $\Sigma_{\mathcal{G}}$ is finite, then $\mathcal{G}_{\mathcal{C}}$ can be treated as a regular term grammar because the values of the constants $\top$, $\bot$ and $\overline{S}$ can be represented by regular term grammars. A grammar for $\top$ can be constructed by treating $\top$ as a non-terminal with productions $\top \Rightarrow f(\top, \ldots, \top)$ for each $f \in \Sigma_{\mathcal{G}}$. A grammar for $\bot$ can be constructed by treating $\bot$ as a non-terminal with no productions. Similarly there is a straightforward construction for each constant $\overline{S}$. For example, if $\Sigma_{\mathcal{G}}$ is $\{c, d, f, g, h\}$, where c and d are constants, f and g are unary, and h is binary, then $\overline{\{f(\top), h(c, d)\}}$ can be represented by the regular term grammar:

$$\begin{array}{lll}
\overline{\{f(\top), h(c,d)\}} \Rightarrow c & \overline{c} \Rightarrow d & \overline{d} \Rightarrow c \\
\overline{\{f(\top), h(c,d)\}} \Rightarrow d & \overline{c} \Rightarrow f(\top) & \overline{d} \Rightarrow f(\top) \\
\overline{\{f(\top), h(c,d)\}} \Rightarrow g(\top) & \overline{c} \Rightarrow g(\top) & \overline{d} \Rightarrow g(\top) \\
\overline{\{f(\top), h(c,d)\}} \Rightarrow h(\overline{c}, \top) & \overline{c} \Rightarrow h(\top, \top) & \overline{d} \Rightarrow h(\top, \top) \\
\overline{\{f(\top), h(c,d)\}} \Rightarrow h(\top, \overline{d}) & &
\end{array}$$

Some Basic Algorithms on Explicit Form Constraints

We have already noted that regular term grammars provide a convenient representation of sets of terms because there are straightforward algorithms to determine membership and emptiness, as well as compute intersections and complementations, and these can be easily adapted to explicit form constraints. Since the set constraint algorithm presented in this chapter shall employ a number of these basic algorithms, we shall conclude this section by providing a brief outline of the necessary details. We stress that these basic algorithms are adaptations of known results (see, for example, [19]), and are included only for the sake of completeness.

We first consider the membership problem. That is, given an explicit form collection of constraints $\mathcal{C}$, a set variable $\mathcal{X} \in var(\mathcal{C})$ and a value v, we wish to determine whether $v \in lm(\mathcal{C})(\mathcal{X})$. Now, by treating $\mathcal{C}$ as a set root-to-frontier tree automaton, we can just use the definition of acceptance for root-to-frontier tree automaton to determine if $v \in lm(\mathcal{C})(a)$. However, this does not lead to a polynomial time algorithm because searching for an accepting computation requires trying all possible transitions at each computation step, and this can lead to exponential behavior. A polynomial time algorithm can be obtained by essentially treating $\mathcal{C}$ as a frontier-to-root automaton. Specifically, let S_v denote all subterms of the given value v. Now, to each value $v' \in S_v$ and each atomic set expression a in $\mathcal{C}$, associate a binary value $in(v', a)$. The intention is that $in(v', a)$ shall be true if $v' \in lm(\mathcal{C})(a)$. Now, it is easy to determine whether $v' \in lm(\mathcal{C})(a)$ if a is a ground atomic expression, and moreover this is independent of $\mathcal{C}$. For example, if Σ is $\{f, c\}$ then $f(c,c) \in lm(\mathcal{C})(f(c,c))$ and $f(c,c) \in lm(\mathcal{C})(\overline{c})$ but $c \notin lm(\mathcal{C})(f(c,c))$ and $c \notin lm(\mathcal{C})(\overline{c})$. Hence, initialize $in(v', a)$ so that $in(v', a)$ is *true* if a is ground and $v \in \mathcal{I}(a)$ for all interpretations $\mathcal{I}$, and false otherwise. Now, repeatedly update the values $nonempty(a)$ using the following steps:

- Set $in(v', a)$ to *true* if v' is the result of replacing each $\mathcal{X}$ in a by $v_{\mathcal{X}}$ and $in(v_{\mathcal{X}}, \mathcal{X})$ is *true* for each $\mathcal{X} \in var(a)$.
- Set $in(v', \mathcal{X})$ to *true* if $\mathcal{X} \supseteq a$ appears in $\mathcal{C}$ and $in(v', a)$ is *true*.

It is easy to verify that these updating steps terminate since v' ranges over subterms of v and a and $\mathcal{X}$ respectively range over all atomic set expressions and variables in $\mathcal{C}$. Moreover, on termination, $in(v, a)$ iff $v \in lm(\mathcal{C})(a)$.

We next deal with non-emptiness. That is, given an explicit form collection of constraints $\mathcal{C}$ and a set variable $\mathcal{X} \in var(\mathcal{C})$, we wish to determine whether $lm(\mathcal{C})(\mathcal{X}) = \{\}$. The basic structure of the algorithm is the same as the membership algorithm. Again, it is easy to determine whether $lm(\mathcal{C})(a) = \{\}$ if a is a ground atomic expression. For example, if Σ is $\{f, c\}$ then $lm(\mathcal{C})(f(c,c))$, $lm(\mathcal{C})(\top)$, $lm(\mathcal{C})(\overline{f(\top,\top)})$ are all non-empty, regardless of $\mathcal{C}$, whereas $lm(\mathcal{C})(\perp)$ and $lm(\mathcal{C})(\overline{\{f(\top,\top),c\}})$ are always empty. (Note that the emptiness of $\overline{\{f(\top,\top),c\}}$ depends on Σ.) Now, associate a boolean value *nonempty*(a) with each atomic expression a appearing in $\mathcal{C}$ and initialize these values so that *nonempty*(a) is *true* if a is a ground atomic expression that is non-empty in all interpretations. Repeatedly update the values *nonempty*(a) using the following steps:

- Set *nonempty*(a) to *true* if a is not ground and *nonempty*$(\mathcal{X})$ is *true* for each $\mathcal{X} \in var(a)$.
- Set *nonempty*$(\mathcal{X})$ to *true* if $\mathcal{C}$ contains $\mathcal{X} \supseteq a$ and *nonempty*(a) is *true*.

It is easy to verify that these updating steps terminate, and that on termination, *nonempty*(a) is *true* iff $lm(\mathcal{C})(a) \neq \{\}$, for all atomic set expressions a appearing in $\mathcal{C}$.

Finally, we deal with singleton sets. That is, given an explicit form collection of constraints $\mathcal{C}$ and a set variable $\mathcal{X} \in var(\mathcal{C})$, we wish to determine whether $lm(\mathcal{C})(a) = \{v\}$ for some value v. The algorithm for this property again follows the structure of the membership algorithm. It is easy to determine whether $lm(\mathcal{C})(a) = \{v\}$ for some value v if a is a ground atomic expression. For example, if Σ is $\{f, c\}$ then $lm(\mathcal{C})(c)$, $lm(\mathcal{C})(f(c,c))$ and $lm(\mathcal{C})(\overline{f(\top,\top)})$ are all singleton sets, but $lm(\mathcal{C})(\perp)$, $lm(\mathcal{C})(\top)$ and $lm(\mathcal{C})(\overline{c})$ are not. Now, consider mappings from each atomic expression a appearing in $\mathcal{C}$ into $\{v : v \text{ is a value}\} \cup \{\top, \perp\}$. Initially, let *singleton* denote the following mapping

$$singleton(a) = \begin{cases} \top & \text{if } a\ |\mathcal{I}(a)| \geq 2 \text{ for all interpretations } \mathcal{I} \\ v & \text{if } a\ \mathcal{I}(a) = \{v\} \text{ for all interpretations } \mathcal{I} \\ \perp & \text{otherwise} \end{cases}$$

where $|S|$ denotes the cardinality of the set S. Note that if a is atomic, then $\mathcal{I}(a)$ is independent of $\mathcal{I}$ iff a is ground. Now, repeatedly update the values *nonempty*(a) using the following steps:

- Set *singleton*(a) to v if a is not ground, *singleton*$(\mathcal{X})$ is not $\perp$ or $\top$, for each $\mathcal{X} \in var(a)$, and v is the result of replacing each $\mathcal{X}$ in a by $singleton(\mathcal{X})$.

- Set $singleton(a)$ to $\top$ if a is not ground and $singleton(\mathcal{X}) \neq \perp$ for each $\mathcal{X} \in var(a)$ and $singleton(\mathcal{Y}) = \top$ for some $\mathcal{Y} \in var(a)$.
- Set $singleton(\mathcal{X})$ to $singleton(a)$ if $\mathcal{X} \supseteq a$ appears in $\mathcal{C}$ and $singleton(\mathcal{X}) = \perp$.
- Set $singleton(\mathcal{X})$ to $\top$ if $\mathcal{X} \supseteq a$ appears in $\mathcal{C}$ and $singleton(a) \neq singleton(\mathcal{X})$.

It is easy to verify that these updating steps terminate, and that on termination, $singleton(a)$ is respectively $\perp$, v or $\top$ if $lm(\mathcal{C})(a) = \{\}$, $lm(\mathcal{C})(a) = \{v\}$ or $|lm(\mathcal{C})(a)| \geq 2$.

7.2 Overview of Algorithm

At a high level, the execution of the algorithm for solving set constraints can be summarized as follows. Starting with the input collection of constraints $\mathcal{C}_0$, a sequence of constraints $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_i, \ldots$ is constructed such that each collection $\mathcal{C}_i$ has essentially the same least model as $\mathcal{C}_0$ and is obtained from its predecessor $\mathcal{C}_{i-1}$ by adding some new constraints. The aim of adding these new constraints is to make the least model of $\mathcal{C}_i$ more "explicit". To formalize this notion, first recall that explicit form constraints are of the form $\mathcal{X} \supseteq a$ where a is a non-variable atomic expression, and that such constraints form the explicit representation that is output by the algorithm. Now, where $\mathcal{C}$ is a collection of constraints, let $explicit(\mathcal{C})$ denote the explicit form constraints in $\mathcal{C}$. In essence, $explicit(\mathcal{C})$ corresponds to what has already been computed about the least model of $\mathcal{C}$. Now, as the algorithm progresses, the constraints $\mathcal{C}_i$ become more explicit in the sense that

$$lm(explicit(\mathcal{C}_0)), lm(explicit(\mathcal{C}_1)), \ldots, lm(explicit(\mathcal{C}_i)), \ldots$$

is an increasing sequence of interpretations that converges towards $lm(\mathcal{C}_0)$. When $lm(explicit(\mathcal{C}_i))$ reaches $lm(\mathcal{C}_0)$, the algorithm terminates and outputs $explicit(\mathcal{C}_i)$.

The process of constructing $\mathcal{C}_i$ from $\mathcal{C}_{i-1}$ is defined by a collection of transformations. These transformations fall into two categories. First there are simplification transformations, which simplify expressions involving set operators to make the constraints more explicit. Such transformations are responsible for augmenting a constraint such as $\mathcal{X} \supseteq f_{(1)}^{-1}(f(c))$ with the more explicit, but equivalent constraint, $\mathcal{X} \supseteq c$. Second, there are substitution transformations, and these are responsible for performing substitutions so that simplification transformations can be applicable. For example, consider the constraints $\mathcal{X} \supseteq f_{(1)}^{-1}(\mathcal{Y})$, $\mathcal{Y} \supseteq f(c)$. Here $\mathcal{Y} \supseteq f(c)$ must be substituted into $\mathcal{X} \supseteq f_{(1)}^{-1}(\mathcal{Y})$ to obtain $\mathcal{X} \supseteq f_{(1)}^{-1}(f(c))$ before the projection can be simplified. Note that substitutions can be cyclic in the sense that substituting $\mathcal{Y} \supseteq f(\mathcal{Y})$ into $\mathcal{X} \supseteq f_{(1)}^{-1}(\mathcal{Y})$ involves replacing $\mathcal{Y}$ by $f(\mathcal{Y})$, yielding the constraint $\mathcal{X} \supseteq f_{(1)}^{-1}(f(\mathcal{Y}))$. This means that substitutions must be carefully controlled to ensure termination. Moreover, they must be done sufficiently often to ensure that simplification transformations can be applied when they are needed.

To facilitate this tradeoff, constraints are maintained by the algorithm in a special form described as follows.

Definition 16 (Standard Form) *A set expression is in standard form if it is either atomic or of the form $op(a_1, \ldots, a_n)$ such that each a_i is atomic and $op \in OP$. A collection of constraints is in standard form if each constraint is of the form $\mathcal{X} \supseteq se$ such that se is in standard form.*

Not only does the use of standard form help control the process of substitution, but it also reduces the number of cases that need to be considered at various points in the algorithm.

The rest of this chapter is organized as follows. First we show how constraints can be converted to standard form. Next, we present the core concepts of the set constraint algorithm in the form of a generic algorithm, parameterized by set operators and corresponding transformations. Abstract criteria are given for ensuring that a particular instance of the generic algorithm (specified by giving the set operators and transformations) is correct and terminates. The last two sections of this chapter describe two instances of the generic algorithm. The first deals with projections and intersections. The second generalizes the first by dealing with quantified set expressions, and is the core algorithm of this thesis. In particular, when input set constraints $\mathcal{SC}_P$ corresponding to a program P, this algorithm outputs an explicit representation of $lm(\mathcal{SC}_P)$.

7.3 Converting Constraints to Standard Form

Recall that standard form constraints are of the form $\mathcal{X} \supseteq se$ such that se is either an atomic set expression or of the form $op(a_1, \ldots, a_n)$ where op is a set operator and each a_i is atomic. As an alternative characterization, first define that the set expression se' is a *strict subexpression* of the set expression se if se' is a subexpression of se that is different from se. Then, $\mathcal{X} \supseteq se$ is in standard form if se is either atomic or of the form $op(\cdots)$ such that all strict subexpressions of se are atomic. Conversely, if a collection of constraints is not in standard form then it must contain a constraint $\mathcal{X} \supseteq se$ such that either se is (a) $se_1 \cup se_2$ or else (b) se has a strict subexpression se_{ns} that is not atomic. In case (b), we call the occurrence of the subexpression se_{ns} in se a *non-standard* occurrence. For example, the constraint $\mathcal{X} \supseteq f(op_1(c)) \cup op_1(op_2(\mathcal{X}))$ has one occurrence of $\cup$ and two non-standard occurrences.

Constraints can be converted to standard form by incrementally removing $\cup$ symbols and non-standard occurrences using the following two rewrite steps:

(1) Replace the constraint $\mathcal{X} \supseteq se_1 \cup se_2$ by the two constraints $\mathcal{X} \supseteq se_1$ and $\mathcal{X} \supseteq se_2$.

(2) If se has a non-standard occurrence se_{ns}, then replace the constraint $\mathcal{X} \supseteq se$ by the two constraints $\mathcal{X} \supseteq se'$, $\mathcal{Z} \supseteq se_{ns}$ where $\mathcal{Z}$ is a new set variable and se' is the result of replacing the occurrence of se_{ns} in se by $\mathcal{Z}$.

Each step, if applicable, rewrites constraints into a form that is closer to standard form in the sense that the number of union symbols or the number of non-standard occurrences decreases. It follows that the repeated application of these steps must terminate. When $\mathcal{C}$ is a collection of set constraints, let STANDARDIZE($\mathcal{C}$) denote the result of exhaustively applying steps (1) and (2). We now show that STANDARDIZE($\mathcal{C}$) produces a standard form collection of constraints that essentially has the same least model as $\mathcal{C}$. A formal statement of this must take into account the fact that the STANDARDIZE may introduce new variables. Hence, the preservation of the least model is defined with respect to a set of variables. Specifically, where $\mathcal{I}$ and $\mathcal{I}'$ are interpretations and var is a collection of variables, define that $\mathcal{I} =_{var} \mathcal{I}'$ if $\mathcal{I}(\mathcal{X}) = \mathcal{I}'(\mathcal{X})$ for each $\mathcal{X} \in var$. Then,

Proposition 21 (Standardize) *Let $\mathcal{C}$ be a collection of constraints. Then* STANDARDIZE($\mathcal{C}$) *is a standard form collection of constraints such that*

$$lm(\text{STANDARDIZE}(\mathcal{C})) =_{var(\mathcal{C})} lm(\mathcal{C}).$$

Proof: The proposition follows from repeated application of the following fact:

if $\mathcal{C}'$ is obtained from $\mathcal{C}$ by an application of step (1) or (2)
then $lm(\mathcal{C}) =_{var(\mathcal{C})} lm(\mathcal{C}')$.

To prove this fact, first consider the case where $\mathcal{C}'$ is obtained by an application of step (1). The difference between $\mathcal{C}$ and $\mathcal{C}'$ is that a constraint $\mathcal{X} \supseteq se_1 \cup se_2$ has been replace by two constraints $\mathcal{X} \supseteq se_1$ and $X \supseteq se_2$. However, it is clear that for all interpretations $\mathcal{I}$

$$\mathcal{I} \models \mathcal{X} \supseteq se_1 \cup se_2 \quad \text{iff} \quad \mathcal{I} \models \mathcal{X} \supseteq se_1 \wedge \mathcal{X} \supseteq \cup se_2$$

and it follows that $\mathcal{I} \models \mathcal{C}$ iff $\mathcal{I} \models \mathcal{C}'$. Hence $lm(\mathcal{C}) = lm(\mathcal{C}')$.

In the remaining case, $\mathcal{C}'$ is obtained from $\mathcal{C}$ by an application of step (2). The difference between $\mathcal{C}$ and $\mathcal{C}'$ is that a constraint $\mathcal{X} \supseteq se$ in $\mathcal{C}$ is replaced by two constraints $\mathcal{X} \supseteq se'$ and $\mathcal{Z} \supseteq se_{ns}$ in $\mathcal{C}'$ where se_{ns} is an occurrence of an expression in se and se' is the result of replacing this occurrence by the new variable $\mathcal{Z}$. It remains to prove that $lm(\mathcal{C})(\mathcal{Y}) = lm(\mathcal{C}')(\mathcal{Y})$ for all $\mathcal{Y} \in var(\mathcal{C})$ and this is done in two parts.

The first part shows that $lm(\mathcal{C})(\mathcal{Y}) \supseteq lm(\mathcal{C}')(\mathcal{Y})$ for all $\mathcal{Y} \in var(\mathcal{C})$. Let $\mathcal{I}$ be the interpretation that maps $\mathcal{Z}$ into $lm(\mathcal{C})(se_{ns})$ and agrees with $lm(\mathcal{C})$ on all other set variables. By definition, $\mathcal{I}(\mathcal{Z}) = \mathcal{I}(se_{ns})$, and from proposition 16 it follows that $\mathcal{I}(se) = \mathcal{I}(se')$. Since $\mathcal{C}$ does not contain the set variable $\mathcal{Z}$ and $\mathcal{I}$ agrees with $lm(\mathcal{C})$ except on $\mathcal{Z}$, it must be the case that $\mathcal{I} \models \mathcal{C}$. Now, $\mathcal{X} \supseteq se$ appears in $\mathcal{C}$, and so $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se) = \mathcal{I}(se')$. In summary, $\mathcal{I}$ is a model of $\mathcal{Z} \supseteq se_{ns}$ and $\mathcal{X} \supseteq se'$, and moreover, $\mathcal{I}$ is a model of all other constraints in $\mathcal{C}'$ since $\mathcal{I} \models \mathcal{C}$. Thus $\mathcal{I} \models \mathcal{C}'$. This implies that $\mathcal{I} \supseteq lm(\mathcal{C}')$, and so $lm(\mathcal{C})(\mathcal{Y}) = \mathcal{I}(\mathcal{Y}) \supseteq lm(\mathcal{C}')(\mathcal{Y})$ for all $\mathcal{Y} \in var(\mathcal{C})$. This completes the proof of the first part.

The second part shows that $lm(\mathcal{C})(\mathcal{Y}) \subseteq lm(\mathcal{C}')(\mathcal{Y})$. This is proved by showing that $lm(\mathcal{C}')$ is a model of $\mathcal{C}$. To prove this, first note that $\mathcal{Z} \supseteq se_{ns}$ is the only lower bound for $\mathcal{Z}$ in $\mathcal{C}'$, and so Proposition 18 implies that $lm(\mathcal{C}')(\mathcal{Z}) = lm(\mathcal{C}')(se_{ns})$. Proposition 16 can now be applied to show that

$lm(\mathcal{C}')(se') = lm(\mathcal{C}')(se)$. Moreover, since $lm(\mathcal{C}')$ is a model of $\mathcal{X} \supseteq se'$, it follows that $lm(\mathcal{C}')$ is a model of $\mathcal{X} \supseteq se$. Finally, $lm(\mathcal{C}')$ is a model of $\mathcal{C} - \{\mathcal{X} \supseteq se\}$ because it is a model of $\mathcal{C}'$. Hence $lm(\mathcal{C}')$ is a model of $\mathcal{C}$. This means that $lm(\mathcal{C}) \subseteq lm(\mathcal{C}')$, and this completes the proof of the second part. $\square$

7.4 The Generic Algorithm

We present a high level description of solving set constraints in the form of a generic set constraint algorithm. The reason for doing this is twofold. First, the details of the set constraint algorithm are substantial, and so the generic algorithm provides a way to explain the central ideas of the algorithm without introducing the many details that are necessary for its complete description. Second, the general structure of the algorithm appears to have wider application than the set constraints solved in this thesis. In particular, the set based analysis of a program involves writing set constraints using set operators corresponding to the semantic operations of the language – different languages require different set operations. The generic algorithm is an attempt to distill the concepts that are likely to be useful during the development of algorithms for the set operations arising in future work on set based analysis.

The generic algorithm is parameterized by a set OP of set operations that defines the class of set constraints on which it computes, and a set Δ of transformations, which define how these constraints may be simplified. The computation of the generic algorithm may be characterized as follows. Starting with an input collection of standard form constraints $\mathcal{C}_0$, the algorithm constructs a sequence of standard constraints $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_i, \ldots$ such that (i) for each i, $lm(\mathcal{C}_i) =_{var(\mathcal{C}_0)} lm(\mathcal{C}_0)$, (ii) each $\mathcal{C}_i$ is obtained from its predecessor $\mathcal{C}_{i-1}$ through the application of one of the transformations in Δ, and (iii) each $\mathcal{C}_i$ is more explicit than its predecessor in the sense that

$$lm(explicit(\mathcal{C}_0), lm(explicit(\mathcal{C}_1), \ldots, lm(explicit(\mathcal{C}_i)), \ldots$$

is an increasing sequence of interpretations that converges towards $lm(\mathcal{C}_0)$. The algorithm terminates when $lm(explicit(\mathcal{C}_i))$ reaches $lm(\mathcal{C}_0)$.

A *transformation* δ is a function that maps from and into (finite) collec-

input a collection $\mathcal{C}_0$ of set constraints in standard form;
$i := 0$;
while $\delta(\mathcal{C}_i) \not\subseteq \mathcal{C}_i$ for some transformation δ in Δ **do**
  $\mathcal{C}_{i+1} := \delta(\mathcal{C}_i) \cup \mathcal{C}_i$
  $i := i + 1$;
output the collection of explicit form constraints $explicit(\mathcal{C}_i)$;

Figure 7.1: **The Generic Algorithm**

tions of set constraints. The intention is that $\delta(\mathcal{C})$ denotes the constraints that should be added to $\mathcal{C}$ according to δ. An instance of the generic algorithm is defined by specifying a collection Δ of such transformations. These transformations are exhaustively applied as outlined in Figure 7.1.

We now address general conditions for establishing the correctness of the generic algorithm. First, each transformation must preserve the least model of the constraints. Since a transformation may introduce new variables, the preservation of the least model must be specified with respect to the initial variables in $\mathcal{C}_0$. Recall that $\mathcal{I} =_{var} \mathcal{I}'$ denotes that $\mathcal{I}(\mathcal{X}) = \mathcal{I}'(\mathcal{X})$ for each $\mathcal{X} \in var$. Using this notation, the required preservation of least model may be stated as $lm(\mathcal{C}_i) =_{var(\mathcal{C}_0)} lm(\mathcal{C}_0)$, for each i. This condition can be established if the transformations are sound in the following sense.

Definition 17 (Transformation Soundness) *A collection of transformations Δ is <u>sound on constraints $\mathcal{C}$</u> if $lm(\mathcal{C}) =_{var(\mathcal{C})} lm(\mathcal{C} \cup \delta(\mathcal{C}))$ for all transformations δ in Δ.* $\square$

It is easy to prove that if Δ is sound on each $\mathcal{C}_i$ constructed by the algorithm, then the least model is preserved.

Lemma 9 (Least Model) *If Δ is sound on each $\mathcal{C}_i$ constructed by the generic algorithm, then $lm(\mathcal{C}_i) =_{var(\mathcal{C}_0)} lm(\mathcal{C}_0)$ for each $\mathcal{C}_i$.*

Proof: Since Δ is sound on each $\mathcal{C}_i$, it follows that $lm(\mathcal{C}_i) =_{var(\mathcal{C}_i)} lm(\mathcal{C}_{i+1})$. Now clearly $var(\mathcal{C}_0) \subseteq var(\mathcal{C}_i)$ for each $\mathcal{C}_i$, and so $lm(\mathcal{C}_i) =_{var(\mathcal{C}_0)} lm(\mathcal{C}_{i+1})$. The lemma then follows by chaining these facts together. $\square$

We now address termination. Although termination proofs tend to be specific to a particular instance of the algorithm, some general observations can be made. An important part of proving termination involves establishing a bound on the number of atomic set expressions that can appear during the generic algorithms execution. Where S is a set of atomic set expressions, let $atomic(S)$ denote the superset of S that consists of all atomic set expressions that are subexpressions of elements of S. Also, where $\mathcal{C}$ is a collection of standard form constraints, define that $atomic(\mathcal{C})$ is the union of the following sets:

- $atomic(\{\mathcal{X}, a_1, \ldots, a_n\})$ for all constraints $\mathcal{X} \supseteq op(a_1, \ldots, a_n)$ appearing in $\mathcal{C}$, and
- $atomic(\{\mathcal{X}, a\})$ for all constraints $\mathcal{X} \supseteq a$ appearing in $\mathcal{C}$.

Now, a collection of transformations Δ is *atomically bounded* if, for each collection $\mathcal{C}_0$, there is a finite bound $\nabla(\mathcal{C}_0)$ such that when the generic algorithm is input $\mathcal{C}_0$, the cardinality of $atomic(\mathcal{C}_i)$ is bounded by $\nabla(\mathcal{C}_0)$ for each i. This bound is proved by establishing an algorithm invariant about the possible atomic set expressions that can appear during execution.

Typically, the remaining part of the termination proof involves using the bound on atomic set expressions to provide a bound on the number of all set expressions that can appear during execution. The main difficulty here is that the set of operations OP may not be finite.

Thus far, we have discussed termination and also conditions under which the least model is preserved by each step of the algorithm. Now, the algorithm does not output the final collection $\mathcal{C}_i$, but rather $explicit(\mathcal{C}_i)$, and so it remains to show that $explicit(\mathcal{C}_i)$ in fact describes $lm(\mathcal{C}_i)$. This requirement is essentially a completeness requirement on the transformations Δ. In other words, we need to show that the transformations are sufficiently powerful that when no new constraints are produced, all the information about the least model of $\mathcal{C}_i$ is in $explicit(\mathcal{C}_i)$. More formally, define that a collection of transformations Δ is *complete* if, whenever the exhaustive application of Δ terminates, the resulting $\mathcal{C}_i$ constructed by the algorithm is such that $lm(\mathcal{C}_i) = lm(explicit(\mathcal{C}_i))$. The following lemma immediately follows from these definitions.

Lemma 10 (Completeness) *If Δ is complete and the generic algorithm terminates after i iterations with output $\mathcal{C}$, then $lm(\mathcal{C}) = lm(\mathcal{C}_i)$.* ▯

Finally, on combining all of these observations:

Theorem 7 (Correctness of Generic Algorithm)
Let Δ be a collection of transformations that is complete. Let $\mathcal{C}_0$ be a collection of standard form constraints and suppose that the instance of the generic algorithm defined by Δ terminates on input $\mathcal{C}_0$ and outputs explicit form constraints $\mathcal{C}_{out}$. If Δ is sound on each $\mathcal{C}_i$ constructed by the algorithm then $lm(\mathcal{C}_{out}) =_{var(\mathcal{C})} lm(\mathcal{C})$.

Proof: Let $\mathcal{C}_i$ denote the collection of constraints constructed during the last iteration of the **while-do** loop. By Lemma 9, $lm(\mathcal{C}_i) =_{var(\mathcal{C})} lm(\mathcal{C})$. Finally, by lemma 10, $lm(\mathcal{C}') = lm(explicit(\mathcal{C}_i)) = lm(\mathcal{C}_i)$, and it follows that $lm(\mathcal{C}') =_{var(\mathcal{C})} lm(\mathcal{C})$. ▯

7.5 Intersection and Projection

We now describe an instance of the generic algorithm for solving set constraints involving projection and intersection. Specifically, the collection of operators for these set constraints, denoted OP_1, shall consist of projections $f_{(i)}^{-1}$ where f is an n-ary symbol from Σ and $1 \leq i \leq n$, and intersections $\cap_n$ where $n \geq 2$. We recall that the subscripts in intersections shall usually be omitted and an expression $\cap_n(se_1, \ldots, se_n)$ shall be written as $se_1 \cap \cdots \cap se_n$. For simplicity, we shall omit consideration of the constant symbol $\top$ from this section. Figure 7.2 gives an example collection of set constraints involving intersection and projection, along with the least model of the constraints. In this example, f and c are function symbols of arity 1 and 0 respectively, and f^n is again used to abbreviate n applications of f.

Now, the generic algorithm works on standard form constraints, and so the first step in solving these equations is to put them into standard form using the algorithm STANDARDIZE. The resulting standard form constraints appear in figure 7.3; let $\mathcal{C}_0$ denote these constraints.

Recall that explicit form constraints are of the form $\mathcal{X} \supseteq c$ such that c is a non-variable atomic set expression. This means that the constraints

$$
\begin{array}{lllllll}
\mathcal{X} & \supseteq & f(f(\mathcal{X})) \cup c & \quad & \mathcal{X} & \mapsto & \{c, f^2(c), f^4(c), f^6(c), \ldots\} \\
\mathcal{Y} & \supseteq & f(f(f(\mathcal{Y}))) \cup c & & \mathcal{Y} & \mapsto & \{c, f^3(c), f^6(c), f^9(c), \ldots\} \\
\mathcal{Z} & \supseteq & (\mathcal{X} \cap \mathcal{Y}) \cup f^{-1}_{(1)}(\mathcal{Y}) & & \mathcal{Z} & \mapsto & \{c, f^2(c), f^5(c) f^6(c), f^8(c), \ldots\}
\end{array}
$$

Figure 7.2: **Example Constraints and Their Least Model**

$$
\begin{array}{lllllll}
\mathcal{X} & \supseteq & f(f(\mathcal{X})) & \quad & \mathcal{X} & \supseteq & c \\
\mathcal{Y} & \supseteq & f(f(f(\mathcal{Y}))) & & \mathcal{Y} & \supseteq & c \\
\mathcal{Z} & \supseteq & (\mathcal{X} \cap \mathcal{Y}) & & \mathcal{Z} & \supseteq & f^{-1}_{(1)}(\mathcal{Y})
\end{array}
$$

Figure 7.3: **Constraints from Figure 7.2 Rewritten in Standard Form**

$\mathcal{X} \supseteq f(f(\mathcal{X}))$, $\mathcal{X} \supseteq c$, $\mathcal{Y} \supseteq f(f(f(\mathcal{X})))$ and $\mathcal{Y} \supseteq c$ in $\mathcal{C}_0$ are already in explicit form. However the constraints $\mathcal{Z} \supseteq (\mathcal{X} \cap \mathcal{Y})$ and $\mathcal{Z} \supseteq f^{-1}_{(1)}(\mathcal{Y})$ are not in explicit form and, moreover, $lm(explicit(\mathcal{C}_0)) \neq lm(\mathcal{C}_0)$. Hence the explicit constraints in $\mathcal{C}_0$ do not characterize the least model of $\mathcal{C}_0$.

As a first step towards making these constraints more explicit, consider $\mathcal{Z} \supseteq f^{-1}_{(1)}(\mathcal{Y})$. Now, the constraints for $\mathcal{Y}$ are $\mathcal{Y} \supseteq c$ and $\mathcal{Y} \supseteq f(f(f(\mathcal{Y})))$. The first says that $\mathcal{Y}$ must contain c and the second says that $\mathcal{Y}$ must contain all terms of the form $f(f(f(y)))$ such that $y \in \mathcal{Y}$. Combining $\mathcal{Z} \supseteq f^{-1}_{(1)}(\mathcal{Y})$ with $\mathcal{Y} \supseteq c$ implies that $\mathcal{Z}$ must contain $f^{-1}_{(1)}(c)$, but this is simply the empty set, and can be ignored. Combining $\mathcal{Z} \supseteq f^{-1}_{(1)}(\mathcal{Y})$ with $\mathcal{Y} \supseteq f(f(f(\mathcal{Y})))$ implies that $\mathcal{Z}$ must contain $f^{-1}_{(1)}(f(f(f(y))))$ for each $y \in \mathcal{Y}$, and this reduces to $f(f(y))$ for each $y \in \mathcal{Y}$. This can be expressed as the constraint $\mathcal{Z} \supseteq f(f(\mathcal{Y}))$. In essence, by substituting the explicit form constraints for $\mathcal{Y}$ into the right hand side of $\mathcal{Z} \supseteq f^{-1}_{(1)}(\mathcal{Y})$, and then simplify the resulting expression, we have obtained a new explicit form constraint for $\mathcal{Z}$. In so doing, we have made the constraints more explicit in the sense that the explicit form constraints now contain more information about the least model of the constraints.

Now consider the constraint $\mathcal{Z} \supseteq (\mathcal{X} \cap \mathcal{Y})$. Since $\mathcal{X} \supseteq c$ and $\mathcal{Y} \supseteq c$, it is clear that both $\mathcal{X}$ and $\mathcal{Y}$ must contain c, and so $\mathcal{Z}$ must contain c. This can be expressed by the constraint $\mathcal{Z} \supseteq c$. However there are other constraints for $\mathcal{X}$ and $\mathcal{Y}$. Consider $\mathcal{X} \supseteq f(f(\mathcal{X}))$ and $\mathcal{Y} \supseteq f(f(f(\mathcal{Y})))$. These constraint imply that $\mathcal{Z} \supseteq f(f(\mathcal{X})) \cap f(f(f(\mathcal{Y})))$. The expression $f(f(\mathcal{X})) \cap f(f(f(\mathcal{Y})))$ can be simplified into $f\,(f(\mathcal{X}) \cap f(f(\mathcal{Y})))$ and then further simplified into

$f(f(\mathcal{X} \cap f(\mathcal{Y})))$, and so the constraint

$$\mathcal{Z} \supseteq f(f(\mathcal{X} \cap f(\mathcal{Y})))$$

is obtained. However this constraint is not in standard form. It can be made into standard form by introducing a new variable, say $\mathcal{V}$, and writing $\mathcal{Z} \supseteq f(f(\mathcal{V}))$ and $\mathcal{V} \supseteq \mathcal{X} \cap f(\mathcal{Y})$. In other words, substituting explicit form constraints for $\mathcal{X}$ and $\mathcal{Y}$ (namely $\mathcal{X} \supseteq f(f(\mathcal{X}))$ and $\mathcal{Y} \supseteq f(f(f(\mathcal{Y}))))$ into $\mathcal{Z} \supseteq (\mathcal{X} \cap \mathcal{Y})$ and simplifying leads to new constraints $\mathcal{Z} \supseteq f(f(\mathcal{V}))$ and $\mathcal{V} \supseteq \mathcal{X} \cap f(\mathcal{Y})$. The first is in explicit form, but the second is not. In short, the substitution and simplification process has made the constraints more explicit. However, in the process we have introduced another constraint and so we must repeat the steps just outlined to simplify this new constraint.

In essence this process of substitution using explicit form constraints and then simplifying the resulting expressions forms the basis of the set constraint algorithms. However there are two difficulties. First the substitution process must be carefully controlled. For example, consider the constraint $\mathcal{X} \supseteq f(\mathcal{X})$. Now, this constraint could be substituted into itself to obtain $\mathcal{X} \supseteq f(f(\mathcal{X}))$ and subsequent substitution steps could lead to $\mathcal{X} \supseteq f^3(\mathcal{X})$, $\mathcal{X} \supseteq f^4(\mathcal{X})$ etc. Clearly this process can continue for ever. To prevent this, substitution must be restricted so that it does not increase the size of atomic set expressions. For example, substitutions into expressions such as $\mathcal{X} \cap \mathcal{Y}$ and $f_{(1)}^{-1}(\mathcal{Y})$ are allowed, but substitutions into expressions such as $f(\mathcal{Y})$ or $f(f(\mathcal{X}))$ are not allowed. Importantly, this restricted form of substitution is sufficient.

The second difficulty is that the operation of intersection produces new constraints that must be simplified. In particular it introduces new variables. The introduction of these new variables must be controlled using a special naming scheme to ensure that the algorithm terminates.

Transformations

We now give the details of the transformations. Each transformation takes a collection of standard form constraints $\mathcal{C}$ as input, and outputs one or more constraints. The first two transformations deal with substitution, and the third deals with simplifying projections.

Transformation 1 (Op-Substitution) *If $\mathcal{C}$ contains the two constraints $\mathcal{X} \supseteq op(a_1, \ldots, a_{i-1}, \mathcal{Y}, a_{i+1}, \ldots, a_n)$ and $\mathcal{Y} \supseteq a$ where a is atomic, then output $\mathcal{X} \supseteq op(a_1, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_n)$.* $\square$

Transformation 2 (Var-Substitution) *If $\mathcal{C}$ contains $\mathcal{X} \supseteq \mathcal{Y}$ and $\mathcal{Y} \supseteq a$ where a is atomic, then output $\mathcal{X} \supseteq a$.* $\square$

Transformation 3 (Projection) *If $\mathcal{C}$ contains $\mathcal{X} \supseteq f_{(i)}^{-1}(f(a_1, \ldots, a_n))$ and $lm(explicit(\mathcal{C}))(a_j) \neq \{\}$ for each $j \neq i$, then output $\mathcal{X} \supseteq a_i$.* $\square$

Note the condition $lm(explicit(\mathcal{C}))(a_j) \neq \{\}$ in the projection transformation. To see why this is needed, suppose that $\mathcal{C}$ consists of the single constraint $\mathcal{X} \supseteq f_{(2)}^{-1}(f(\mathcal{Y}, c))$. The least model of this constraint maps both $\mathcal{X}$ and $\mathcal{Y}$ into the empty set. However, if the side condition is not present on the projection transformation, then the constraint $\mathcal{X} \supseteq c$ would be added and this would alter the least model.

The final transformation deals with simplifying intersection. As noted before, one difficulty with simplifying intersections is that new variables need to be introduced, and this leads to termination problems. To overcome this problem, a special naming scheme is used for these new variables. Specifically, the introduced variables are of the form $\mathcal{V}_N$ where N is a finite subset of the atomic set expressions that appear in $\mathcal{C}_0$ (the input collection of constraints). Call such variables *intersection variables.* The intention is that an intersection variable $\mathcal{V}_{\{a_1,\ldots,a_n\}}$ should be equivalent to the expression $a_1 \cap \cdots \cap a_n$. All of the variables of form $\mathcal{V}_N$ are assumed to be distinct variables that do not appear in the input constraints $\mathcal{C}_0$. Define a function $\mathcal{N}$ that maps from such a variable back into the atomic set expressions it represents:

$$\mathcal{N}(a) \stackrel{\text{def}}{=} \begin{cases} N & \text{if } a \text{ is the intersection variable } \mathcal{V}_N. \\ \{a\} & \text{otherwise.} \end{cases}$$

The intersection transformation can now be defined.

Transformation 4 (Intersection) *If $\mathcal{C}$ contains $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$ such that for some $f \in \Sigma$, each a_i is of the form $f(a_{i,1}, \ldots, a_{i,n})$, then let $N_j = \bigcup_{i=1..m} \mathcal{N}(a_{i,j})$, $j = 1..n$, and output the constraints*

- $\mathcal{X} \supseteq f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n})$, *and*
- $\mathcal{V}_{N_j} = a_{1,j} \cap \cdots \cap a_{m,j}$, *for each* $\mathcal{V}_{N_j} \notin var(\mathcal{C})$. □

Strictly speaking, this transformation takes the input constraints $\mathcal{C}_0$ as an implicit parameter since the treatment of the variables $\mathcal{V}_N$ requires knowledge about $\mathcal{C}_0$, however for notational convenience this extra parameter shall be suppressed. We note that the treatment of the intersection variables $\mathcal{V}_N$ is analogous to the subset construction used in the conversion of a non-deterministic finite state automaton to a deterministic finite state automaton.

Let Δ_1 denote Transformations 1–4. (Strictly speaking, Transformations 1–4 are schemas for transformations, and Δ_1 consists of all instances of these four transformations schemas. We shall frequently blur this distinction.) Now, define that the *intersection-projection* algorithm inputs constraints $\mathcal{C}$ and first converts $\mathcal{C}$ into standard form constraints $\mathcal{C}_0$ and then exhaustively applies the transformations Δ_1 to C_0 as outlined by the generic algorithm. As an example of the execution of the algorithm, recall the example set constraints given in Figure 7.2. The execution of the algorithm on these constraints is traced in Figure 7.4 (the original constraints, in standard form, appear in the left hand column). In this example execution, we have given preference to lower numbered transformations when more than one transformation is applicable. The explicit form constraints are marked with an asterisk, and each new constraint is marked with either ①,②,③ or ④ to indicate the transformation used. We remark that, for efficiency reasons, it is appropriate to remove duplicate expressions when adding new constraints involving intersection. For example, instead of adding $\mathcal{Z} \supseteq c \cap c$, one could immediately simplify this into $\mathcal{Z} \supseteq c$. However, in this section we shall strive for a simple presentation of the algorithm, and so a number of straightforward modifications relating to efficiency, such as this one involving deletion of duplicate expressions in intersections, shall be omitted.

Correctness

The proof of the correctness of this algorithm follows the outline given for the generic algorithm in the previous section. We begin by proving an important invariant of the execution of the algorithm.

$$
\begin{aligned}
\mathcal{X} &\supseteq c^{\star} \\
\mathcal{X} &\supseteq f(f(\mathcal{X}))^{\star} \\
\mathcal{Y} &\supseteq c^{\star} \\
\mathcal{Y} &\supseteq f(f(f(\mathcal{Y})))^{\star} \\
\mathcal{Z} &\supseteq f_{(1)}^{-1}(\mathcal{Y}) \\
\mathcal{Z} &\supseteq \mathcal{X} \cap \mathcal{Y}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{Z} &\supseteq f_{(1)}^{-1}(c) && \textcircled{1} \\
\mathcal{Z} &\supseteq f_{(1)}^{-1}(f(f(f(\mathcal{Y})))) && \textcircled{1} \\
\mathcal{Z} &\supseteq c \cap c && \textcircled{1} \\
\mathcal{Z} &\supseteq c \cap f(f(f(\mathcal{Y}))) && \textcircled{1} \\
\mathcal{Z} &\supseteq f(f(\mathcal{X})) \cap c && \textcircled{1} \\
\mathcal{Z} &\supseteq f(f(\mathcal{X})) \cap f(f(f(\mathcal{Y}))) && \textcircled{1} \\
\mathcal{Z} &\supseteq f(f(\mathcal{Y}))^{\star} && \textcircled{3} \\
\mathcal{Z} &\supseteq c^{\star} && \textcircled{4} \\
\mathcal{Z} &\supseteq f(\mathcal{V}_{\{f(\mathcal{X}),f(f(\mathcal{Y}))\}})^{\star} && \textcircled{4} \\
\mathcal{V}_{\{f(\mathcal{X}),f(f(\mathcal{Y}))\}} &\supseteq f(\mathcal{X}) \cap f(f(\mathcal{Y})) && \textcircled{4} \\
\mathcal{V}_{\{f(\mathcal{X}),f(f(\mathcal{Y}))\}} &\supseteq f(\mathcal{V}_{\{\mathcal{X},f(\mathcal{Y})\}})^{\star} && \textcircled{4} \\
\mathcal{V}_{\{\mathcal{X},f(\mathcal{Y})\}} &\supseteq \mathcal{X} \cap f(\mathcal{Y}) && \textcircled{4} \\
\mathcal{V}_{\{\mathcal{X},f(\mathcal{Y})\}} &\supseteq c \cap f(\mathcal{Y}) && \textcircled{1} \\
\mathcal{V}_{\{\mathcal{X},f(\mathcal{Y})\}} &\supseteq f(f(\mathcal{X})) \cap f(\mathcal{Y}) && \textcircled{1} \\
\mathcal{V}_{\{\mathcal{X},f(\mathcal{Y})\}} &\supseteq f(\mathcal{V}_{\{f(\mathcal{X}),\mathcal{Y}\}})^{\star} && \textcircled{4} \\
\mathcal{V}_{\{f(\mathcal{X}),\mathcal{Y}\}} &\supseteq f(\mathcal{X}) \cap \mathcal{Y} && \textcircled{4} \\
\mathcal{V}_{\{f(\mathcal{X}),\mathcal{Y}\}} &\supseteq f(\mathcal{X}) \cap c && \textcircled{1} \\
\mathcal{V}_{\{f(\mathcal{X}),\mathcal{Y}\}} &\supseteq f(\mathcal{X}) \cap f(f(f(\mathcal{Y}))) && \textcircled{1} \\
\mathcal{V}_{\{f(\mathcal{X}),\mathcal{Y}\}} &\supseteq f(\mathcal{V}_{\{\mathcal{X},f(f(\mathcal{Y}))\}})^{\star} && \textcircled{4} \\
\mathcal{V}_{\{\mathcal{X},f(f(\mathcal{Y}))\}} &\supseteq \mathcal{X} \cap f(f(\mathcal{Y})) && \textcircled{4} \\
\mathcal{V}_{\{\mathcal{X},f(f(\mathcal{Y}))\}} &\supseteq c \cap f(f(\mathcal{Y})) && \textcircled{1} \\
\mathcal{V}_{\{\mathcal{X},f(f(\mathcal{Y}))\}} &\supseteq f(f(\mathcal{X})) \cap f(f(\mathcal{Y})) && \textcircled{1} \\
\mathcal{V}_{\{\mathcal{X},f(f(\mathcal{Y}))\}} &\supseteq f(\mathcal{V}_{\{f(\mathcal{X}),f(\mathcal{Y})\}})^{\star} && \textcircled{4} \\
\mathcal{V}_{\{f(\mathcal{X})),f(\mathcal{Y})\}} &\supseteq f(\mathcal{X}) \cap f(\mathcal{Y}) && \textcircled{4} \\
\mathcal{V}_{\{f(\mathcal{X})),f(\mathcal{Y})\}} &\supseteq f(\mathcal{V}_{\{\mathcal{X},\mathcal{Y}\}})^{\star} && \textcircled{4} \\
\mathcal{V}_{\{\mathcal{X},\mathcal{Y}\}} &\supseteq \mathcal{X} \cap \mathcal{Y} && \textcircled{4} \\
\mathcal{V}_{\{\mathcal{X},\mathcal{Y}\}} &\supseteq c \cap c && \textcircled{1} \\
\mathcal{V}_{\{\mathcal{X},\mathcal{Y}\}} &\supseteq c \cap f(f(f(\mathcal{Y}))) && \textcircled{1} \\
\mathcal{V}_{\{\mathcal{X},\mathcal{Y}\}} &\supseteq f(f(\mathcal{X})) \cap c && \textcircled{1} \\
\mathcal{V}_{\{\mathcal{X},\mathcal{Y}\}} &\supseteq f(f(\mathcal{X})) \cap f(f(f(\mathcal{Y}))) && \textcircled{1} \\
\mathcal{V}_{\{\mathcal{X},\mathcal{Y}\}} &\supseteq c^{\star} && \textcircled{4} \\
\mathcal{V}_{\{\mathcal{X},\mathcal{Y}\}} &\supseteq f(\mathcal{V}_{\{f(\mathcal{X}),f(f(\mathcal{Y}))\}})^{\star} && \textcircled{4}
\end{aligned}
$$

Figure 7.4: **Example Algorithm Execution**

Invariant 1 (Atomic Set Expression Invariant) *A collection of constraints $\mathcal{C}$ satisfies the atomic set expression invariant if each element of $atomic(\mathcal{C})$ either*

- *appears in $atomic(\mathcal{C}_0)$, or*
- *is an intersection variable or of the form $f(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ such that $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are intersection variables and f is a function symbol appearing in $\mathcal{C}_0$.* $\Box$

Proposition 22 *Each $\mathcal{C}_i$ constructed by the algorithm satisfies the atomic set expression invariant.*

Proof: Clearly $\mathcal{C}_0$ satisfies the atomic set expression invariant. Now suppose that $\mathcal{C}_{i-1}$ satisfies this invariant. The constraints $\mathcal{C}_i$ are defined to be $\delta(\mathcal{C}_{i-1}) \cup \mathcal{C}_{i-1}$. If δ is either Transformation 1, 2 or 3, then the only difference between $\mathcal{C}_i$ and $\mathcal{C}_{i-1}$ is that $\mathcal{C}_i$ contains a new constraint of the form $\mathcal{X} \supseteq a$ such that a is an atomic set expression, and moreover a appears in $\mathcal{C}_{i-1}$. It follows that $atomic(\mathcal{C}_i) \subseteq atomic(\mathcal{C}_{i-1})$, and so $\mathcal{C}_i$ satisfies the atomic set expression invariant.

The remaining case is where δ is Transformation 4. In this case, the difference between $\mathcal{C}_i$ and $\mathcal{C}_{i-1}$ is that $\mathcal{C}_i$ contains a number of new constraints. These new constraints are either of the form $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_n$ where the a_i are atomic set expressions that appear in $\mathcal{C}_{i-1}$, or $\mathcal{X} \supseteq f(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ where the $\mathcal{X}_i$ are intersection variables. Again $\mathcal{C}_i$ satisfies the atomic set expression invariant. $\Box$

The atomic set expression invariant proves two things. First, it verifies that the specification of Transformation 4 and the use of the naming scheme $\mathcal{V}_N$ is consistent. In particular, it shows that elements of the sets N_j constructed in Transformation 4 are either atomic set expressions from $\mathcal{C}_0$ or else atomic set expressions that appear in some set N such that $\mathcal{V}_N$ is a previously constructed intersection variable. It follows that all variables introduced by this transformation are of the form $\mathcal{V}_N$ such that $N \subseteq atomic(\mathcal{C}_0)$.

Second, it shows that Δ_1 is atomically bounded. This is because the set $atomic(\mathcal{C}_0)$ is finite and so there are only a finite number of variables of the form $\mathcal{V}_N$ such that $N \subseteq atomic(\mathcal{C}_0)$. It follows that there are only a finite number of expressions of the form $f(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ where each $\mathcal{X}_n$ is an intersection variable. Hence, the cardinality of each $\mathcal{C}_i$ is bounded by $K + F.(2^K)^n$ where K is the cardinality of $\mathcal{C}_0$, F is the number of function symbols appearing in $\mathcal{C}_0$, and n is the maximum arity of function symbols appearing in $\mathcal{C}_0$.

We next address the soundness of Δ_1. The soundness of Transformations 1, 2 and 3 is straightforward.

Proposition 23 $lm(\mathcal{C}) = lm(\mathcal{C} \cup \delta(\mathcal{C}))$ *where* δ *is one of Transformations 1–3.*

Proof: Let δ be one of Transformations 1–3. It is sufficient to prove that $lm(\mathcal{C})$ is a model of $\delta(\mathcal{C})$. Let $\mathcal{I}$ be $lm(\mathcal{C})$, and first consider Transformations 1 and 2. In this case $\mathcal{C}$ contains $\mathcal{X} \supseteq se$ and $\mathcal{Y} \supseteq a$, and $\delta(\mathcal{C})$ contains the constraint $\mathcal{X} \supseteq se'$ such that se' is the result of replacing an occurrence of $\mathcal{Y}$ in se by a. Since $\mathcal{I}$ is a model of $\mathcal{C}$, it follows that $\mathcal{I}(\mathcal{Y}) \supseteq \mathcal{I}(a)$ and $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$. Hence, by Proposition 19, $\mathcal{I}(se) \supseteq \mathcal{I}(se')$, and it follows that $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se')$. This completes the proof for this case.

In the case of Transformation 3, $\mathcal{C}$ contains $\mathcal{X} \supseteq f_{(i)}^{-1}(f(a_1, \ldots, a_n))$ and $\mathcal{X} \supseteq a_i$ is output. The proof proceeds by showing that $\mathcal{I}$ is a model of $\mathcal{X} \supseteq a_i$. Let $v_i \in \mathcal{I}(a_i)$. Now, since $explicit(\mathcal{C}) \subseteq \mathcal{C}$, it follows that $lm(explicit(\mathcal{C})) \subseteq \mathcal{I}$. Hence if $lm(explicit(\mathcal{C}))(a_j)$ is non-empty for each $j \neq i$, then it must be the case that for each a_j, $j \neq i$, there exists a $v_j \in \mathcal{I}(a_j)$. It follows that $f(v_1, \ldots, v_n) \in \mathcal{I}(f(a_1, \ldots, a_n))$, and so $v_i \in \mathcal{I}(f_{(i)}^{-1}(f(a_1, \ldots, a_n)))$. But $\mathcal{I}$ satisfies $\mathcal{X} \supseteq f_{(i)}^{-1}(f(a_1, \ldots, a_n))$, and so $v_i \in \mathcal{I}(\mathcal{X})$. This completes the proof that $\mathcal{I}$ is a model of $\mathcal{X} \supseteq a_i$. $\square$

The correctness of Transformation 4 is somewhat more difficult to prove since it does not in general preserve the least model of the constraints. This is because the correctness of this transformation relies on properties of the intersection variables $\mathcal{V}_{N_i}$. For example, consider the constraints

$$\mathcal{X} \supseteq f(\mathcal{X}) \cap f(c) \quad \text{and} \quad \mathcal{V}_{\{\mathcal{X},c\}} \supseteq c.$$

The intersection transformation adds the constraints $\mathcal{X} \supseteq f(\mathcal{V}_{\{\mathcal{X},c\}})$ and $\mathcal{V}_{\{\mathcal{X},c\}} \supseteq \mathcal{X} \cap c$. Clearly this does not preserve the least model of the constraints: before the transformation the least model maps $\mathcal{X}$ into the empty set, and after the transformation the least model maps $\mathcal{X}$ into $\{f(c)\}$.

Instead, we must first establish some properties of the intersection variables introduced by the algorithm. Where N is a finite set of atomic set expressions $\{a_1, \ldots, a_n\}$, let $(\bigcap N)$ denote $a_1 \cap \cdots \cap a_n$. Now, consider the following invariant.

Invariant 2 (Intersection Variable Invariant) *A collection of constraints $\mathcal{C}$ satisfies the intersection variable invariant if $\mathcal{V}_N \in var(\mathcal{C})$ implies that $lm(\mathcal{C}) \models \mathcal{V}_N = (\bigcap N)$.* $\square$

It is straightforward to verify that this invariant is preserved by Transformations 1, 2 and 3 because these transformation do not introduce any new intersection variables. Specifically:

Proposition 24 *If $\mathcal{C}$ satisfies the intersection variable invariant then $\mathcal{C} \cup \delta(\mathcal{C})$ satisfies the intersection variable invariant where δ is one of Transformations 1–3.*

Proof: Let $\mathcal{C}$ be a collection of constraints that satisfy the intersection variable invariant and let δ be one of Transformations 1–3. Now, if $\mathcal{V}_N$ is an intersection variable in $\mathcal{C} \cup \delta(\mathcal{C})$ then by the definition of Transformations 1, 2 and 3, it is clear that $\mathcal{V}_N$ must appear in $\mathcal{C}$. Since $\mathcal{C}$ satisfies the intersection variable invariant, $lm(\mathcal{C}) \models \mathcal{V}_N = (\bigcap N)$. Also, by Proposition 23, $lm(\mathcal{C}) = lm(\mathcal{C} \cup \delta(\mathcal{C}))$. It is immediate that $lm(\mathcal{C} \cup \delta(\mathcal{C})) \models \mathcal{V}_N = (\bigcap N)$. $\square$

The next proposition deals with the Transformation 4. It not only shows that the transformation preserves the intersection variable invariant, but also that the transformation is sound when applied to constraints that satisfy this invariant.

Proposition 25 *If $\mathcal{C}$ satisfies the intersection variable invariant and δ is Transformation 4, then*

(1) $lm(\mathcal{C}) =_{var(\mathcal{C})} lm(\mathcal{C} \cup \delta(\mathcal{C}))$, and

(2) $\mathcal{C} \cup \delta(\mathcal{C})$ satisfies the intersection variable invariant.

Proof: Recall that if Transformation 4 is applied, then $\mathcal{C}$ contains a constraint of the form $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$ where, for some $f \in \Sigma$, each a_i is of the form $f(a_{i,1}, \ldots, a_{i,n})$, and $\delta(\mathcal{C})$ consists of the constraints

$$\begin{array}{rcll} \mathcal{X} & \supseteq & f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n}) & \text{and} \\ \mathcal{V}_{N_j} & \supseteq & a_{1,j} \cap \cdots \cap a_{m,j} & \text{for each } \mathcal{V}_{N_j} \text{ that does not appear in } \mathcal{C} \end{array}$$

where $N_j = \bigcup_{i=1..m} \mathcal{N}(a_{i,j})$, $j = 1..n$.

The core part of the proof shows that the interpretation $\mathcal{I}$ defined by

$$\mathcal{I}(\mathcal{X}) = \begin{cases} lm(\mathcal{C})(a_{1,j} \cap \cdots \cap a_{m,j}) & \text{if } \mathcal{X} \notin var(\mathcal{C}) \text{ and } \mathcal{X} \text{ is } \mathcal{V}_{N_j} \\ lm(\mathcal{C})(\mathcal{X}) & \text{otherwise} \end{cases}$$

is the least model of $\mathcal{C} \cup \delta(\mathcal{C})$. By definition, it is clear that $\mathcal{I}$ is a model of

$$\delta(\mathcal{C}) - \{\mathcal{X} \supseteq f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n})\}.$$

To show that $\mathcal{I}$ is also a model of $\mathcal{X} \supseteq f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n})$, consider the following property of $\mathcal{I}$ for each i and j:

$$\mathcal{I}(a_{i,j}) = \mathcal{I}(\textstyle\bigcap \mathcal{N}(a_{i,j})) \tag{7.15}$$

If $a_{i,j}$ is not an intersection variable, then $\mathcal{N}(a_{i,j})$ is just $\{a_{i,j}\}$ and so (7.15) is trivially true. On the other hand, if $a_{i,j}$ is an intersection variable, then since it appears in $\mathcal{C}$, (7.15) follows from the assumption that $\mathcal{C}$ satisfies the intersection variable invariant. Using equation (7.15), the following chain of equalities can be established

$$\begin{array}{rcl} \mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j}) & = & \mathcal{I}(a_{1,j}) \cap \cdots \cap \mathcal{I}(a_{m,j}) \\ & = & \mathcal{I}\left(\bigcap \mathcal{N}(a_{1,j})\right) \cap \cdots \cap \mathcal{I}\left(\bigcap \mathcal{N}(a_{m,j})\right) \\ & = & \mathcal{I}\left(\bigcap \mathcal{N}(a_{1,j}) \cup \cdots \cup \mathcal{N}(a_{m,j})\right) \\ & = & \mathcal{I}\left(\bigcap N_j\right) \end{array}$$

and this proves that, for all i, $\mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j}) = \mathcal{I}(\cap N_j)$. Now, if $\mathcal{V}_{N_j}$ is introduced by δ, then $\mathcal{I}(\mathcal{V}_{N_j}) = \mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j})$ by definition of $\mathcal{I}$. If $\mathcal{V}_{N_j}$ is not introduced by δ, then it appears in $\mathcal{C}$ and so $\mathcal{I}(\mathcal{V}_{N_j}) = \mathcal{I}(\bigcap N_j)$ because $\mathcal{C}$ satisfies the intersection variable invariant. Combining these two cases with $\mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j}) = \mathcal{I}(\cap N_j)$ proves that, for $j = 1..n$,

$$\mathcal{I}(\mathcal{V}_{N_j}) \;=\; \mathcal{I}\left(\textstyle\bigcap N_j\right) \;=\; \mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j}) \tag{7.16}$$

Using this equality, it is easy to see that

$$\begin{aligned}
&\mathcal{I}\left(f(a_{1,1},\ldots,a_{1,n}) \cap \cdots \cap f(a_{m,1},\ldots,a_{m,n})\right)\\
=\;& \mathcal{I}\left(f(a_{1,1} \cap \cdots \cap a_{m,1},\ldots,a_{1,n} \cap \cdots \cap a_{m,n})\right)\\
=\;& \mathcal{I}\left(f(\mathcal{V}_{N_1},\ldots,\mathcal{V}_{N_m})\right).
\end{aligned}$$

and since $\mathcal{I}$ is a model of $\mathcal{X} \supseteq f(a_{1,1},\ldots,a_{1,n}) \cap \cdots \cap f(a_{m,1},\ldots,a_{m,n})$ it follows that $\mathcal{I}$ is a model of $\mathcal{X} \supseteq \mathcal{I}\left(\mathcal{V}_{N_1},\ldots,\mathcal{V}_{N_m}\right)$. This concludes the proof that $\mathcal{I}$ is a model of $\mathcal{C} \cup \delta(\mathcal{C})$, and so $\mathcal{I} \supseteq lm(\mathcal{C} \cup \delta(\mathcal{C}))$.

$\mathcal{I}$ is not only a model of $\mathcal{C} \cup \delta(\mathcal{C})$, it is in fact the least model. To see this, let $\mathcal{I}'$ be an arbitrary model of $\mathcal{C} \cup \delta(\mathcal{C})$. If $\mathcal{X}$ is a variable that appears in $\mathcal{C}$, then $\mathcal{I}'(\mathcal{X}) \supseteq \mathcal{I}(\mathcal{X})$ because $\mathcal{I}' \supseteq lm(\mathcal{C})$ and $\mathcal{I}(\mathcal{X}) = lm(\mathcal{C})(\mathcal{X})$. If $\mathcal{X}$ is one of the variables $\mathcal{V}_{N_j}$ introduced at this step, then consider the following chain:

$$\mathcal{I}'(\mathcal{V}_{N_j}) \quad \supseteq \quad \mathcal{I}'(a_{1,j} \cap \cdots \cap a_{m,j}) \quad \supseteq \quad \mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j}) \quad = \quad \mathcal{I}(\mathcal{V}_{N_j}).$$

The first containment follows because $\mathcal{I}'$ is a model of $\delta(\mathcal{C})$. The second is because $a_{1,j} \cap \cdots \cap a_{m,j}$ contains only variables from $\mathcal{C}$, and it has just been proved that $\mathcal{I}'(\mathcal{X}) \supseteq \mathcal{I}(\mathcal{X})$ for variables $\mathcal{X} \in var(\mathcal{C})$. The final equality follows from (7.16). Hence $\mathcal{I}'(\mathcal{X}) \supseteq \mathcal{I}(\mathcal{X})$ for all variables $\mathcal{X}$, and so $lm(\mathcal{C} \cup \delta(\mathcal{C})) \supseteq \mathcal{I}$. Combining this with $\mathcal{I} \supseteq lm(\mathcal{C} \cup \delta(\mathcal{C}))$ proves that $\mathcal{I} = lm(\mathcal{C} \cup \delta(\mathcal{C}))$.

To complete the proof, note that by definition $\mathcal{I}$ agrees with $lm(\mathcal{C})$ on $var(\mathcal{C})$, and so $lm(\mathcal{C}) =_{var(\mathcal{C})} lm(\mathcal{C} \cup \delta(\mathcal{C}))$. This proves part (1) of the proposition. To prove part (2), we need to show that $\mathcal{I}(\mathcal{V}_N) = \mathcal{I}(\bigcap N)$ for all intersection variables $\mathcal{V}_N$ appearing in $\mathcal{C}$. If $\mathcal{V}_N \in var(\mathcal{C})$, then the fact that $\mathcal{C}$ satisfies the intersection invariant implies that

$$\mathcal{I}(\mathcal{V}_N) = lm(\mathcal{C})(\mathcal{V}_N) = lm(\mathcal{C})(\textstyle\bigcap N) = \mathcal{I}(\textstyle\bigcap N)$$

On the other hand, if $\mathcal{V}_N$ is introduced by $\delta(\mathcal{C})$, then $\mathcal{I}(\mathcal{V}_N) = \mathcal{I}(\bigcap N)$ follows from (7.16). $\square$

Combining these propositions proves the necessary soundness property of Δ_1.

Lemma 11 (Soundness) Δ_1 *is sound on each* $\mathcal{C}_i$ *constructed by the algorithm.*

Proof: First, it is clear that each $\mathcal{C}_i$ constructed by the algorithm satisfies the intersection variable invariant. This is because $\mathcal{C}_0$ satisfies the invariant (since it does not contain any intersection variables) and each transformation preserves the invariant (see Propositions 24 and 25). It remains to show that $lm(\mathcal{C}_i) =_{var(\mathcal{C}_i)} lm(\mathcal{C}_i \cup \delta(\mathcal{C}_i))$ for each $\mathcal{C}_i$ and δ. This is easy since if δ is Transformation 1, 2 or 3, then $lm(\mathcal{C}_i) = lm(\mathcal{C}_i \cup \delta(\mathcal{C}_i))$ by Proposition 23, and if δ is Transformation 4, then $lm(\mathcal{C}_i) =_{var(\mathcal{C}_i)} lm(\mathcal{C}_i \cup \delta(\mathcal{C}_i))$ by Proposition 25. $\square$

As an aside, note that the correctness of this lemma makes the implicit assumption that the intersection variables $\mathcal{V}_N$ introduced during the execution of the algorithm are distinct from $var(\mathcal{C}_0)$. Clearly this can always be done by choosing the intersection variables introduced in Transformations 4 from $\textsc{var} - var(\mathcal{C}_0)$. Strictly speaking, Δ should be parameterized by $var(\mathcal{C}_0)$ to denote this dependence on $var(\mathcal{C}_0)$.

So far, we have prove that Δ_1 is atomically bounded (this follows from the atomic set expression invariant) and sound. It remains to prove termination and completeness.

Lemma 12 (Termination) *Let* $\mathcal{C}_0$ *be a collection of constraints in standard form. Then the instance of the generic algorithm defined by* Δ_1 *terminates on* $\mathcal{C}_0$.

Proof: Each $\mathcal{C}_i$ constructed by the algorithm is in standard form. Moreover, by inspection of the transformations, it is clear that *op* appears in $\mathcal{C}_i$ iff it appears in $\mathcal{C}_0$. Hence, each constraint in each $\mathcal{C}_i$ must be of one of the following forms:

- $\mathcal{X} \supseteq a$ where a is an atomic set expression;
- $\mathcal{X} \supseteq f_{(i)}^{-1}(a)$ where a is atomic and $f_{(i)}^{-1}$ appears in $\mathcal{C}_0$, or
- $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_n$ where $\mathcal{C}_0$ contains an expression of the form $a'_1 \cap \cdots \cap a'_n$.

Combining this with the atomic set expression invariant (proved in Proposition 22) proves that there are only a finite number of different constraints that may be constructed by the algorithm. Since the collections $\mathcal{C}_i$ are monotonically increasing during the algorithm's execution, it follows that for some $i \geq 1$, $\mathcal{C}_i = \mathcal{C}_{i-1}$, at which point the algorithm terminates with output $\mathit{explicit}(\mathcal{C}_i)$. $\square$

We now address completeness. The proof of this is somewhat involved and represents the core part of the correctness of the algorithm. In essence there is a tension between completeness and termination: the transformations must be applied sufficiently often that the least model of the constraints eventually becomes explicit, but not so often that they can be applied infinitely often.

Lemma 13 (Completeness) Δ_1 *is complete.*

Proof: Let $\mathcal{C}$ be the result of exhaustively applying Δ_1 to a collection of constraints. This implies that $\delta(\mathcal{C}) \subseteq \mathcal{C}$ for all transformations δ in Δ_1. Adopting the notation of the generic algorithm, let the sequence of constraints obtained by this exhaustive application be $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_i$ where $\mathcal{C}_i = \mathcal{C}$. Let $\mathcal{D}$ denote the subset of constraints in $\mathcal{C}$ of form $\mathcal{X} \supseteq a$ where a is a non-variable atomic set expression. Clearly $\mathcal{D} \subseteq \mathit{explicit}(\mathcal{C}) \subseteq \mathcal{C}$, and so $lm(\mathcal{D}) \subseteq lm(\mathit{explicit}(\mathcal{C})) \subseteq lm(\mathcal{C})$. The remainder of the proof shows that $lm(\mathcal{D}) = lm(\mathcal{C})$, and it is clear that this implies $lm(\mathit{explicit}(\mathcal{C})) = lm(\mathcal{C})$, as required by the definition of completeness.

Since $lm(\mathcal{D}) \subseteq lm(\mathcal{C})$, it only remains to prove that $lm(\mathcal{D}) \supseteq lm(\mathcal{C})$, and this can be established by showing that $lm(\mathcal{D})$ is a model of $\mathcal{C}$. Let $\mathcal{I}_\mathcal{D}$ denote $lm(\mathcal{D})$. Proposition 17 shows that $v \in \mathcal{I}_\mathcal{D}(\mathcal{X})$ iff there exists a constraint $\mathcal{X} \supseteq a$ in $\mathcal{D}$ such that $v \in \mathcal{I}_\mathcal{D}(a)$. Since $\mathcal{D}$ consists of those constraints in $\mathcal{C}$ that have the form $\mathcal{X} \supseteq a$ where a is atomic and non-variable, it follows that

$$v \in \mathcal{I}_\mathcal{D}(\mathcal{X}) \text{ iff } \begin{array}{l} v \in \mathcal{I}_\mathcal{D}(a) \text{ for some constraint } \mathcal{X} \supseteq a \text{ in } \mathcal{C} \\ \text{where } a \text{ is a non-variable atomic set expression} \end{array} \tag{7.17}$$

The remainder of the proof uses this fact to show that $\mathcal{I}_\mathcal{D}$ is a model of $\mathcal{C}$. Consider each possible constraint in $\mathcal{C}$ in turn:

Case (i): Consider a constraint of the form $\mathcal{X} \supseteq a$ where a is an atomic set expression. First suppose that a is not a set variable. This means that

$\mathcal{X} \supseteq a$ appears in $\mathcal{D}$ and so is immediate that $\mathcal{I}_\mathcal{D}$ is a model of such a constraint. On the other hand, suppose that a is a set variable, say $\mathcal{Y}$, and let v be a value such that $v \in \mathcal{I}_\mathcal{D}(\mathcal{Y})$. From (7.17) it follows that there exists a constraint $\mathcal{Y} \supseteq a'$ in $\mathcal{C}$ where a' is a non-variable atomic set expression such that $v \in \mathcal{I}_\mathcal{D}(a')$. So, $\mathcal{C}$ contains $\mathcal{X} \supseteq \mathcal{Y}$ and $\mathcal{Y} \supseteq a'$, and since the application of Transformation 2 to $\mathcal{C}$ does not produce any new constraints, it must be the case that $\mathcal{X} \supseteq a'$ already appears in $\mathcal{C}$. Hence $\mathcal{X} \supseteq a'$ is in $\mathcal{D}$ and so $v \in \mathcal{I}_\mathcal{D}(\mathcal{X})$. This completes the proof that $\mathcal{I}_\mathcal{D}$ is a model of $\mathcal{X} \supseteq a$.

Case (ii): Consider a constraint of the form $\mathcal{X} \supseteq f^{-1}_{(i)}(a)$ where a is an atomic set expression. First suppose that a is not a set variable and let $v \in \mathcal{I}_\mathcal{D}(f^{-1}_{(i)}(a))$. This means that there exists a value $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(a)$ such that v_i is v. Since a is not a set variable or $\top$ (recall that $\top$ is omitted from this section), it must be the case that a is of the form $f(a_1, \ldots, a_n)$ where each a_i is an atomic set expression such that $v_i \in \mathcal{I}_\mathcal{D}(a_i)$. This implies that each a_i is non-empty in the least model of *explicit*$(\mathcal{C})$ and so the preconditions of Transformation 3 are satisfied. Hence the constraint $\mathcal{X} \supseteq a_i$ must already appear in $\mathcal{C}$. By case (i), $\mathcal{I}_\mathcal{D}$ is a model of $\mathcal{X} \supseteq a_i$, and it follows that $v \in \mathcal{I}_\mathcal{D}(\mathcal{X})$, and so $\mathcal{I}_\mathcal{D}$ is a model of $\mathcal{X} \supseteq f^{-1}_{(i)}(a)$.

On the other hand, suppose that a is a set variable, say $\mathcal{Y}$, and let $v \in \mathcal{I}_\mathcal{D}(f^{-1}_{(i)}(\mathcal{Y}))$. This means that there exists a value $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(\mathcal{Y})$ such that v_i is v. By (7.17), there exists a constraint $\mathcal{Y} \supseteq a$ such that a is a non-variable atomic set expression and $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(a)$. Since the application of Transformation 2 to $\mathcal{C}$ does not produce any new constraints, it must be the case that $\mathcal{X} \supseteq f^{-1}_{(i)}(a)$ already appears in $\mathcal{C}$. Clearly $v = v_i \in \mathcal{I}_\mathcal{D}(f^{-1}_{(i)}(a))$. Moreover, we have just argued that $\mathcal{I}_\mathcal{D}$ must satisfy such a constraint. It follows that $v \in \mathcal{I}_\mathcal{D}(\mathcal{X})$, and this completes the proof that $\mathcal{I}_\mathcal{D}$ satisfies $\mathcal{X} \supseteq f^{-1}_{(i)}(a)$.

Case (iii): The final case deals with constraints involving intersection. Such constraints are of the form $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$ where each a_i is an atomic set expression and $m \geq 2$. The proof is by induction on v and the induction hypothesis is: for all values v and for all constraints $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$, $m \geq 2$, appearing in $\mathcal{C}$,

(a) $v \in \mathcal{I}_\mathcal{D}(a_1 \cap \cdots \cap a_m)$ implies $v \in \mathcal{I}_\mathcal{D}(\mathcal{X})$, and

(b) if $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$ is introduced by an application of Transformation 4 then $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$ implies $v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m)$.

Let v be a value such that the induction hypothesis holds for all values with fewer function symbols that v. Before considering (a) and (b), it is convenient to first prove the following statement: if v' has fewer symbols that v and $a_1, \ldots, a_k$ appear in $\mathcal{C}_i$ then

$$v' \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_k) \quad \text{iff} \quad \bigwedge_{a \in N} v' \in \mathcal{I}_{\mathcal{D}}(a) \text{ where } N = \bigcup_{j=1..k} \mathcal{N}(a_j) \tag{7.18}$$

This is proved by a secondary induction on i. Suppose that (7.18) holds for all $i' < i$. Let $a_1, \ldots, a_k$ appear in $\mathcal{C}_i$ and consider the following chain of propositions.

$$\begin{aligned} v' \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m) \quad & \text{iff} \quad \bigwedge_{j=1..m} v' \in \mathcal{I}_{\mathcal{D}}(a_j) \\ & \text{iff} \quad \bigwedge_{j=1..m} \bigwedge_{a \in \mathcal{N}(a_j)} v' \in \mathcal{I}_{\mathcal{D}}(a) \\ & \text{iff} \quad \bigwedge_{a \in N} v' \in \mathcal{I}_{\mathcal{D}}(a) \text{ where } N = \bigcup_{j=1..k} \mathcal{N}(a_j) \end{aligned}$$

The first step is just an expansion of $\cap$. For the second step, take each a_j in turn and consider two cases. If a_j is not an intersection variable then $\mathcal{N}(a_j) = \{a_j\}$ and so the second step is trivial. On the other hand, suppose that a_j *is* an intersection variable, say $\mathcal{V}_{N_j}$. Corresponding to $\mathcal{V}_{N_j}$, there exists a constraint $\mathcal{V}_{N_j} \supseteq a'_1 \cap \cdots a'_l$, $l \geq 2$, that is introduced by Transformation 4. Moreover, this constraint must appear in $\mathcal{C}_{i-1}$ and $N_j = \mathcal{N}(a'_1) \cup \cdots \cup \mathcal{N}(a'_l)$. Now, since v' is smaller than v and $\mathcal{C}_{i-1}$ is constructed before $\mathcal{C}_i$, the main induction hypothesis and the secondary induction hypothesis respectively imply that

$$\begin{aligned} v' \in \mathcal{I}_{\mathcal{D}}(a'_1 \cap \cdots \cap a'_l) \quad & \text{iff} \quad v' \in \mathcal{I}_{\mathcal{D}}(\mathcal{V}_{N_j}), \qquad \text{and} \\ v' \in \mathcal{I}_{\mathcal{D}}(a'_1 \cap \cdots \cap a'_l) \quad & \text{iff} \quad \bigwedge_{a \in N_j} v' \in \mathcal{I}_{\mathcal{D}}(a) \end{aligned}$$

and the second step follows immediately. The final step in the chain follows from the definition of N. This completes the inductive proof of (7.18). The following key property is an immediate corollary of (7.18): if v' has fewer symbols than v, and $V_N, a_1, \ldots, a_k$ appear in $\mathcal{C}_i$ where $N = \mathcal{N}(a_1) \cup \cdots \cup$

$\mathcal{N}(a_k)$, then

$$v' \in \mathcal{I}_{\mathcal{D}}(\mathcal{X}) \text{ iff } v' \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_k) \tag{7.19}$$

Now consider part (a) of the main induction hypothesis. Assume that $v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m)$. It follows that $v \in a_i$, $i = 1..m$. Now, if one of the a_i is a set variable, say $\mathcal{Y}$, then (7.17) implies that there exists a constraint $\mathcal{Y} \supseteq a$ in $\mathcal{C}$ where a is a non-variable atomic set expression such that $v \in \mathcal{I}_{\mathcal{D}}(a)$. This means that the preconditions of Transformation 3 are satisfied, and so the constraint $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_{i-1} \cap a \cap a_{i+1} \cap \cdots \cap a_m$ must appear in $\mathcal{C}$.

This argument may be repeated if necessary, and it follows that $\mathcal{C}$ must contain a constraint of the form $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$ where each a_i is a non-variable atomic set expression such that $v \in a_i$, $i = 1..m$. Let v be $f(v_1, \ldots, v_n)$. Then each a_i must be of the form $f(a_{i,1}, \ldots, a_{i,n})$ such that $v_j \in \mathcal{I}_{\mathcal{D}}(a_{i,j})$, $i = 1..m$, $j = 1..n$. This implies that $v_j \in \mathcal{I}_{\mathcal{D}}(a_{1,j} \cap \cdots \cap a_{m,j})$. Since $\mathcal{C}$ contains all constraints generated by Transformation 4, it follows that $\mathcal{C}$ contains $\mathcal{X} \supseteq f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n})$ such that $N_j = \mathcal{N}(a_{1,j}) \cup \cdots \cup \mathcal{N}(a_{m,j})$. By (7.19), $v_j \in \mathcal{I}_{\mathcal{D}}(\mathcal{V}_{N_j})$, $j = 1..n$, and so $v \in \mathcal{I}_{\mathcal{D}}(f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n}))$. Since $\mathcal{X} \supseteq f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n})$ appears in $\mathcal{C}$, and hence in $\mathcal{D}$, it follows that $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$, and this completes the proof of (a).

To prove (b), suppose that $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$ is introduced by an application of Transformation 4. By inspection of Transformation 4, $\mathcal{X}$ must be an intersection variable. The first part of the proof shall establish

$$\text{if } (\mathcal{X} \supseteq se) \in \mathcal{C} \text{ then } v \in \mathcal{I}_{\mathcal{D}}(se) \text{ implies } v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m) \tag{7.20}$$

Now, any constraint in $\mathcal{C}$ that has the form $\mathcal{X} \supseteq se$ must be the result of applications of transformations to $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$ (since initially this is the only constraint in $\mathcal{C}$ involving $\mathcal{X}$). Moreover, the only transformations that can be involved in this process are Transformations 2 and 4. From inspection of these transformations it must be the case that a constraint of the form $\mathcal{X} \supseteq se$ in $\mathcal{C}$ must be such that se is either:

(α) $a'_1 \cap \cdots \cap a'_m$ such that either a'_i is a_i or a non-variable atomic set expression a'_i such that $a_i \supseteq a'_i$ appears in $\mathcal{C}$, or

(β) $f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n})$ such that there is a constraint $\mathcal{X} \supseteq f(a_{1,1}, \ldots, a_{1,n}) \cap \cdots \cap f(a_{m,1}, \ldots, a_{m,n})$ in $\mathcal{C}$ such that $N_j = \bigcup_{i=1..m} \mathcal{N}(a_{i,j})$, $j = 1..n$.

Now, in case (α), it is easy to see that $\mathcal{I}_{\mathcal{D}}(a_i) \supseteq \mathcal{I}_{\mathcal{D}}(a_i')$ because either a_i is identical to a_i' or else $a_i \supseteq a_i'$ appears in $\mathcal{C}$ and hence in $\mathcal{D}$ (noting that a_i must be a variable and a_i' is a non-variable atomic set expression). Hence it follows that $\mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m) \supseteq \mathcal{I}_{\mathcal{D}}(se)$ and this proves (7.20).

To prove (7.20) in case (β), assume that $v \in \mathcal{I}_{\mathcal{D}}(se)$. This implies that v must be of the form $f(v_1, \ldots, v_n)$ such that $v_j \in \mathcal{I}_{\mathcal{D}}(\mathcal{V}_{N_j})$, $j = 1..n$. By (7.19), $v_j \in \mathcal{I}_{\mathcal{D}}(a_{i,j})$, $i = 1..m$, $j = 1..n$. It follows that $v \in \mathcal{I}_{\mathcal{D}}(f(a_{j,1}, \ldots, a_{j,n}))$, $j = 1..m$. Hence

$$v \in \mathcal{I}_{\mathcal{D}}(f(a_{1,1}, \ldots, a_{1,n}) \cap \cdots \cap f(a_{m,1}, \ldots, a_{m,n})).$$

Now, the expression $f(a_{1,1}, \ldots, a_{1,n}) \cap \cdots \cap f(a_{m,1}, \ldots, a_{m,n})$ must satisfy (α), for which (7.20) has already been proved. Hence $v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m)$ and this proves (7.20) for case (β).

Finally, (b) can be proved as follows. If $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$ then there exists a constraint $\mathcal{X} \supseteq a$ in $\mathcal{D}$ such that $v \in \mathcal{I}_{\mathcal{D}}(a)$. Since $\mathcal{X} \supseteq a$ also appears in $\mathcal{C}$, it follows from (7.20) that $v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m)$. $\square$

Now, combining the above lemmas with Theorem 7 proves that

Theorem 8 (Correctness of projection-intersection Algorithm) *When input with a collection $\mathcal{C}_0$ of standard form constraints, the projection-intersection algorithm terminates and outputs explicit form constraints $\mathcal{C}_{out}$ such that $lm(\mathcal{C}_{out}) =_{var(\mathcal{C}_0)} lm(\mathcal{C}_0)$.*

We note that a large part of the correctness proof for the intersection-projection algorithm involves showing that the algorithm computes exactly the least model of the set constraints. This arises because of the philosophy behind set based approximation: that we should strive for simple *declarative* notions of program approximation, independent algorithmic details. Hence, algorithm "correctness" in this context involves showing that our algorithm is faithful to the definition of set based program approximation, and this is achieved by showing that each step of the algorithm is equivalence preserving. In contrast, if we are only interested in showing that the algorithm is a conservative approximation of the program (and this is case in most other works on program analysis) then only lemmas 12 and 13 are needed.

We also note that the above algorithm is incremental in the sense that the execution of the main loop could be halted at any stage and extra constraints could be added. Moreover, the solving of projections and intersections is closely intertwined. An alternative formulation of the algorithm can be obtained by separating the solving of projection and intersection (such an approach is taken by Heintze and Jaffar in [22]). The advantage of this is that the proofs relating the solving of intersections become simpler. In essence this is because all of the intersections can be identified and performed together and so complex induction hypotheses and intersection variable invariants are not needed. The disadvantage is efficiency. In particular, by integrating the solving of projection and intersection, the simplification of projections becomes significantly cheaper.

7.6 Quantified Set Expressions

This section generalizes the algorithm of the last section to deal with the quantified set expressions that appear in $\mathcal{SC}_P$. Specifically, the set constraints considered are constructed from OP_2, where OP_2 consists of quantified operators and complement constants, as well as the set operators in OP_1. The constraints considered in this section shall be constructed from OP_2 (note that, unlike the previous section, we shall include $\top$). Three main complications arise in extending the algorithm from the previous section to deal with the $\mathcal{SC}_P$. First, reasoning about quantified set expressions involves reasoning about whether a program term is defined under an environment, and this in inherently complex. Second, reasoning about apartness conditions $s \dagger se$ requires the introduction of complement constants. Third, although $lm(\mathcal{SC}_P)$ turns out to be decidable, $lm(\mathcal{C})$ for a collection $\mathcal{C}$ of arbitrary set constraints constructed from OP_2 is not in general decidable. This means that termination of the algorithm requires careful reasoning about the details of the constraints that arise during its execution.

Before presenting the algorithm, we shall first outline why the general problem of solving set constraints constructed from OP_2 is not decidable. In essence, this is because the combination of projection and function symbols on the left hand sides of quantified set expressions allows unbounded notions of dependency to be introduced (see the discussion in Section 5.4, page 123). More concretely, let g, f and c be function symbols of arity 2, 1 and 0 respectively and consider the constraints

$$\begin{array}{lll}\mathcal{X} & \supseteq & g(c,c) \\ \mathcal{X} & \supseteq & \{X : g(f^{-1}_{(1)}(g^{-1}_{(1)}(X)), f^{-1}_{(1)}(g^{-1}_{(2)}(X))) \in \mathcal{X}\}\end{array}$$

In essence the quantified set expression in the second constraint unpackages values in $\mathcal{X}$ (which are all of the form $g(v_1, v_2)$), wraps an "f" around both and then packages them up again. The least model of this constraint maps $\mathcal{X}$ into the set $\{g(f^n(c), f^n(c)) : n \geq 0\}$. This example can be extended to code up undecidable problems such a Post's correspondence problem. For example, let $(\sigma_1, \nu_n), \ldots, (\sigma_n, \nu_n)$ be a collection of pairs of strings. Now, treat the letters that make up these strings as unary function symbols, and write the constraints

$$\begin{array}{lll}\mathcal{X} & \supseteq & g(c,c) \\ \mathcal{X} & \supseteq & \{X : g(\sigma_i^{-1}(g^{-1}_{(1)}(X)), \nu_i^{-1}(g^{-1}_{(2)}(X))) \in \mathcal{X}\}, \quad i = 1..n\end{array}$$

where the notation σ^{-1} denotes the sequence of projection operators such that $\mathcal{I} \models \sigma^{-1}(\sigma(se)) = se$ for all interpretations $\mathcal{I}$ and set expressions se. For example, if σ is the string fgg then $\sigma^{-1}(se)$ denotes $g^{-1}_{(1)}(g^{-1}_{(1)}(f^{-1}_{(1)}(se)))$. It is easy to see that the least model of these constraints maps $\mathcal{X}$ into the set of all values of the form

$$g(\sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_k}(c),\ \nu_{i_1}\nu_{i_2}\cdots\nu_{i_k}(c))$$

where $\{i_i, i_2, \ldots, i_k\} \subseteq \{1, \ldots, n\}$. As a special case, we can also define the set $\{g(\sigma(c), \sigma(c))\ :\ \text{for all strings } \sigma\}$. Combining these two observations with the fact $\mathcal{OP}_2$ includes intersection operators, proves that any Post's correspondence problem can be coded up in constraints $\mathcal{C}$ such that, for some variable $\mathcal{Y} \in var(\mathcal{C})$, $lm(\mathcal{C})(\mathcal{Y}) = \{\}$ iff the correspondence problem has no solution.

The undecidability of the class of set constraints constructed from $\mathcal{OP}_2$ means that the correctness and termination of the algorithm presented in this section depends crucially on the form of the quantified set expressions appearing in $\mathcal{SC}_P$. By inspection, these expressions are of the form $\{X : conj\}$ such that X is a program variable and each quantified condition in $conj$ has one of the following forms:

(I) $s \in se$ where s is constructed from projections and program variables,

(II) $s \dagger se$ where s is constructed from projections and program variables,

(III) $t \in a$ where t is an arbitrary program term and a is a ground atomic set expression, or

(IV) $s \in se$ where s is a constructed from function symbols and program variables,

where, in each case, se is a set expression containing only set variables, projections and function symbols. The first kind of quantified condition arises from program conditions of the form $s = t$. The second arises from program conditions of the form $s \neq t$. The third arises from the construction of *defined*(t) in the constraints corresponding to an assignment statement. Finally, the last kind of quantified condition arises during the translation of environment constraints corresponding to logic programs rules.

To control the form of the quantified set expressions that are constructed during the algorithm, it is convenient to maintain quantified set expressions in an even more restrictive form, which we now describe.

Reduced Form Quantified Set Expressions

A conjunction of quantified conditions *conj* is in *reduced form* if no condition appears twice in *conj* and each quantified condition in *conj* is one of the following forms:

(a) $X \in se$,

(b) $s \in X$ where s consists of program variables and function symbols, or

(c) $s \dagger se$,

where, unless otherwise specified, X is a program variable, se is a set expression and s is a program term. In other words, quantified conditions such as $g_{(2)}^{-1}(X) \in \mathcal{Z}$ and $g(X, Y) \in g(\mathcal{W}, \mathcal{Z})$ are excluded. A collection of constraints $\mathcal{C}$ is in *reduced form* if all conjunctions therein are in reduced form. The algorithm maintains constraints in reduced form via a reduction process that converts constraints into reduced form constraints. In essence, this involves rewriting quantified conditions such as $g_{(2)}^{-1}(X) \in \mathcal{Z}$ into $X \in g(\top, \mathcal{Z})$ and $g(X, Y) \in g(\mathcal{W}, \mathcal{Z})$ into $X \in \mathcal{W} \quad \wedge \quad Y \in \mathcal{Z}$.

(i) Replace $f^{-1}_{(i)}(s) \in se$
by $s \in f(\top, \ldots, \top, se, \top, \ldots, \top)$
(se appears at the i^{th} argument of f).

(ii) Replace $f(s_1, \ldots, s_n) \in f(se_1, \ldots, se_n)$
by $s_1 \in se_1 \wedge \cdots \wedge s_n \in se_n$.

(iii) Replace $f(\cdots) \in \top$ or $f(\cdots) \in \overline{g(\cdots)}$ such that $f \neq g$
by *true*.

(iv) Replace $f(s_1, \ldots, s_n) \in \overline{\{se_1, \ldots, se_m\}}$
by $f(s_1, \ldots, s_n) \in \overline{se_1} \ \wedge \ \cdots \ \wedge \ f(s_1, \ldots, s_n) \in \overline{se_m}$

(v) Delete $\mathcal{X} \supseteq \{X : conj \wedge f(\cdots) \in se\}$ if se is either $\bot$,
$g(\cdots)$ such that $f \neq g$, or $\overline{S}$ such that $f(\top, \ldots, \top) \in S$

(vi) Replace $\mathcal{X} \supseteq \{X : conj \wedge f(s_1, \ldots, s_n) \in \overline{f(se_1, \ldots, se_n)}\}$
by $\mathcal{X} \supseteq \{X : conj \wedge s_1 \in \overline{se_1}\}$, ..., $\mathcal{X} \supseteq \{X : conj \wedge s_n \in \overline{se_n}\}$

Figure 7.5: **The Rewrite Steps of REDUCE()**

For notational convenience we shall treat a conjunctions *conj* of quantified conditions as a set of quantified conditions. That is, if $exp_1, \ldots exp_n$ are quantified conditions, then the conjunction $exp_1 \wedge \cdots \wedge exp_n$ and the set $\{exp_1, \ldots, exp_n\}$ shall be used interchangeably. Now, consider the rewrite steps in Figure 7.5 for simplify the quantified conditions in a collection $\mathcal{C}$ of set constraints. In these steps, $se, se_1, \ldots, se_n$ are set expressions, and $s, s_1, \ldots, s_n$ are program terms, and $|S|$ denotes the cardinality of the set S. Steps (i–iv) work at the level of quantified conditions, and replace a quantified condition by a (possibly empty) conjunction of quantified conditions. The remaining steps work at the level of constraints – steps (v) just deletes a constraint and step (vi) replaces a constraint by $n \geq 0$ constraints. Note that in (i), the i^{th} argument of $f(se_1, \ldots, se_n)$ is the set expression se that appears in $f_{(i)}^{-1}(s) \in se$, and the remaining arguments are all $\top$. For example, if h has arity 3, then $h_{(2)}^{-1}(s_1) \in f(c)$ is replaced by $s_1 \in h(\top, f(c), \top)$. It is assumed that if a quantified condition appears more than once in a conjunction constructed by these steps, then copies of the condition are deleted until only one remains.

Where $\mathcal{C}$ is a collection of constraints, define that REDUCE($\mathcal{C}$) is the result of exhaustively applying the steps in Figure 7.5 to $\mathcal{C}$. We now prove the correctness of REDUCE. We begin with some observations about the complement constants. As mentioned earlier, all complement constants used in our constraints shall be of the form

$$\begin{array}{l}\overline{S} \text{ where each } se \in S \text{ either has the form } f(\top, \ldots, \top) \\ \text{or consists only of function symbols.}\end{array} \tag{7.21}$$

In what follows, we shall implicitly assume that this property holds, and only make reference to it when new complement constants are generated. To see that REDUCE preserves property (7.21), note that the only steps of REDUCE that may generate new complement constants are (iv) and (vi). Now, any new complement constants introduced by step (iv) are of the form $\overline{S}$ such that $S \subseteq S'$ for some complement constant $\overline{S'}$ that already appears in the constraints. Similarly, any new complement constants introduced by step (vi) are of the form $\overline{se}$ such that se is different from $\top$ and such that a complement symbol of the form $\overline{f(\ldots, se, \ldots)}$ is already present. We now prove that REDUCE always terminates.

Proposition 26 REDUCE *terminates on any collection $\mathcal{C}$ of set constraints (constructed from $\mathcal{OP}_2$).*

Proof: Consider step (iv). This step serves to replace an occurrence of a constant $\overline{\{se_1, \ldots, se_m\}}$, where by m constants $\overline{se_1}, \ldots, \overline{se_m}$. Moreover, no transformation introduces occurrences of constants $\overline{S}$ such that $|S| \geq 2$. It is clear that there can only be a finite number of applications of step (iv).

Since step (iv) cannot be applied indefinitely, it follows that in an exhaustive application of steps (i–vi) to $\mathcal{C}$, a point must eventually be reached such that there are no further applications of step (ii). Termination can then be proved by observing that steps (i), (ii), (iii), (v) and (vi) all reduce the total number of symbols in the program terms appearing in the quantified conditions in $\mathcal{C}$. To see this, note that in cases (ii) and (vi) a program term $f(s_1, \ldots, s_n)$ is replace by s_i, $i = 1..n$, in cases (iii), and (v) a program term is removed, and in case (i) a program term $f^{-1}_{(i)}(s)$ is replace by s. $\square$

We next show that REDUCE preserves standard form and also preserves the form described by (I–IV). This involves a straightforward verification for each step of REDUCE.

Proposition 27 *If $\mathcal{C}$ is in standard form then so is* REDUCE($\mathcal{C}$).

Proof: Let $\mathcal{C}$ be a collection of standard form constraints. From the definition of standard form, any quantified conditions appearing in $\mathcal{C}$ must be of the form $s \in a$ or $s \dagger a$ such that a is atomic. Now, suppose that one of steps (i–vi) of REDUCE is applied to $\mathcal{C}$. Then, the only new constraints introduced by this step must be of the form $\mathcal{X} \supseteq \{X : conj\}$ such that each quantified condition in *conj* either

1. appears somewhere in $\mathcal{C}$;
2. has the form $s \in f(se_1, \ldots, se_n)$ such that $\mathcal{C}$ contains a quantified condition of the form $t \in se$ and each se_i is either se or $\top$ (see step (i));
3. has the form $s \in se_i$ such that $\mathcal{C}$ contains a quantified condition of the form $t \in f(se_1, \ldots, se_n)$ (see step (ii)), or
4. has the form $s \in \overline{S}$ (see steps (iv) and (vi))

In each case it is clear that the quantified condition must have the form $s \in a$ or $s \dagger a$ where a is atomic. It follows that any new constraints generated by the step must be in standard form. Hence each step of REDUCE preserves standard form, and the proposition follows. ∎

Proposition 28 *Let $\mathcal{C}$ be a collection of constraints such that each quantified condition appearing in $\mathcal{C}$ is of one of the forms (I–IV). Then each quantified condition in* REDUCE($\mathcal{C}$) *is of one of the forms (I–IV).*

Proof: It suffices to show that the steps (i–vi) preserve the forms (I–IV). Now, only steps (i), (ii), (iv) and (vi) can introduce new quantified conditions. Consider each of these steps in turn. In step (i), the new quantified conditions is of the form $s \in f(\top, \ldots, se, \ldots, \top)$ such that $f_{(i)}^{-1}(s) \in se$ is either of form (I) or (III). If $f_{(i)}^{-1}(s) \in se$ is of form (I), then s contains only projections and program variables, and so $s \in f(\top, \ldots, se, \ldots, \top)$ is of form (I). If $f_{(i)}^{-1}(s) \in se$ is of form (III) then $f(\top, \ldots, se, \ldots, \top)$ is ground, and so $s \in f(\top, \ldots, se, \ldots, \top)$ is of form (III). In steps (ii), the new quantified conditions are of the form $s_i \in se_i$ such that $f(s_1, \ldots, s_n) \in f(se_1, \ldots, se_n)$ is either of form (III) or (IV), since forms (I) and (II) are not applicable. Hence either each s_i is a term constructed from function symbols and variables, or else each se_i is ground and atomic. This means that each $s_i \in se_i$ is either of form (III) or (IV). In steps (iv) and (vi), the new quantified conditions are of the form $s \in \overline{S}$, and it is immediate that such a quantified conditions are of form (III). ∎

Combining the previous two propositions with some simple observations about the rewrite steps that make up REDUCE proves that REDUCE achieves the desired rewriting of constraints into reduced form.

Proposition 29 *Let $\mathcal{C}$ be a collection of standard form constraints such that each quantified condition appearing in $\mathcal{C}$ is of one of the forms (I–IV). Then* REDUCE($\mathcal{C}$) *is in reduced form.*

Proof: By definition, none of the steps of REDUCE can be applied to REDUCE($\mathcal{C}$). Hence, if $s \in se$ appears in REDUCE($\mathcal{C}$) then s cannot have the form $f_{(i)}^{-1}(s')$. Furthermore, if s has the form $f(s_1, \ldots, s_n)$ then se cannot be of the form $f(\cdots)$, $\top$, $\overline{S}$, $\perp$ or $g(\cdots)$, $g \neq f$. Hence, each quantified condition in REDUCE($\mathcal{C}$) must be of one of the following forms:

(a) $X \in se$.

(b) $f(\cdots) \in se$ where se is either a set variable, or of the form $se_1 \cup se_2$ or $op(se_1, \ldots, se_n)$.

(c) $s \dagger se$.

Now, by Proposition 27, REDUCE($\mathcal{C}$) is in standard form and so all constraints in REDUCE($\mathcal{C}$) are of the form $\mathcal{X} \supseteq a$ or $\mathcal{X} \supseteq op(a_1, \ldots, a_n)$ where $a, a_1, \ldots, a_n$ are atomic. Hence the quantified conditions in REDUCE($\mathcal{C}$) must be of the form $s \in a$ or $s \dagger a$ where s is a program term and a is an atomic set expression. Also, by Proposition 28, each quantified condition in REDUCE($\mathcal{C}$) must satisfy one of the forms (I–IV). Hence, in case (a) above, se must be atomic. Now consider case (b). Since se is atomic it must be the case that se is a set variable, say $\mathcal{X}$. Hence se cannot be ground and so $f(\cdots) \in \mathcal{X}$ must satisfy either (I) or (IV), and it follows that $f(\cdots)$ must consist of function symbols. In case (c), no further conditions can be established. In summary, each quantified condition in REDUCE($\mathcal{C}$) is either of the form (a) $X \in se$, (b) $s \in \mathcal{X}$ where s consists of program variables and function symbols, or (c) $s \dagger se$, and so each conjunction of quantified conditions in REDUCE($\mathcal{C}$) is in reduced form. $\square$

It remains to prove that REDUCE is correct. That is, we seek to show that $lm(\mathcal{C}) = lm(\text{REDUCE}(\mathcal{C}))$. Unfortunately this is not always the case. For example, consider the constraint

$$\mathcal{X} \supseteq \{X : g(g^{-1}_{(1)}(X), X) \in \overline{g(c,b)}\} \tag{7.22}$$

where b and c are constants and g is a binary function symbol. Let exp denote $g(g^{-1}_{(1)}(X), X) \in \overline{g(c,b)}$ and let $\mathcal{I}$ be an interpretation. By definition, $\rho \in \mathcal{I}(exp)$ if $\rho \rhd g(g^{-1}_{(1)}(X), X)$ and $\rho(g(g^{-1}_{(1)}(X), X)) \notin \{g(c,b)\}$. Now, the first condition implies that $\rho(X)$ must be of the form $g(\cdots)$ and this subsumes the second condition. Hence, $\rho \in \mathcal{I}(exp)$ iff $\rho(X)$ has the form $g(v_1, v_2)$ for some values v_1 and v_2. Thus, the least model of the constraint (7.22) maps $\mathcal{X}$ into the set $\{g(v_1, v_2) :\ v_1 \text{ and } v_2 \text{ are values}\}$.

Now, consider the application of REDUCE to the constraint (7.22). This involves replacing the constraint by $\mathcal{X} \supseteq \{X : g^{-1}_{(1)}(X) \in \overline{c}\}$ and $\mathcal{X} \supseteq \{X : X \in \overline{b}\}$ (using step (vi)), and then replacing $g^{-1}_{(1)}(X) \in \overline{c}$ by $X \in g(\overline{c}, \top)$ (using step (i)). Hence, the final reduced form constraints are:

$$\begin{array}{rcl}\mathcal{X} & \supseteq & \{X : X \in g(\overline{c}, \top)\} \\ \mathcal{X} & \supseteq & \{X : X \in \overline{b}\}\end{array}$$

The least model of these constraints maps $\mathcal{X}$ into the set of all values different from b. Clearly the least model of (7.22) has not been preserved.

The problem occurs during the application of step (vi), when (7.22) is replaced by $\mathcal{X} \supseteq \{X : g^{-1}_{(1)}(X) \in \overline{c}\}$ and $\mathcal{X} \supseteq \{X : X \in \overline{b}\}$. Before the step, (7.22) is equivalent to

$$\mathcal{X} \supseteq \left\{\rho(X) : \left(\rho\left(g^{-1}_{(1)}(X)\right) \notin \{c\} \text{ or } \rho(X) \notin \{b\}\right) \text{ and } \rho \rhd g^{-1}_{(1)}(X)\right\}.$$

However, after the step, the resulting constraints are equivalent to

$$\begin{array}{rl}\mathcal{X} \supseteq & \left\{\rho(X) : \rho\left(g^{-1}_{(1)}(X)\right) \notin \{c\} \text{ and } \rho \rhd g^{-1}_{(1)}(X)\right\} \\ & \bigcup \ \{\rho(X) : \rho(X) \notin \{b\}\}.\end{array}$$

Hence, the least model is not preserved by REDUCE because the condition $\rho \rhd g^{-1}_{(1)}(X)$ is not present in the quantified set expression $\{X : X \in \overline{b}\}$. In general, there are two steps of REDUCE that are potentially incorrect in this sense, namely steps (iii) and (vi). The problem in both cases is due to the requirement that environments be defined on each program term appearing in a quantified condition.

The proof of correctness proceeds by showing that whenever one of steps (iii) or (vi) is applied to a quantified condition $s \in se$ in a conjunction $conj$, the requirement that ρ is defined on s is in fact redundant because this requirement essentially appears elsewhere in $conj$. For example, consider the set constraint

$$\mathcal{X} \supseteq \{X : X \in g(\top, \ldots, \top) \wedge g(g^{-1}_{(1)}(X), X) \in \overline{g(c, b)}\}.$$

An application of step (vi) to this constraint yields the two constraints

$$\begin{array}{rcl}\mathcal{X} & \supseteq & \{X : X \in g(\top, \ldots, \top) \wedge X \in g(\overline{c}, \top)\} \\ \mathcal{X} & \supseteq & \{X : X \in g(\top, \ldots, \top) \wedge X \in \overline{b}\}\end{array}$$

This step is correct because the implicit condition that X must have the form $g(\cdots)$, which is dropped during this step, is in fact redundant because it appears elsewhere in the conjunction.

More generally, recall that $\rho \in \mathcal{I}(conj)$ if

$$\begin{array}{lllll} & \rho \rhd s & \wedge & \rho(s) \in \mathcal{I}(se) & \text{for all } s \in se \text{ in } conj \\ \text{and} & \rho \rhd s & \wedge & \exists v\,(v \neq \rho(s) \;\wedge\; v \in \mathcal{I}(se)) & \text{for all } s \dagger se \text{ in } conj \end{array}$$

Now, consider an alternative definition of $\rho \in \mathcal{I}(conj)$, which omits quantified conditions that are not defined:

$$\begin{array}{lll} & \rho(s) \in \mathcal{I}(se) & \text{for all } s \in se \text{ in } conj \text{ such that } \rho \rhd s \\ \text{and} & \exists v\,(v \neq \rho(s) \wedge v \in \mathcal{I}(se)) & \text{for all } s \dagger se \text{ in } conj \text{ such that } \rho \rhd s \end{array}$$

Under certain circumstances, these two definitions coincide, and this turns out to be the key property for correctness. To formalize this, first extend the $\rhd$ notation. Define that $\rho \rhd exp$ holds if exp is $s \in se$ or $s \dagger se$ and $\rho \rhd s$. In other words, $\rho \rhd exp$ denotes the condition that the program term appearing in exp is defined. Now, define that a quantified set expression is *safe* with respect to an interpretation as follows.

Definition 18 *A quantified set expression $\{X : conj\}$ is <u>safe with respect to</u> an interpretation $\mathcal{I}$ if, for each $\rho \notin \mathcal{I}(conj)$, conj contains a quantified condition exp_ρ such that $\rho \rhd exp_\rho$ and $\rho \notin \mathcal{I}(exp_\rho)$. A collection of set constraints is <u>safe with respect to</u> $\mathcal{I}$ if all of the quantified set expressions it contains are safe with respect to $\mathcal{I}$.*

In other words $\{X : conj\}$ is safe with respect to $\mathcal{I}$ if $\rho \notin \mathcal{I}(conj)$ implies that there is some quantified condition that is defined under ρ but not satisfied by ρ, and this means that *conj* contains no implicit information in the definedness of program terms. Hence, to determine the relation $\rho \in \mathcal{I}(conj)$, all undefined quantified conditions can be safely ignored.

We now proceed with the proof of correctness for REDUCE. In essence we shall prove that each step of REDUCE is correct in the sense that if it replaces a set expression se_{old} by se_{new} and se_{old} is safe with respect to an interpretation $\mathcal{I}$ then $\mathcal{I}(se_{new}) = \mathcal{I}(se_{old})$. In order to show that an entire application of REDUCE is correct, we shall also need to argue that each step preserves safeness. We begin by proving a somewhat abstract property about preservation of safety that will be used repeatedly in the propositions that follow. In essence, the property considers the replacement of a quantified set expression of the form $\{X : conj \wedge exp\}$ with $\{X : conj \wedge exp_1 \wedge \cdots \wedge exp_n\}$.

Proposition 30 *Let conj be a conjunctions of quantified conditions and let $exp, exp_1, \ldots, exp_n$ be individual quantified conditions. If $\mathcal{I}$ is an interpre-*

tation such that:

(a) $(\forall \rho \rhd exp)\Big(\rho \in \mathcal{I}(exp_1 \wedge \cdots \wedge exp_n)$ *implies* $\rho \in \mathcal{I}(exp)\Big)$,

(b) $\rho \rhd exp$ *implies* $\bigwedge_{i=1..n} \rho \rhd exp_i$, *and*

(c) $\{X : conj \wedge exp\}$ *is safe with respect to* $\mathcal{I}$,

then $\{X : conj \wedge exp_1 \wedge \cdots \wedge exp_n\}$ *is safe with respect to* $\mathcal{I}$.

Proof: To show that $\{X : conj \wedge exp_1 \wedge \cdots \wedge exp_n\}$ is safe with respect to $\mathcal{I}$, suppose that $\rho \notin \mathcal{I}(conj \wedge exp_1 \wedge \cdots \wedge exp_n)$ and we need to show that there exists a quantified condition exp_ρ in $conj \wedge exp_1 \wedge \cdots \wedge exp_n$ such that $\rho \rhd exp_\rho$ and $\rho \notin \mathcal{I}(exp_\rho)$. Now, either (1) $\rho \notin \mathcal{I}(conj)$ or (2) $\rho \notin \mathcal{I}(exp_1 \wedge \cdots \wedge exp_n)$.

In case (1), $\rho \notin \mathcal{I}(conj \wedge exp)$, and so by assumption (c), there must exist exp_ρ in $conj \wedge exp$ such that $\rho \rhd exp_\rho$ and $\rho \notin \mathcal{I}(exp_\rho)$. Now, consider two subcases. Either exp_ρ appears in $conj$, in which case the proof is complete, or else exp_ρ is exp. In this latter case, $\rho \rhd exp$ and $\rho \notin \mathcal{I}(exp)$, and combining this with (a) proves that $\rho \notin \mathcal{I}(exp_1 \wedge \cdots \wedge exp_n)$. It follows that, for some i, $\rho \notin \mathcal{I}(exp_i)$. Moreover, $\rho \rhd exp$ and it follows from (b) that $\rho \rhd exp_i$. Hence exp_i is the required quantified condition.

In case (2), $\rho \notin \mathcal{I}(exp_1 \wedge \cdots \wedge exp_n)$, and so there must exist some i such that $\rho \notin \mathcal{I}(exp_i)$. Again consider two subcases. If $\rho \rhd exp_i$ then the proof is complete. Otherwise, if it is not the case that $\rho \rhd exp_i$, then (b) implies that $\rho \rhd \mathcal{I}(exp)$ does not hold, and this in turn implies that $\rho \notin \mathcal{I}(conj \wedge exp)$. Since $\{X : conj \wedge exp\}$ is assumed to be safe with respect to $\mathcal{I}$ and $\rho \notin \mathcal{I}(conj \wedge exp)$, there is some exp_ρ in $conj \wedge exp$ such that $\rho \rhd exp_\rho$ and $\rho \notin \mathcal{I}(exp_\rho)$. Now exp_ρ cannot be exp because of the assumption that $\rho \rhd \mathcal{I}(exp)$ does not hold. Hence exp_ρ must appear in $conj$ and hence in $conj \wedge exp_1 \wedge \cdots \wedge exp_n$. $\square$

The following two propositions form the core part of the proof of the correctness of REDUCE. In essence, they show that each step is correct and preserves safeness.

Proposition 31 *If one of steps (i–iv) of* REDUCE *is applied to a collection of constraints then the effect of the step is to replace a constraint* $\mathcal{X} \supseteq \{X : conj_{old}\}$ *by* $\mathcal{X} \supseteq \{X : conj_{new}\}$ *such that, for any interpretation* $\mathcal{I}$, *(a)* $\mathcal{I}(\{X : conj_{old}\}) \subseteq \mathcal{I}(\{X : conj_{new}\})$*, and*

(b) if $\{X : conj_{old}\}$ *is safe with respect to* $\mathcal{I}$ *then*

(b1) $\mathcal{I}(\{X : conj_{old}\}) = \mathcal{I}(\{X : conj_{new}\})$*, and*

(b2) $\{X : conj_{new}\}$ *is safe with respect to* $\mathcal{I}$*.*

Proof: If step (i) is used, then $conj_{old}$ is $conj \wedge f_{(i)}^{-1}(s) \in se$, and $conj_{new}$ is $conj \wedge s \in f(\top, \ldots, \top, se, \top, \ldots, \top)$ where se appears at the i^{th} argument of f. Now let $\mathcal{I}$ be an arbitrary interpretation and consider the following equivalences:

$$
\begin{array}{ll}
 & \rho \in \mathcal{I}(f_{(i)}^{-1}(s) \in se) \\
\text{iff} & \rho \rhd f_{(i)}^{-1}(s) \quad \wedge \quad \rho(f_{(i)}^{-1}(s)) \in \mathcal{I}(se) \\
\text{iff} & \rho \rhd s \quad \wedge \quad \rho(s) \text{ is } f(v_1, \ldots, v_n) \quad \wedge \quad v_i \in \mathcal{I}(se) \\
\text{iff} & \rho \rhd s \quad \wedge \quad \rho(s) \text{ is } f(v_1, \ldots, v_n) \quad \wedge \quad v_i \in \mathcal{I}(se) \quad \wedge \quad \bigwedge_{j \neq i} v_j \in \mathcal{I}(\top) \\
\text{iff} & \rho \rhd s \quad \wedge \quad \rho(s) \in \mathcal{I}(f(se_1, \ldots, se_n)) \\
\text{iff} & \rho \in \mathcal{I}(s \in f(se_1, \ldots, se_n))
\end{array}
$$

These equivalences imply that, for all interpretations $\mathcal{I}$, $\rho \in \mathcal{I}(conj_{old})$ iff $\rho \in \mathcal{I}(conj_{new})$. Hence $\mathcal{I}(\{X : conj_{old}\}) = \mathcal{I}(\{X : conj_{new}\})$, and this proves (a) and (b1). It remains to show (b2). Suppose that $\{X : conj_{old}\}$ is safe with respect to $\mathcal{I}$. Then clearly the following properties hold:

- $\mathcal{I}(f_{(i)}^{-1}(s) \in se) = \mathcal{I}(s \in f(se_1, \ldots, se_n))$
- $\rho \rhd f_{(i)}^{-1}(s)$ implies $\rho \rhd s$,
- $\{X : conj \wedge f_{(i)}^{-1}(s) \in se\}$ is safe with respect to $\mathcal{I}$

and hence the preconditions of Proposition 30 are satisfied. It follows that $\{X : conj_{new}\}$ is safe with respect to $\mathcal{I}$.

If step (ii) is used, then $conj_{old}$ is $conj \wedge f(s_1, \ldots, s_n) \in f(se_1, \ldots, se_n)$ and $conj_{new}$ is $conj \wedge s_1 \in se_1 \wedge \cdots \wedge s_n \in se_n$. Now let $\mathcal{I}$ be an arbitrary interpretation and consider the following equivalences:

$$
\begin{array}{ll}
 & \rho \in \mathcal{I}(f(s_1, \ldots, s_n) \in f(se_1, \ldots, se_n)) \\
\text{iff} & \rho \rhd f(s_1, \ldots, s_n) \quad \wedge \quad \rho(f(s_1, \ldots, s_n)) \in \mathcal{I}(f(se_1, \ldots, se_n)) \\
\text{iff} & \bigwedge_{i=1..n} \rho \rhd s_i \ \wedge \ \bigwedge_{i=1..n} \rho(s_i) \in \mathcal{I}(se_i) \\
\text{iff} & \bigwedge_{i=1..n} (\rho \rhd s_i \ \wedge \ \rho(s_i) \in \mathcal{I}(se_i)) \\
\text{iff} & \bigwedge_{i=1..n} \rho \in \mathcal{I}(s_i \in se_i) \\
\text{iff} & \rho \in \mathcal{I}\,(s_1 \in se_1 \wedge \cdots \wedge s_n \in se_n)
\end{array}
$$

These equivalences imply that, for all interpretations $\mathcal{I}$, $\rho \in \mathcal{I}(conj_{old})$ iff $\rho \in \mathcal{I}(conj_{new})$. Hence $\mathcal{I}(\{X : conj_{old}\}) = \mathcal{I}(\{X : conj_{new}\})$, and this proves (a) and (b1).

To prove (b2), assume that $\{X : conj_{old}\}$ is safe with respect to $\mathcal{I}$. Clearly the following properties hold:

- $\mathcal{I}(f(s_1, \ldots, s_n) \in f(se_1, \ldots, se_n)) = \mathcal{I}\,(s_1 \in se_1 \wedge \cdots \wedge s_n \in se_n)$,
- $\rho \rhd f(s_1, \ldots, s_n)$ implies $\rho \rhd s_1 \wedge \cdots \wedge \rho \rhd s_n$
- $\{X : conj_{old}\}$ is safe with respect to $\mathcal{I}$

and so Proposition 30 proves that $\{X : conj_{new}\}$ is safe with respect to $\mathcal{I}$.

If step (iii) is used, then $conj_{old}$ is $conj \wedge s_1 \in se_1$ where s_1 is of the form $f(\cdots)$ and se_1 is either $\top$ or of the form $\overline{g(\cdots)}$ such that $f \neq g$, and $conj_{new}$ is $conj$. Clearly

$$\rho \in \mathcal{I}(conj \wedge s_1 \in se_1) \quad \text{implies} \quad \rho \in \mathcal{I}(conj)$$

and it follows that $\mathcal{I}(\{X : conj_{old}\}) \subseteq \mathcal{I}(\{X : conj\})$. This proves (a). Now consider (b1) and suppose that $\{X : conj_{old}\}$ is safe with respect to $\mathcal{I}$, and consider the following property:

$$\rho \in \mathcal{I}(conj) \quad \text{implies} \quad \rho \rhd s_1 \tag{7.23}$$

To prove (7.23), suppose that $\rho \rhd s_1$ does not hold. This implies that $\rho \notin \mathcal{I}(conj_{old})$. Since se_{old} is safe with respect to $\mathcal{I}$, there exists a quantified condition exp_ρ in $conj_{old}$ such that $\rho \rhd exp_\rho$ and $\rho \notin \mathcal{I}(exp_\rho)$. Since exp_ρ

cannot be $s_1 \in se_1$, it must be the case that exp_ρ appears in $conj$. Hence $\rho \notin \mathcal{I}(conj)$. This completes the proof of (7.23).

Now, consider the condition $s_1 \in se_1$. If $\rho \rhd s_1$, then it is clear that $\rho(s_1) \in \mathcal{I}(se_1)$ because if se_1 is either $\top$ or $\overline{g(\cdots)}$ then $\mathcal{I}(se_1)$ contains all values of the form $f(\cdots)$. Hence $\rho \in \mathcal{I}(s_1 \in se_1)$ iff $\rho \rhd s_1$. Combining this with (7.23) proves that

$$\rho \in \mathcal{I}(conj) \quad \text{implies} \quad \rho \in \mathcal{I}(conj \wedge s_1 \in se_1)$$

and so $\mathcal{I}(\{X : conj_{new}\}) \subseteq \mathcal{I}(\{X : conj_{old}\})$. Combining this with (a) proves (b1).

To prove (b2), assume that $\{X : conj_{old}\}$ is safe with respect to $\mathcal{I}$. Now, we have already proved that $\rho \in \mathcal{I}(s_1 \in se_1)$ iff $\rho \rhd s_1$, and so if $\rho \rhd \mathrm{s}_1$ then $\rho \in \mathcal{I}(s_1 \in se_1)$ implies $\rho \in \mathcal{I}(true)$. Hence the three preconditions of Proposition 30 hold (note that the second is vacuous since $n = 0$) and it follows that $\{X : conj_{new}\}$ is safe with respect to $\mathcal{I}$.

If step (iv) is used, then $conj_{old}$ is $conj \wedge f(s_1, \ldots, s_n) \in \overline{\{se_1, \ldots, se_m\}}$ and $conj_{new}$ is $conj \wedge f(s_1, \ldots, s_n) \in \overline{se_1} \wedge \cdots \wedge f(s_1, \ldots, s_n) \in \overline{se_m}$. It is easy to verify that if $\rho \rhd f(s_1, \ldots, s_n)$ then:

$$\begin{array}{l} \rho(f(s_1, \ldots, s_n)) \in \mathcal{I}(\overline{\{se_1, \ldots, se_m\}}) \quad \text{iff} \\ \rho(f(s_1, \ldots, s_n)) \in \mathcal{I}(\overline{se_1}) \wedge \cdots \wedge \rho(f(s_1, \ldots, s_n)) \in \mathcal{I}(\overline{se_m}) \end{array} \tag{7.24}$$

and this implies that $\mathcal{I}(\{X : conj_{old}\}) = \mathcal{I}(\{X : conj_{new}\})$, for all interpretations $\mathcal{I}$, and hence proves (a) and (b1). To prove (b2), assume that $\{X : conj_{old}\}$ is safe with respect to $\mathcal{I}$. Now, it follows from (7.24) that

$$\begin{array}{l} \mathcal{I}\left(f(s_1, \ldots, s_n) \in \overline{\{se_1, \ldots, se_m\}}\right) = \\ \mathcal{I}\left(f(s_1, \ldots, s_n) \in \overline{se_1} \wedge \cdots \wedge f(s_1, \ldots, s_n) \in \overline{se_m}\right). \end{array}$$

Also, it is clear that

$$\begin{array}{l} \rho \rhd \left(f(s_1, \ldots, s_n) \in \overline{\{se_1, \ldots, se_m\}}\right) \text{ iff} \\ \rho \rhd \left(f(s_1, \ldots, s_n) \in \overline{se_1} \wedge f(s_1, \ldots, s_n) \in \overline{se_m}\right). \end{array}$$

This establishes the preconditions of Proposition 30. Hence $\{X : conj_{new}\}$ is safe with respect to $\mathcal{I}$. $\square$

Proposition 32 *If one of steps (v) or (vi) of* REDUCE *is applied to a collection of constraints then the effect of the step is to replace a constraint* $\mathcal{X} \supseteq \{X : conj_{old}\}$ *by* $n \geq 0$ *constraints* $\mathcal{X} \supseteq \{X : conj_1\}$, *…*, $\mathcal{X} \supseteq \{X : conj_n\}$ *such that, for any interpretation* $\mathcal{I}$,

(a) $\mathcal{I}(\{X : conj_{old}) \subseteq \mathcal{I}(\{X : conj_1\} \cup \cdots \cup \{X : conj_n\})$, *and*

(b) if $\{X : conj\}$ *is safe with respect to* $\mathcal{I}$ *then*

(b1) $\mathcal{I}(\{X : conj_{old}\}) = \mathcal{I}(\{X : conj_1\} \cup \cdots \cup \{X : conj_n\})$, *and*

(b2) each $\{X : conj_i\}$ *is safe with respect to* $\mathcal{I}$.

Proof: If step (v) is used, then the effect of the step is to delete $\mathcal{X} \supseteq \{X : conj_{old}\}$ (i.e. $n = 0$). The proofs for (a) and (b1) follow from the fact that there do not exist environments ρ and interpretations $\mathcal{I}$ such that $\rho \rhd f(\cdots)$ and $\rho(f(\cdots)) \in \mathcal{I}(g(\cdots))$. Hence $\mathcal{I}(\{X : conj_{old}\})$ is the empty set, for all $\mathcal{I}$. Moreover, condition (b2) is vacuous.

If step (vi) is used, then $conj_{old}$ is $f(s_1, \ldots, s_n) \in \overline{f(se_1, \ldots, se_n)}$ and each $conj_i$ is $conj \wedge s_i \in \overline{se_i}$. The proof essentially follows from the following chain of equivalences in which ρ is an environment such that $\rho \rhd f(s_1, \ldots, s_n)$:

$$\begin{aligned}
& \rho \in \mathcal{I}\left(f(s_1, \ldots, s_n) \in \overline{f(se_1, \ldots, se_n)}\right) \\
\text{iff} \quad & \rho\left(f(s_1, \ldots, s_n)\right) \in \mathcal{I}\left(\overline{f(se_1, \ldots, se_n)}\right) \\
\text{iff} \quad & \rho\left(f(s_1, \ldots, s_n)\right) \notin \mathcal{I}\left(f(se_1, \ldots, se_n)\right) \\
\text{iff} \quad & \exists i \left(\rho(s_i) \notin \mathcal{I}(se_i)\right) \\
\text{iff} \quad & \exists i \left(\rho(s_i) \in \mathcal{I}(\overline{se_i})\right) \\
\text{iff} \quad & \exists i \ \rho \in \mathcal{I}\left(s_i \in \overline{se_i}\right).
\end{aligned} \tag{7.25}$$

To prove (a), suppose that $v \in \mathcal{I}(\{X : conj_{old}\})$. This implies that for some ρ, $\rho(X) = v$ and $\rho \in \mathcal{I}(conj_{old})$. Hence $\rho \in \mathcal{I}(conj)$, $\rho \rhd f(s_1, \ldots, s_n)$ and $\rho \in \mathcal{I}\left(f(s_1, \ldots, s_n) \in \overline{f(se_1, \ldots, se_n)}\right)$. The chain of equivalences (7.25) proves that there is an i such that $\rho \in \mathcal{I}\left(s_i \in \overline{se_i}\right)$. Also, $\rho \rhd f(s_1, \ldots, s_n)$ implies that $\rho \rhd s_i$. Hence $\rho \in \mathcal{I}(conj \wedge s_i \in \overline{se_i})$. This means that, for some i, $v \in \mathcal{I}(\{X : conj_i\})$. Hence $v \in \mathcal{I}(\{X : conj_1\} \cup \cdots \cup \{X : conj_n\})$, and this proves (a).

Consider (b1), and assume that $\{X : conj_{old}\}$ is safe with respect to $\mathcal{I}$. It is first necessary to show that

$$\rho \in \mathcal{I}(conj) \text{ implies } \rho \rhd s_i \tag{7.26}$$

where i ranges from 1 to n. To prove (7.26), fix i and suppose that $\rho \rhd s_i$ does not hold. This means that ρ is not defined on $f(s_1, \ldots, s_n)$, and so $\rho \notin \mathcal{I}(conj_{old})$. Since $conj_{old}$ is safe with respect to $\mathcal{I}$, there exists a quantified condition exp_ρ in $conj_{old}$ such that $\rho \rhd exp_\rho$ and $\rho \notin \mathcal{I}(exp_\rho)$. Since exp_ρ cannot be $f(s_1, \ldots, s_n) \in \overline{f(se_1, \ldots, se_n)}$, it must be the case that exp_ρ appears in $conj$. Hence $\rho \notin \mathcal{I}(conj)$. This completes the proof of (7.26).

To complete the proof of (b1), suppose that $v \in \mathcal{I}(\{X : conj_i\})$, for some i, $i = 1..n$. This implies that there is an environment ρ such that $\rho \in \mathcal{I}(conj)$ and $\rho \in \mathcal{I}(s_i \in \overline{se_i})$. From the implication (7.26), it follows that $\rho \rhd f(s_1, \ldots, s_n)$. Combining this with $\rho \in \mathcal{I}(s_i \in \overline{se_i})$ and the chain of equivalences (7.25) proves that $\rho \in \mathcal{I}\left(f(s_1, \ldots, s_n) \in \overline{f(se_1, \ldots, se_n)}\right)$. Hence $\rho \in \mathcal{I}(conj_{old})$ and so $v \in \mathcal{I}(\{X : conj_{old}\})$.

Finally, consider (b2) and assume that $\{X : conj_{old}\}$ is safe with respect to $\mathcal{I}$. Now, the chain of equivalences (7.25) can be used to verify that, for all environments such that $\rho \rhd f(s_1, \ldots, s_n)$,

$$\rho \in \mathcal{I}(s_i \in \overline{se_i}) \text{ implies } \rho \in \mathcal{I}(f(s_1, \ldots, s_n) \in \overline{f(se_1, \ldots, se_n)})$$

Moreover, $\rho \rhd f(s_1, \ldots, s_n)$ implies $\rho \rhd s_i$, for $i = 1..n$. Hence, Proposition 30 can be applied to prove that $\{X : conj \wedge s_i \in se_i\}$ is safe with respect to $\mathcal{I}$. $\square$

The following lemma combines these two propositions with Propositions 26 and 29 to prove the correctness of REDUCE. In essence, this lemma says that REDUCE terminates, and produces reduced form constraints whose least model is the same as the least model of the input constraints. However, it is convenient to prove the lemma in a somewhat more general form because REDUCE is used in a variety of different contexts during the algorithm. Note that it is straightforward to combine the third and fourth parts of the lemma to prove that REDUCE, preserves least models, and this proof is contained in the corollary immediately following the lemma. Also note that the last part of the lemma describes safeness properties of the result of applying reduce, and this is needed because REDUCE will in general be applied many times during the algorithm.

Lemma 14 (Correctness of Reduce) *Let $\mathcal{C}$ be a collection of constraints and let $\mathcal{I}$ be an interpretation. Then:*

- REDUCE *terminates on $\mathcal{C}$;*
- *if $\mathcal{C}$ is in standard form and each quantified condition in $\mathcal{C}$ is of one of the forms (I–IV) then* REDUCE$(\mathcal{C})$ *is in reduced form;*
- *if $v \in \mathcal{I}(se)$ for some constraint $\mathcal{Y} \supseteq se$ in $\mathcal{C}$ then there is a constraint $\mathcal{Y} \supseteq se'$ in* REDUCE$(\mathcal{C})$ *such that $v \in \mathcal{I}(se')$;*
- *if $\mathcal{C}$ is safe w.r.t. $\mathcal{I}$ and $v \in \mathcal{I}(se)$ for some constraint $\mathcal{Y} \supseteq se$ in* REDUCE$(\mathcal{C})$ *then there is a constraint $\mathcal{Y} \supseteq se'$ in $\mathcal{C}$ such that $v \in \mathcal{I}(se')$, and*
- *if $\mathcal{C}$ is safe w.r.t. $\mathcal{I}$ then* REDUCE$(\mathcal{C})$ *is safe w.r.t. $\mathcal{I}$;*

Proof: The two parts of the lemma are just a restatements of Propositions 26 and 29. The remaining three parts essentially follow from repeated applications of Propositions 31 and 32. To summarize these two propositions, let $\mathcal{C}$ be a collection of constraints and consider a single application of one of the steps that make up REDUCE. Let $\mathcal{C}'$ be the result of the application of this step. Note that each step of REDUCE can be thought of as replacing one constraint by a (possibly empty) collection of constraints. Let the replaced constraint be $\mathcal{X} \supseteq se_{\mathcal{X}}$ and let the collection of constraints be $\mathcal{X} \supseteq se_1, \ldots, \mathcal{X} \supseteq se_n$, $n \geq 0$. That is, $\mathcal{C}'$ is $(\mathcal{C} - \{\mathcal{X} \supseteq se_{\mathcal{X}}\}) \cup \{\mathcal{X} \supseteq se_1, \ldots, \mathcal{X} \supseteq se_n\}$. Propositions 31 and 32 imply that:

(a) $\mathcal{I}(se_{\mathcal{X}}) \subseteq \mathcal{I}(se_1 \cup \cdots \cup se_n)$, and

(b) if $\mathcal{C}$ is safe with respect to $\mathcal{I}$ then

  (b1) $\mathcal{I}(se_{\mathcal{X}}) = \mathcal{I}(se_1 \cup \cdots \cup se_n)$, and

  (b2) $se_1, \ldots, se_n$ are safe with respect to $\mathcal{I}$.

Using (a), (b1) and (b2), the following fact can be established: if $\mathcal{C}'$ is obtained from $\mathcal{C}$ by one application of the steps that compose REDUCE then

(1) if $v \in \mathcal{I}(se)$ for some constraint $\mathcal{Y} \supseteq se$ in $\mathcal{C}$ then there is a constraint $\mathcal{Y} \supseteq se'$ in $\mathcal{C}'$ such that $v \in \mathcal{I}(se')$;

(2) if $\mathcal{C}$ is safe w.r.t. $\mathcal{I}$ and $v \in \mathcal{I}(se)$ for some constraint $\mathcal{Y} \supseteq se$ in $\mathcal{C}'$ then there is a constraint $\mathcal{Y} \supseteq se'$ in $\mathcal{C}$ such that $v \in \mathcal{I}(se')$, and

(3) if $\mathcal{C}$ is safe w.r.t. $\mathcal{I}$ then REDUCE($\mathcal{C}$) is safe w.r.t. $\mathcal{I}$.

To prove (1), suppose that $v \in \mathcal{I}(se)$ for some constraint $\mathcal{Y} \supseteq se$ in $\mathcal{C}$. If this constraint is in fact $\mathcal{X} \supseteq se_{\mathcal{X}}$ then $v \in \mathcal{I}\,(se_1 \cup \cdots \cup se_n)$ by (a). Hence, for some i, $v \in se_i$. Since $\mathcal{C}'$ contains $X \supseteq se_i$, $i = 1..n$, the proof is complete. On the other hand, if $\mathcal{Y} \supseteq se$ is different from $\mathcal{X} \supseteq se_{\mathcal{X}}$, then $\mathcal{Y} \supseteq se$ appears in $\mathcal{C}$ and so the proof is immediate.

To prove (2), suppose that $\mathcal{C}$ is safe with respect to $\mathcal{I}$ and suppose that $v \in \mathcal{I}(se)$ for some constraint $\mathcal{Y} \supseteq se$ in $\mathcal{C}$. If this constraint is one of the constraints $\mathcal{X} \supseteq se_i$, $i = 1..n$, then $v \in \mathcal{I}\,(se_1 \cup \cdots \cup se_n)$ and so by (b1), $v \in \mathcal{I}(se_{\mathcal{X}})$. Since $\mathcal{C}$ contains $\mathcal{X} \supseteq se_{\mathcal{X}}$, the proof for this case is complete. On the other hand, if $\mathcal{Y} \supseteq se$ is different from the $X \supseteq se_i$, then $\mathcal{Y} \supseteq se$ appears in $\mathcal{C}$ and so the proof is immediate.

To prove (3), let se be a quantified set expression in $\mathcal{C}'$. If se is one of the se_i, then it follows from (b2) that se is safe with respect to $\mathcal{I}$. On the other hand, if se is not one of the se_i, then se appears in $\mathcal{C}$, and hence se is safe with respect to $\mathcal{I}$ because $\mathcal{C}$ is safe with respect to $\mathcal{I}$.

The proof of the lemma can now be completed by applying this fact to each step performed during REDUCE($\mathcal{C}$) and chaining the results together. $\Box$

Corollary 1 *Let $\mathcal{C}$ be a collection of constraints. If $\mathcal{C}$ is safe with respect to $lm(\mathcal{C})$ then $lm(\text{REDUCE}(\mathcal{C})) = lm(\mathcal{C})$.*

Proof: Let $\mathcal{I}$ denote $lm(\mathcal{C})$ and consider a constraint $\mathcal{X} \supseteq se$ in REDUCE($\mathcal{C}$). Let v be any value in $\mathcal{I}(se)$. By part (c) of Lemma 14 (noting that $\mathcal{C}$ is safe with respect to $\mathcal{I} = lm(\mathcal{C})$), there is a constraint $\mathcal{X} \supseteq se'$ in $\mathcal{C}$ such that

$v \in \mathcal{I}(se')$. Since $\mathcal{I}$ is a model of $\mathcal{C}$, it follows that $v \in \mathcal{I}(\mathcal{X})$. Hence $\mathcal{I}$ is a model of $\mathcal{X} \supseteq se$, and since this was an arbitrary constraint in REDUCE($\mathcal{C}$), $\mathcal{I} \models$ REDUCE($\mathcal{C}$). Thus $lm(\mathcal{C}) \supseteq lm(\text{REDUCE}(\mathcal{C}))$.

Conversely, let $\mathcal{I}$ denote $lm(\text{REDUCE}(\mathcal{C}))$ and consider a constraint $\mathcal{X} \supseteq se$ in $\mathcal{C}$. If $v \in \mathcal{I}(se)$ then by part (b) of Lemma 14 (noting that safeness is not required here) there is a constraint $\mathcal{X} \supseteq se'$ in REDUCE($\mathcal{C}$) such that $v \in \mathcal{I}(se')$. Since $\mathcal{I}$ is a model of REDUCE($\mathcal{C}$), it follows that $v \in \mathcal{I}(\mathcal{X})$. Hence $\mathcal{I}$ is a model of each constraint in $\mathcal{C}$, and so $lm(\mathcal{C}) \subseteq lm(\text{REDUCE}(\mathcal{C}))$. $\square$

This completes the proof of correctness for REDUCE. Now, the set constraints $\mathcal{SC}_P$ for a program P must be initially put into reduced form. This can be achieved by first applying STANDARDIZE (to put them into standard form) and then applying REDUCE (to put them into reduced form). We now prove that this initialization process is correct. The main part of this is to prove that $\mathcal{SC}_P$ is safe with respect to $lm(\mathcal{SC}_P)$.

Lemma 15 (Initialization) *Let $\mathcal{SC}_P$ be the set constraints for a program P, and let $\mathcal{C}_0$ be* REDUCE(STANDARDIZE($\mathcal{SC}_P$)). *Then*

(a) $\mathcal{C}_0$ is in reduced form;

(b) $lm(\mathcal{C}_0) =_{var(\mathcal{SC}_P)} lm(\mathcal{SC}_P)$.

(c) $\mathcal{C}_0$ is safe with respect to $lm(\mathcal{C}_0)$.

Proof: First consider (a). By proposition 14, we only need to show that each quantified condition in STANDARDIZE($\mathcal{SC}_P$) is of one of the forms (I–IV). Now, it has already been argued that the quantified conditions in $\mathcal{SC}_P$ are of one of the forms (I–IV). When STANDARDIZE to $\mathcal{SC}_P$ the only step that may alter quantified set expressions is the step that replaces a set expression by a new set variable and adds a constraint between the new set variable and the replaced expression. It follows that STANDARDIZE preserves the form (I–IV).

To prove the remaining parts of the lemma, it must first be established that $\mathcal{SC}_P$ is safe with respect to $lm(\mathcal{SC}_P)$. Recall the construction of $\mathcal{SC}_P$ from section 6.2, page 152, where set constraints are described for each of the five different kinds of environment constraints. In most cases it is easy to see that the quantified set expressions introduced are safe with respect to $lm(\mathcal{SC}_P)$ because they contain only quantified conditions of the form $X \in se$ or $s \in a$ where a contains only function symbols. There are only three non-trivial cases.

The first involves the set constraints

$$\mathcal{X}_i \quad \supseteq \quad \{X_i : \mathit{defined}(t) \ \wedge \ \bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda\}$$

introduced during the translation of $\Psi^\mu \supseteq \Psi^\lambda[X_l{\mapsto}t]$, where $\mathit{defined}(t)$ denotes the conjunction of all conditions of the form $s \in f(\top, \ldots, \top)$ such that $f_{(i)}^{-1}(s)$ is a subterm of t. Let conj denote $\mathit{defined}(t) \wedge \bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda$. To show that the quantified set expressions $\{X_i : \mathit{conj}\}$ are safe with respect to $lm(\mathcal{SC}_P)$, consider an environment ρ such that $\rho \notin lm(\mathcal{SC}_P)(\mathit{conj})$. Clearly there must exist at least one quantified condition $s \in se$ in conj such that $\rho \notin lm(\mathcal{SC}_P)(s \in se)$. Pick the quantified condition such that the number of function symbols in s is minimized. Now, if it is not the case that $\rho \rhd s$, then s must contain a subterm of the form $f_{(i)}^{-1}(s')$ such that either $\rho(s')$ is not defined or else $\rho(s')$ is not of the form $f(\cdots)$. This means that $s' \in f(\top, \ldots, \top)$ must appear in conj and that $\rho \notin \mathcal{I}(s' \in f(\top, \ldots, \top))$. However s' contains fewer function symbols than s, and this contradicts the choice of s. Hence it must be the case that $\rho \rhd s$, and this completes the proof that each quantified set expression $\{X_i : \mathit{defined}(t) \ \wedge \ \bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda\}$ is safe with respect to $lm(\mathcal{SC}_P)$.

The second non-trivial case involves the set constraints

$$\mathcal{X}_i \quad \supseteq \quad \{X_i : \mathit{defined}(\mathit{conj}_k) \ \wedge \ \bigwedge_{j=1..m} X_j \in \mathcal{X}_j^\lambda\}$$

introduced during the translation of $\Psi^\mu \supseteq \Psi^\lambda[\mathit{conj}_1 \vee \cdots \vee \mathit{conj}_n]$. The proof for the quantified conditions in these constraints is identical to the first case.

The third non-trivial case involves the set constraints

$$\mathcal{X}_i^\mu \quad \supseteq \quad \{X_i : \mathit{translate}(\mathit{conj}_k) \ \wedge \ \bigwedge_{j=1..m} X_j \in \mathcal{X}_j\}$$

introduced during the translation of $\Psi^\mu \supseteq \Psi^\lambda[\mathit{conj}_1 \vee \cdots \vee \mathit{conj}_n]$. Let $\mathcal{I}$ denote $lm(\mathcal{SC}_P)$. The safeness of these quantified set expressions relies on a combination of two factors: first, the quantified conditions $X_j \in \mathcal{X}_j$, $j = 1..m$, and second, properties of the sets assigned to the $\mathcal{X}_j$ under $\mathcal{I}$.

These two factors are combined in the property

$$\text{if} \quad \rho \in \mathcal{I}\left(\bigwedge_{j=1..m} X_j \in \mathcal{X}_j\right) \quad \text{then} \quad \rho \rhd \mathit{conj}_k \tag{7.27}$$

which clearly implies that $\{X_i : \mathit{translate}(\mathit{conj}_k) \wedge \bigwedge_{j=1..m} X_j \in \mathcal{X}_j\}$ is safe with respect to $\mathcal{I}$. To prove (7.27), consider the values of the variables $\mathcal{X}_j$ under $\mathcal{I}$. Now, the constraints for these variables are

$$\mathcal{X}_j \;\supseteq\; \{X_j : \mathit{defined}(\mathit{conj}_k) \;\wedge\; \bigwedge_{l=1..m} X_l \in \mathcal{X}_l^{\lambda}\}$$

and moreover, there is only one lower bound for each variable $\mathcal{X}_j$. Hence, by Proposition 18, these inequalities are in fact equalities in $\mathcal{I}$. That is,

$$\mathcal{I}(\mathcal{X}_j) \;=\; \mathcal{I}(\{X_j : \mathit{defined}(\mathit{conj}_k) \wedge \bigwedge_{l=1..m} X_l \in \mathcal{X}_l^{\lambda}\}), \; j = 1..m \tag{7.28}$$

Using the equality (7.28), the expression $\rho \in \mathcal{I}(\bigwedge_{j=1..m} X_j \in \mathcal{X}_j)$ can be significantly simplified. Recall that $X_1, \ldots, X_m$ is a list of the program variables, and let ϱ denote the set environment that maps the program variables X_j into $\mathcal{I}(\mathcal{X}_j)$. Now, consider the following chain of equivalences:

$$\begin{array}{ll}
& \rho \in \mathcal{I}\left(\bigwedge_{j=1..m} X_j \in \mathcal{X}_j\right) \\
\text{iff} & \displaystyle\bigwedge_{j=1..m} \rho(X_j) \in \mathcal{I}(\mathcal{X}_j) \\
\text{iff} & \displaystyle\bigwedge_{j=1..m} \rho(X_j) \in \mathcal{I}(\{X_j : \mathit{defined}(\mathit{conj}_k) \;\wedge\; \bigwedge_{j=1..m} X_j \in \mathcal{X}_j^{\lambda}\}) \\
\text{iff} & \displaystyle\bigwedge_{j=1..m} \rho(X_j) \in \{\rho(X_j) : \rho \in \mathcal{I}(\mathit{defined}(\mathit{conj}_k)) \;\wedge\; \bigwedge_{j=1..m} \rho(X_j) \in \mathcal{I}(\mathcal{X}_j^{\lambda})\} \\
\text{iff} & \displaystyle\bigwedge_{j=1..m} \rho(X_j) \in \{\rho(X_j) : \rho \in \mathcal{I}(\mathit{defined}(\mathit{conj}_k)) \;\wedge\; \rho \in \varrho\} \\
\text{iff} & \rho \in \mathcal{A}\left(\{\rho \in \varrho : \rho \in \mathcal{I}(\mathit{defined}(\mathit{conj}_k))\}\right) \\
\text{iff} & \rho \in \mathcal{A}\left(\{\rho \in \varrho : \rho(s) \in \mathcal{I}(se) \text{ for each } s \in se \text{ in } \mathit{defined}(\mathit{conj}_k)\}\right) \\
\text{iff} & \rho \in \mathcal{A}\left(\{\rho \in \varrho : \rho(s) \text{ has form } f(\cdots) \text{ for each term } f_{(i)}^{-1}(s) \text{ in } \mathit{cond}_k\}\right) \\
\text{iff} & \rho \in \mathcal{A}\left(\{\rho \in \varrho : \rho \rhd \mathit{cond}_k\}\right)
\end{array}$$

Now, by Proposition 14, $\{\rho \in \varrho : \rho \rhd \mathit{cond}_k\}$ is set based. Hence the condition $\rho \in \mathcal{A}(\{\rho \in \varrho : \rho \rhd \mathit{cond}_k\})$ is equivalent to $\rho \rhd \mathit{cond}_k$. This means that $\rho \in \mathcal{I}\left(\bigwedge_{j=1..m} X_j \in \mathcal{X}_j\right)$ implies that $\rho \rhd \mathit{cond}_k$, and this completes the proof of (7.27). This completes the proof that all of the quantified set expressions appearing in $\mathcal{SC}_P$ are safe with respect to $lm(\mathcal{SC}_P)$.

Consider the application of STANDARDIZE to $\mathcal{SC}_P$. Suppose that $\mathcal{C}'$ is obtained form $\mathcal{C}$ by a single step of STANDARDIZE and that $\mathcal{C}$ is safe with respect to $lm(\mathcal{C})$. It has already been proved that $lm(\mathcal{C})(\mathcal{X}) = lm(\mathcal{C}')(\mathcal{X})$ for each $\mathcal{X} \in var(\mathcal{C})$ (see the proof of Proposition 21). Now, suppose that this step of does not introduce any new quantified set expressions, and this implies that $\mathcal{C}'$ is safe with respect to $lm(\mathcal{C})$. Since the safety of a quantified set expressions se with respect to an interpretation only depends on the set variables appearing in se, it follows that $\mathcal{C}'$ is safe with respect to $lm(\mathcal{C})$.

Now consider the case where the step of STANDARDIZE *does* introduce new quantified set expressions. As has already been noted, the only step that can do this is the step that replaces a non-standard occurrence with a new variable. Hence, $\mathcal{C}$ must contain a quantified set expression $\{X : conj \wedge exp\}$ such that exp is either $s \in se$ or $s \dagger se$ and the new quantified set expression in $\mathcal{C}'$ is $\{X : conj \wedge exp'\}$ where exp' is $s \in \mathcal{Z}$ or $s \dagger \mathcal{Z}$ and $\mathcal{Z}$ is a new set variable. Moreover, $\mathcal{Z} \supseteq se$ must be the only lower bound for $\mathcal{Z}$ in $\mathcal{C}'$. Proposition 18 implies that $lm(\mathcal{C}')(\mathcal{Z}) = lm(\mathcal{C}')(se)$ and so $lm(\mathcal{C}')(exp) = lm(\mathcal{C}')(exp')$. Since $\{X : conj \wedge exp\}$ contains only variables from $\mathcal{C}$, and $\{X : conj \wedge exp\}$ is safe with respect to $lm(\mathcal{C})$, it follows that $\{X : conj \wedge exp\}$ is safe with respect to $lm(\mathcal{C}')$. In summary,

- $lm(\mathcal{C}')(exp) = lm(\mathcal{C}')(exp')$;
- $\rho \rhd exp$ iff $\rho \rhd exp'$ (this is easy to verify), and
- $\{X : conj \wedge exp\}$ is safe with respect to $lm(\mathcal{C}')$.

Hence the three preconditions of Proposition 30 hold. It follows that $\{X : conj \wedge exp'\}$ is safe with respect to $lm(\mathcal{C}')$.

This proves that a single step of STANDARDIZE preserves safeness with respect to the least model. Repeatedly applying this fact proves that if $\mathcal{C}$ is safe with respect to $lm(\mathcal{C})$ then STANDARDIZE($\mathcal{C}$) is safe with respect to lm(STANDARDIZE($\mathcal{C}$)).

Now, it was previously shown that $\mathcal{SC}_P$ is safe with respect to $lm(\mathcal{SC}_P)$, and so STANDARDIZE($\mathcal{SC}_P$) is safe with respect to lm(STANDARDIZE($\mathcal{SC}_P$)). Hence Corollary 1 can be applied to prove that

$$lm(\text{STANDARDIZE}(\mathcal{SC}_P)) = lm(\text{REDUCE}(\text{STANDARDIZE}(\mathcal{SC}_P))).$$

By the correctness of STANDARDIZE (Proposition 21), $lm(\mathcal{SC}_P) =_{var(\mathcal{SC}_P)} lm(\text{STANDARDIZE}(\mathcal{SC}_P))$. Hence

$$\begin{array}{lll} lm(\mathcal{SC}_P) & =_{var(\mathcal{SC}_P)} & lm(\text{STANDARDIZE}(\mathcal{SC}_P)) \\ & = & lm(\text{REDUCE}(\text{STANDARDIZE}(\mathcal{SC}_P))) \\ & = & lm(\mathcal{C}_0). \end{array}$$

Furthermore, since the constraints STANDARDIZE($\mathcal{SC}_P$) are safe with respect to $lm(\text{STANDARDIZE}(\mathcal{SC}_P))$, Lemma 14 also implies that $\mathcal{C}_0$ is safe with respect to $lm(\mathcal{C}_0)$. □

Transformations

We have just shown how set constraints $\mathcal{SC}_P$ can be converted into equivalent constraints $\mathcal{C}_0$ that are in standard form and reduced form. We now present an instance of the generic algorithm for obtaining the least model of the constraints $\mathcal{C}_0$. The instance of the generic algorithm is defined by the following collection of transformations. The first group of transformations deal with substitution.

Transformation 5 (Qexp-Substitution) *If $\mathcal{C}$ contains the two constraints $\mathcal{X} \supseteq \{X : (s \in \mathcal{Y}) \wedge conj\}$ and $\mathcal{Y} \supseteq a$ where a is atomic, then output* REDUCE($\mathcal{X} \supseteq \{X : (s \in a) \wedge conj\}$). □

Transformation 6 ($\bigcap$-Substitution) *If $\mathcal{C}$ contains the two constraints $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_{i-1} \cap \mathcal{Y} \cap a_{i+1} \cap \cdots \cap a_n$ and $\mathcal{Y} \supseteq a$ where a is atomic and $n \geq 2$, then output $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_{i-1} \cap a \cap a_{i+1} \cap \cdots \cap a_n$.* □

Transformation 7 (Var-Substitution) *If $\mathcal{C}$ contains $\mathcal{Y} \supseteq a$ and $\mathcal{X} \supseteq \mathcal{Y}$ and where a is atomic, then output $\mathcal{X} \supseteq a$.* □

We remark that the notion of substitution described by these transformations is more restrictive than the substitution used in the intersection-projection algorithm. In particular, recall that Transformation 6 substituted for any set variable $\mathcal{Y}$ appearing in an expression $op(a_1, \ldots, a_{i-1}, \mathcal{Y}, a_{i+1}, a_n)$.

If this very general notion of substitution was carried over to quantified set expressions, then there would be a substitution for the set variable $\mathcal{Y}$ in the constraint $\mathcal{X} \supseteq \{X : s \dagger \mathcal{Y}\}$. However the above transformations do not admit substitutions into quantified conditions of the form $s \dagger \mathcal{Y}$ and, as we shall see later, this is specifically required for termination.

Note that in the first transformation, the expression $\{X : (s \in a) \wedge conj\}$ may not be in reduced form. Hence REDUCE must be applied. For example, consider the constraints

$$\begin{aligned} \mathcal{X} &\supseteq \{X : g(X,X) \in \mathcal{Y}\} \\ \mathcal{Y} &\supseteq \overline{g(b,c)}. \end{aligned}$$

where b and c are constants and g is binary function symbol. When the second constraint is substituted into the first, the resulting constraint $\mathcal{X} \supseteq \left\{X : g(X,X) \in \overline{g(b,c)}\right\}$ is not in reduced form. The subsequent application of REDUCE results in the following reduced form constraints.

$$\begin{aligned} \mathcal{X} &\supseteq \left\{X : X \in \overline{b}\right\} \\ \mathcal{X} &\supseteq \{X : X \in \overline{c}\} \end{aligned}$$

The second group of transformations deal with simplifying projections. We note that there are two possible approaches to dealing with projections. We could just extend the transformation for projections from the projection-intersection algorithm (see Transformation 3, page 182) to deal with the additional cases involving $\top$ and constants of the form $\overline{S}$. However, since projections are essentially special cases of quantified set expressions, another approach is to convert them to quantified set expressions. We choose the latter approach for presentational simplicity, since it avoids some duplication of work. (However, note that their are implementation reasons for distinguishing between arbitrary quantified set expressions and the special case of projections.) For projections, we therefore have the single transformation:

Transformation 8 (Projection) *If $\mathcal{C}$ contains $\mathcal{X} \supseteq f_{(i)}^{-1}(a)$ then output* REDUCE$(\mathcal{X} \supseteq \{X_i : f(X_1, \ldots, X_n) \in a\})$ *where the arity of f is n and $X_1, \ldots, X_n$ are distinct program variables.* □

It is assumed that the $X_1, \ldots, X_n$ are chosen in some canonical manner (for example, using some fixed listing of VAR) so that this transformation cannot be repeatedly applied to produce REDUCE($\mathcal{X} \supseteq \{X_i : f(X_1, \ldots, X_n) \in a\}$), and then REDUCE($\mathcal{X} \supseteq \{X_i' : f(X_1', \ldots, X_n') \in a\}$) etc.

The next group of transformation deal with intersection. These transformations generalize the intersection transformation used in the projection-intersection algorithm (see Transformation 4, page 182) to deal with the $\top$ and $\overline{S}$.

Transformation 9 (Intersection-1) *If $\mathcal{C}$ contains $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$, $m \geq 2$, then let $\overline{S_1}, \ldots, \overline{S_n}$ and $a_1', \ldots, a_{m-n}'$ be subsequences of $a_1, \ldots, a_m$ such that the first subsequence contains the complement constants in $a_1, \ldots, a_m$, and the second contains the remaining atomic set expressions, and output the constraint $\mathcal{X} \supseteq a_1' \cap \cdots \cap a_{m-n}' \cap \overline{S}$ where $S = S_1 \cup \cdots \cup S_n$.* $\square$

Transformation 10 (Intersection-2) *If $\mathcal{C}$ contains $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m \cap \overline{S}$ such that $m \geq 2$, and each a_i is of the form $f(a_{i,1}, \ldots, a_{i,n})$, then let $N_j = \bigcup_{i=1..m} \mathcal{N}(a_{i,j})$, $j = 1..n$, and output the constraints*

- $\mathcal{X} \supseteq f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n}) \cap \overline{S}$, *and*
- $\mathcal{V}_{N_j} \supseteq a_{1,j} \cap \cdots \cap a_{m,j}$ *for each* $\mathcal{V}_{N_j} \notin var(\mathcal{C})$. $\square$

Transformation 11 (Intersection-3) *If $\mathcal{X} \supseteq f(a_1, \ldots, a_n) \cap \overline{S}$ appears in $\mathcal{C}$ and $f(\top, \ldots, \top) \not\subseteq S$ then*

(a) if $f'(\cdots) \in S$ implies $f \neq f'$ then output $\mathcal{X} \supseteq f(a_1, \ldots, a_n)$;

(b) otherwise, pick an element of the form $f(se_1, \ldots, se_n)$ from S, let S' be $S - \{f(se_1, \ldots, se_n)\}$, let N_j be $\mathcal{N}(a_j) \cup \{\overline{se_j}\}$, $j = 1..n$, and output the constraints:

- $\mathcal{X} \supseteq f(a_1, \ldots, a_{j-1}, \mathcal{V}_{N_j}, a_{j+1}, \ldots, a_n) \cap \overline{S'}$, $j = 1..n$, *and*
- $\mathcal{V}_{N_j} \supseteq a_i \cap \overline{se_i}$ *for each* $\mathcal{V}_{N_j} \notin var(\mathcal{C})$. $\square$

The first transformation serves to collect constants of the form $\overline{S}$ together so that constraints of the form $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_k \cap \overline{S_1} \cap \cdots \cap \overline{S_n}$ are converted into the form $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_k \cap \overline{S}$. Note that in the boundary case where

$n = 0$, S is the empty set and the constraint $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_k \cap \top$ is constructed. The second transformation then combines expressions of the form $f(\cdots) \cap \cdots \cap f(\cdots)$ into $f(\cdots)$. The final transformation deals with the interaction of expressions of the form $f(\cdots)$ and $\overline{S}$. Note that these transformations also simplify constraints involving $\top$ because $\top$ is identified with $\overline{\{\ \}}$. Hence Transformation 11 simplifies $\mathcal{X} \supseteq a \cap \top$ into $\mathcal{X} \supseteq a$.

To illustrate the behavior of these transformations, consider the constraints

$$\begin{array}{lcl} \mathcal{X} & \supseteq & g(b,b) \cap g(\mathcal{Y},\mathcal{Y}) \cap \overline{\{g(b,c)\}} \cap \overline{\{g(c,b)\}} \\ \mathcal{Y} & \supseteq & b. \end{array}$$

In particular, note that in the least model of these constraints, $g(b,b)$ is an element of $\mathcal{X}$ (in fact it is the only element of $\mathcal{X}$). We now show how the intersection transformations add to these constraints to make this fact explicit. First Transformation 9 is applied to obtain

$$\mathcal{X} \supseteq g(b,b) \cap g(\mathcal{Y},\mathcal{Y}) \cap \overline{\{g(b,c), g(c,b)\}}$$

Transformation 10 can now be applied to this constraint to yield:

$$\begin{array}{lcl} \mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y}\}}, \mathcal{V}_{\{b,\mathcal{Y}\}}) \cap \overline{\{g(b,c), g(c,b)\}} \\ \mathcal{V}_{\{b,\mathcal{Y}\}} & \supseteq & b \cap \mathcal{Y} \end{array}$$

A subsequent application of Transformation 10 to $\mathcal{V}_{\{b,\mathcal{Y}\}} \supseteq b \cap \mathcal{Y}$ yields $\mathcal{V}_{\{b,\mathcal{Y}\}} \supseteq b$. Also, Transformation 11 can be applied to $\mathcal{X} \supseteq g(\mathcal{V}_{\{b,\mathcal{Y}\}}, \mathcal{V}_{\{b,\mathcal{Y}\}}) \cap \overline{\{g(b,c), g(c,b)\}}$ to obtain:

$$\begin{array}{lcl} \mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y},\overline{b}\}}, \mathcal{V}_{\{b,\mathcal{Y}\}}) \cap \overline{g(c,b)} \\ \mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y}\}}, \mathcal{V}_{\{b,\mathcal{Y},\overline{c}\}}) \cap \overline{g(c,b)} \\ \mathcal{V}_{\{b,\mathcal{Y},\overline{b}\}} & \supseteq & \mathcal{V}_{\{b,\mathcal{Y}\}} \cap \overline{b} \\ \mathcal{V}_{\{b,\mathcal{Y},\overline{c}\}} & \supseteq & \mathcal{V}_{\{b,\mathcal{Y}\}} \cap \overline{c} \end{array}$$

Applying Transformation 11 to the first two of these constraints yields:

$$
\begin{array}{rcl}
\mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y},\bar{b},\bar{c}\}}, \mathcal{V}_{\{b,\mathcal{Y}\}}) \\
\mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y},\bar{b}\}}, \mathcal{V}_{\{b,\mathcal{Y},\bar{b}\}}) \\
\mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y},\bar{c}\}}, \mathcal{V}_{\{b,\mathcal{Y},\bar{c}\}}) \\
\mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y}\}}, \mathcal{V}_{\{b,\mathcal{Y},\bar{b},\bar{c}\}}) \\
\mathcal{V}_{\{b,\mathcal{Y},\bar{b},\bar{c}\}} & \supseteq & \mathcal{V}_{\{b,\mathcal{Y},\bar{b}\}} \cap \bar{c}
\end{array}
$$

Finally, consider applying transformations to the constraints for $\mathcal{V}_{\{b,\mathcal{Y},\bar{b}\}}$, $\mathcal{V}_{\{b,\mathcal{Y},\bar{c}\}}$ and $\mathcal{V}_{\{b,\mathcal{Y},\bar{b},\bar{c}\}}$. This yields the single constraint

$$
\mathcal{V}_{\{b,\mathcal{Y},\bar{c}\}} \supseteq b.
$$

To summarize, the collection now contains the following explicit form constraints:

$$
\begin{array}{rcl}
\mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y},\bar{b},\bar{c}\}}, \mathcal{V}_{\{b,\mathcal{Y}\}}) \\
\mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y},\bar{b}\}}, \mathcal{V}_{\{b,\mathcal{Y},\bar{b}\}}) \\
\mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y},\bar{c}\}}, \mathcal{V}_{\{b,\mathcal{Y},\bar{c}\}}) \\
\mathcal{X} & \supseteq & g(\mathcal{V}_{\{b,\mathcal{Y}\}}, \mathcal{V}_{\{b,\mathcal{Y},\bar{b},\bar{c}\}}) \\
\mathcal{Y} & \supseteq & b \\
\mathcal{V}_{\{b,\mathcal{Y}\}} & \supseteq & b \\
\mathcal{V}_{\{b,\mathcal{Y},\bar{c}\}} & \supseteq & b
\end{array}
$$

Hence, $g(b, b) \in \mathcal{X}$ is now explicit (that is, $g(b, b)$ is now an element of $\mathcal{X}$ in the least model of the *explicit*$(\mathcal{C})$).

The final group of transformations deals with quantified set expressions. The first two transformations serve to remove apartness conditions. One deals with $s \dagger a$ in the case where a is a singleton set under the interpretation $lm(\mathit{explicit}(\mathcal{C}))$, and the other deals with the case where a contains more than one element. The last transformation replaces a quantified set expression with an intersection of atomic set expressions, and for this transformation some preliminary definitions are needed. Define that a conjunction of quantified conditions *conj* is in *variable-expression* form if each quantified condition in *conj* is of the form $X \in a$ where X is a program variable and a is an atomic set expressions. For such a conjunction, let X be a program variable appearing in *conj* and define that $\textcircled{\cap}_X\, conj$ is $a_1 \cap \cdots \cap a_n$ where $X \in a_1, \ldots, X \in a_n$ lists all of the quantified conditions in *conj* that have the form $X \in a$.

Transformation 12 (Qexp-†-1) *If $\mathcal{C}$ contains $\mathcal{X} \supseteq \{X : s \dagger a \wedge conj\}$ and $lm(explicit(\mathcal{C}))(a) = \{v\}$ then output* REDUCE$(\mathcal{X} \supseteq \{X : s \in \overline{v} \wedge conj\})$. $\square$

Transformation 13 (Qexp-†-2) *If $\mathcal{C}$ contains $\mathcal{X} \supseteq \{X : s \dagger a \wedge conj\}$ and $lm(explicit(\mathcal{C}))(a)$ contains more than one element then output the constraint $\mathcal{X} \supseteq \{X : conj\}$.* $\square$

Transformation 14 (Qexp-Compaction) *If $\mathcal{C}$ contains $\mathcal{X} \supseteq \{X : conj\}$ such that conj is in compaction form and $lm(explicit(\mathcal{C}))(\bigcap_{\mathrm{Y}} conj)$ is non-empty for each program variable Y appearing in conj, then output the constraint $\mathcal{X} \supseteq \bigcap_{\mathrm{X}} conj$.* $\square$

To illustrate this group of transformations, consider a collection of constraints $\mathcal{C}$ that contains the following constraints:

$$\begin{array}{lll}\mathcal{X} & \supseteq & \{X : f(X,X) \in \mathcal{Y} \quad \wedge \quad X \dagger \mathcal{Z}\} \\ \mathcal{Y} & \supseteq & f(b,c) \\ \mathcal{Y} & \supseteq & f(d,d) \\ \mathcal{Y} & \supseteq & f(e,e)\end{array}$$

Suppose that there are additional constraints for $\mathcal{Y}$ and that, in the least model of *explicit*$(\mathcal{C})$, $\mathcal{Y}$ is $\{e\}$. We now show how transformations can be applied so that $d \in \mathcal{X}$ in the least model becomes explicit. First, Transformation 12 can be applied to produce

$$\mathcal{X} \quad \supseteq \quad \{X : \quad f(X,X) \in \mathcal{Y} \quad \wedge \quad X \in \overline{e}\} \ .$$

Now, the constraints for $\mathcal{Y}$ can all be used to substitute for the occurrence of $\mathcal{Y}$ in $\mathcal{X} \supseteq \{X : \ f(X,X) \in \mathcal{Y} \ \wedge \ X \in \overline{e}\}$, and the result is the following constraints:

$$\begin{array}{lll}\mathcal{X} & \supseteq & \{X : \quad X \in b \quad \wedge \quad X \in c \quad \wedge \quad X \in \overline{e}\} \\ \mathcal{X} & \supseteq & \{X : \quad X \in d \quad \wedge \quad X \in \overline{e}\} \\ \mathcal{X} & \supseteq & \{X : \quad X \in e \quad \wedge \quad X \in \overline{e}\}\end{array}$$

The compaction transformation can be applied to these three constraints to obtain:

$$\begin{array}{lll}\mathcal{X} & \supseteq & b \cap c \cap \overline{e} \\ \mathcal{X} & \supseteq & d \cap \overline{e} \\ \mathcal{X} & \supseteq & e \cap \overline{e}\end{array}$$

On applying the intersection transformation (Transformation 10) to this last constraint, the constraint $\mathcal{X} \supseteq d$ is obtained, and thus $d \in \mathcal{X}$ becomes explicit.

Finally, we can define the algorithm for solving quantified set expressions. Let Δ_2 denote Transformations 5–14, and define that the *quantified expression algorithm* inputs set constraints $\mathcal{C}$, converts $\mathcal{C}$ into constraints $\mathcal{C}_0$ by applying STANDARDIZE and then REDUCE, and then exhaustively applies the transformations Δ_2 to C_0 as outlined by the generic algorithm.

Correctness

The proof of correctness of the quantified set expression algorithm is fairly similar in structure to that for the intersection-projection algorithm described in the previous section. The main differences are that atomic set expressions now include $\top$ and $\overline{S}$, and the presence of quantified set expressions. We shall often omit proof details for cases that are essentially the same as those in the previous section, and instead focus on the new cases. We begin by considering a generalization of the atomic set expression invariant employed in the intersection-projection algorithm (see Proposition 22, page 185).

Invariant 3 (Atomic Set Expression Invariant) *A collection $\mathcal{C}$ of constraints satisfies the atomic set expression invariant if each atomic set expressions in atomic$(\mathcal{C})$ either*

(i) appears in atomic$(\mathcal{C}_0)$;

(ii) is introduced by an application of Transformation 12;

(iii) is of the form $f(a_1, \ldots, a_n)$ where f is a function symbol appearing in $\mathcal{C}_0$, and each a_i is either an intersection variable or a strict subterm of some atomic set expression that falls into cases (i) or (ii); or

(iv) is of the form $\overline{S}$ such that $S \subseteq$ atomic$(S_1 \cup \cdots \cup S_n)$ for some complement constants $\overline{S_1}, \ldots, \overline{S_k}$ satisfying cases (i) or (ii) of the invariant.
□

This main changes in the (modified) atomic expression invariant are due to Transformations 9, 11 and 12, which may introduce new atomic set expressions of the form $\overline{S}$. It is convenient to prove this invariant in tandem with two additional properties: during the algorithm all constraints are both in standard form and reduced form.

Proposition 33 (Invariants) *Each $\mathcal{C}_i$ constructed by the algorithm is in reduced form and standard form, and satisfies the atomic set expression invariant.*

Proof: By the Initialization Lemma (Lemma 15), the initial constraints $\mathcal{C}_0 = \textsc{reduce}(\textsc{standardize}(\mathcal{SC}_P))$ are in reduced form and standard form. It is easy to verify, that each transformation preserves reduced form and standard form, noting that whenever a transformation may construct constraints that are not in reduced form, the procedure REDUCE is immediately applied to return the constraints to reduced form. This completes the first part of the proof.

Now consider the atomic expression invariant. Clearly $\mathcal{C}_0$ satisfies the atomic set expression invariant, and it therefore remains to prove that each transformation preserves this invariant. To this end, let $\mathcal{C}$ be a collection of constraints that satisfies the atomic set expression invariant, and consider the sets $\delta(\mathcal{C})$ for each transformation δ in Δ_2. First suppose that δ is one of Transformations 6, 7, 13 and 14. In all of these cases, it is clear that $atomic(\delta(\mathcal{C})) \subseteq atomic(\mathcal{C})$, and so the proof is trivial.

Before considering the remaining transformations, it is useful to outline some properties of the atomic set expression invariant. Suppose that $\mathcal{C}$ satisfies the atomic set expression invariant. Now, note that condition (iii) can only be satisfied by an atomic set expression of the form $f(\cdots)$ and that condition (iv) of this invariant can only be satisfied by a complement constant. Hence, if $f(\cdots)$ is an atomic set expression appearing in $\mathcal{C}$ then it must satisfy one of conditions (i–iii) of the invariant. Similarly, if $\overline{S}$ appears in $\mathcal{C}$ then it must satisfy one of conditions (i), (ii) and (iv), and this implies that there are constants $\overline{S_1}, \ldots, \overline{S_k}$ that satisfy parts (i) or (ii) of the invariant such that $S \subseteq atomic(S_1 \cup \cdots \cup S_n)$.

Transformation 5:. Let se_{old} denote $\{X : (s \in \mathcal{Y}) \wedge conj\}$ and let se_{new} denote $\{X : (s \in a) \wedge conj\}$. Note that a is a non-variable atomic set expression. Since $\mathcal{X} \supseteq se_{old}$ and $\mathcal{Y} \supseteq a$ both satisfy the atomic set expression invariant, it follows that $\mathcal{X} \supseteq se_{new}$ also satisfies this invariant. If s is a program variable, then $\mathcal{X} \supseteq se_{new}$ is in reduced form, and REDUCE has no effect. Hence in this case $\delta(\mathcal{C})$ trivially satisfies the atomic set expression invariant.

In the remaining case, s is not a program variable, and REDUCE is needed to return the constraint $\mathcal{X} \supseteq se_{new}$ to reduced form. Recall that REDUCE is defined to be the exhaustive application of the six steps show in Figure 7.5, page 199. Clearly these steps do not affect the quantified conditions in *conj*. Hence, the application of REDUCE to $\mathcal{X} \supseteq se_{new}$ can only involve applications of steps to the quantified condition $s \in a$ and any new quantified conditions produced by such steps. It is easy to verify that all new quantified conditions produced must have the form $s' \in a'$ where s' is a subterm of s and a' is either a subterm of a or else of the form $\overline{S}$ where there exists a constant $\overline{S'}$ in a such that $S \subseteq atomic(S')$. (Note that step (vi) cannot be applied during REDUCE($\mathcal{X} \supseteq se_{new}$), and this is the only step that builds up completely new set expressions.) It follows that each element of $atomic(\delta(\mathcal{C}))$ either (a) appears in $atomic(\mathcal{C})$ or (b) has the form $\overline{S}$ such that $atomic(\mathcal{C})$ contains a constant $\overline{S'}$ and $S \subseteq atomic(S')$. By combining this with the assumption that $\mathcal{C}$ satisfies the atomic set expression invariant, it is easy to verify that REDUCE($\mathcal{X} \supseteq se_{new}$) satisfies the atomic set expression invariant.

Transformation 8: Since $\mathcal{C}$ satisfies the atomic set expression invariant it is clear that $\mathcal{X} \supseteq \{X_i : f(X_1, \ldots, X_n) \in a\}$ also satisfies the atomic set expression invariant. The argument that $\delta(\mathcal{C}) =$ REDUCE($\mathcal{X} \supseteq \{X_i : f(X_1, \ldots, X_n) \in a\}$) also satisfies this invariant is identical that for Transformation 5.

Transformation 9: This transformation may introduce new atomic set expressions of the form $\overline{S}$ such that $\overline{S_1}, \ldots, \overline{S_n}$ appear in $atomic(\mathcal{C})$ and $S = S_1 \cup \cdots \cup S_n$. It is immediate that $\overline{S}$ satisfies part (iv) of the invariant.

Transformation 10: This transformation may introduce new atomic set expressions of the form $f(\mathcal{V}_{N_1} \ldots, \mathcal{V}_{N_n})$ such that each $\mathcal{V}_{N_i}$ is an intersection variable and f is a function symbol and appearing in $\mathcal{C}$ (and hence in $\mathcal{C}_0$, since no transformation can introduce new function symbols). Such expressions satisfy part (iii) of the atomic set expression invariant.

Transformation 11: The new atomic set expressions introduced by this transformation are either of the form (a) $f(a_1, \ldots, a_n)$ such that each a_i is an intersection variable or a strict subterm of an atomic set expression appearing in $atomic(\mathcal{C})$, or (b) $\overline{se_i}$ such that $atomic(\mathcal{C})$ contains an expression of the form $\overline{S}$ where $f(se_1, \ldots, se_i, \ldots, se_n) \in S$. First consider case (a). Since $\mathcal{C}$ satisfies the atomic expression invariant, it follows that if a_i is a strict subterm of an atomic set expression appearing in $atomic(\mathcal{C})$, then a_i must either appear in $atomic(\mathcal{C}_0)$, be introduced by Transformation 12, or else be an intersection variable. Hence, it is clear that in case (a), the new atomic set expression satisfies part (iii) of the atomic set expression invariant. Now consider case (b). In this case, it is easy to verify that the expression $\overline{se_i}$ satisfies the atomic set expression invariant because $\overline{S}$ must satisfy part (iv) of the invariant.

Transformation 12: By definition, any new atomic set expressions introduced by this transformation satisfy part (ii) of the atomic expression invariant. $\square$

We next establish some basic properties of the transformations. Intuitively, each transformation picks a constraint $\mathcal{X} \supseteq se$ from the current collection $\mathcal{C}$ of constraints and endeavors to make the information contained in the constraint explicit by adding new constraints $\mathcal{X} \supseteq se_1$, ..., $\mathcal{X} \supseteq se_n$ that are "closer" to explicit form (strictly speaking, the transformations dealing with intersection output some additional constraints for new intersection variables). Now, the following sequence of propositions consider each transformation in turn and essentially relate the expression se in the constraint $\mathcal{X} \supseteq se$ picked from $\mathcal{C}$ with the expressions $se_1, \ldots, se_n$ in the constraints constructed by the transformation. These relationships shall be used to prove that the transformations are sound (in the sense that they preserve $lm(\mathcal{C})$) and complete (in the sense that, on termination of the exhaustive application of the transformations, all information about the least model is contained in the explicit form constraints). Note that each proposition corresponds to a transformations in Δ_2; each proposition employs the notation used in the specification of the corresponding transformation.

Proposition 34 (Transformation 5) *Let the application of* REDUCE *to* $\mathcal{X} \supseteq \{X : (s \in a) \wedge conj\}$ *result in the constraints* $\mathcal{X} \supseteq se_1, \ldots, \mathcal{X} \supseteq se_n$, *and let* $\mathcal{I}$ *be an interpretation.*

(a) *If* $\mathcal{I} \models \mathcal{Y} \supseteq a$ *and* $\{X : (s \in \mathcal{Y}) \wedge conj\}$ *is safe with respect to* $\mathcal{I}$ *then* $\mathcal{I}(\{X : (s \in \mathcal{Y}) \wedge conj\}) \supseteq \mathcal{I}(se_i)$, $i = 1..n$.

(b) $\mathcal{I}(\{X : s \in a \wedge conj\}) \subseteq \mathcal{I}(se_1 \cup \cdots \cup se_n)$.

Proof: Consider part (a) and suppose that $\mathcal{I}$ is a model of $\mathcal{Y} \supseteq a$ and $\{X : (s \in \mathcal{Y}) \wedge conj\}$ is safe with respect to $\mathcal{I}$. This implies that $\mathcal{I}(\mathcal{Y}) \supseteq \mathcal{I}(a)$ and so:

$$\mathcal{I}(s \in a) \;\subseteq\; \mathcal{I}(s \in \mathcal{Y}) \tag{7.29}$$

Moreover, $\rho \rhd (s \in \mathcal{Y})$ iff $\rho \rhd (s \in \mathcal{Y})$. Combining this with the assumption that $\{X : s \in \mathcal{Y} \wedge conj\}$ is safe with respect to $\mathcal{I}$ establishes the three preconditions of Proposition 30. Hence $\{X : s \in a \wedge conj\}$ is safe with respect to $\mathcal{I}$. It follows from Lemma 14 that

$$\mathcal{I}(\{X : s \in a \wedge conj\}) \supseteq \mathcal{I}(se_i),\ i = 1..n.$$

Now, (7.29) implies that $\mathcal{I}(\{X : (s \in \mathcal{Y}) \wedge conj\}) \supseteq \mathcal{I}(\{X : (s \in a) \wedge conj\})$. Hence, $\mathcal{I}(\{X : (s \in \mathcal{Y}) \wedge conj\}) \supseteq \mathcal{I}(se_i)$, $i = 1..n$, and this proves (a).

Finally, part (b) follows immediately from Lemma 14, and this completes the proof of the proposition. □

Proposition 35 (Transformation 6) *If* $\mathcal{I} \models \mathcal{Y} \supseteq a$ *then* $\mathcal{I}(a_1 \cap \cdots \cap a_{i-1} \cap \mathcal{Y} \cap a_{i+1} \cap \cdots \cap a_n) \supseteq \mathcal{I}(a_1 \cap \cdots \cap a_{i-1} \cap a \cap a_{i+1} \cap \cdots \cap a_n)$. □

Proposition 36 (Transformation 7) *If* $\mathcal{I} \models \mathcal{Y} \supseteq a$ *then* $\mathcal{I}(\mathcal{Y}) \supseteq \mathcal{I}(a)$. □

Proposition 37 (Transformation 8) *For all interpretations* $\mathcal{I}$,

$$\mathcal{I}(f_{(i)}^{-1}(a)) \;=\; \mathcal{I}(\{X_i : f(X_1, \ldots, X_n) \in a\}).$$

Proof: The proof follows easily from the following chain of equalities:

$$\begin{aligned}
\mathcal{I}(f_{(i)}^{-1}(a)) &= \{v_i : f(v_1, \ldots, v_n) \in \mathcal{I}(a)\} \\
&= \{\rho(X_i) : \rho(f(X_1, \ldots, X_n)) \in \mathcal{I}(a)\} \\
&= \mathcal{I}(\{X_i : f(X_1, \ldots, X_n) \in a\}). \quad \square
\end{aligned}$$

Proposition 38 (Transformation 9) *If $\overline{S_1}, \ldots, \overline{S_n}$ and $a'_1, \ldots, a'_{m-n}$ are two subsequences of $a_1 \cap \cdots \cap a_m$ such that each a_i is in one of the two subsequences, then, for all interpretations $\mathcal{I}$,*

$$\mathcal{I}(a_1 \cap \ldots \cap a_m) \;=\; \mathcal{I}(a'_1 \cap \ldots \cap a'_{m-n} \cap \overline{S}) \quad \textit{where } S = S_1 \cup \cdots \cup S_n.$$

Proof: The proof follows easily from the following chain of equalities:

$$\begin{aligned}
\mathcal{I}(a_1 \cap \ldots \cap a_m) &= \left\{ v : \begin{array}{l} \bigwedge_{i=1..n} (v \notin \mathcal{I}(se) \text{ for each } se \in S_i) \\ v \in \mathcal{I}(a'_1 \cap \ldots \cap a'_{m-n}) \end{array} \right\} \\
&= \left\{ v : \begin{array}{l} v \notin \mathcal{I}(se) \text{ for each } se \in S_1 \cup \cdots \cup S_n \\ v \in \mathcal{I}(a'_1 \cap \ldots \cap a'_{m-n}) \end{array} \right\} \\
&= \left\{ v : \begin{array}{l} v \notin \mathcal{I}(se) \text{ for each } se \in S \\ v \in \mathcal{I}(a'_1 \cap \ldots \cap a'_{m-n}) \end{array} \right\} \\
&= \mathcal{I}(a'_1 \cap \ldots \cap a'_{m-n} \cap \overline{S}). \quad \Box
\end{aligned}$$

Proposition 39 (Transformation 10) *If $\mathcal{I}$ is an interpretation such that for each j, $\mathcal{I}\left(\mathcal{V}_{N_j}\right) = \mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j})$, then*

$$\mathcal{I}\left(a_1 \cap \cdots \cap a_m \cap \overline{S}\right) \;=\; \mathcal{I}\left(f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n}) \cap \overline{S}\right)$$

Proof: The proof is straightforward:

$$\begin{aligned}
&\mathcal{I}\left(a_1 \cap \cdots \cap a_m \cap \overline{S}\right) \\
&= \mathcal{I}\left(f(a_{1,1}, \ldots, a_{1,n}) \cap \cdots \cap f(a_{m,1}, \ldots, a_{m,n}) \cap \overline{S}\right) \\
&= \mathcal{I}\left(f(a_{1,1} \cap \cdots \cap a_{m,1}, \ldots, a_{1,n} \cap \cdots \cap a_{m,n}) \cap \overline{S}\right) \\
&= \mathcal{I}\left(f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n}) \cap \overline{S}\right)
\end{aligned}$$

Proposition 40(a) (Transformation 11) *If S contains no elements of the form $f(\cdots)$ then, for all interpretations $\mathcal{I}$,*

$$\mathcal{I}(f(a_1, \ldots, a_n) \cap \overline{S}) \;=\; \mathcal{I}\left(f(a_1, \ldots, a_n)\right).$$

Proof: The proof follows from the following chain of equalities:

$$
\begin{aligned}
&\mathcal{I}\left(f(a_1,\ldots,a_n)\cap\overline{S}\right)\\
&=\{v : v\in\mathcal{I}(f(a_1,\ldots,a_n)) \text{ and } v\notin\mathcal{I}(se) \text{ for all } se\in S\}\\
&=\left\{f(v_1,\ldots,v_n) : \begin{array}{l} f(v_1,\ldots,v_n)\in\mathcal{I}(f(a_1,\ldots,a_n)) \text{ and}\\ f(v_1,\ldots,v_n)\notin\mathcal{I}(se) \text{ for all } se\in S\end{array}\right\}\\
&=\left\{f(v_1,\ldots,v_n) :\ f(v_1,\ldots,v_n)\in\mathcal{I}(f(a_1,\ldots,a_n))\ \right\}\\
&=\mathcal{I}\left(f(a_1,\ldots,a_n)\right).
\end{aligned}
$$

The first equality follows from the definition of $\mathcal{I}$. The second step follows from the fact that any element of $\mathcal{I}(f(a_1,\ldots,a_n))$ must have the form $f(v_1,\ldots,v_n)$. Now consider the third equality. Recall that a constant of the form $\overline{S}$ is such that S is a set of atomic set expressions of the form $g(\cdots)$. Now, by assumption, S does not contain any elements of the form $f(\cdots)$. Hence, S contains only expressions of the form $g(\cdots)$ where $g\neq f$. It follows that if $se\in S$ then $\mathcal{I}(se)$ cannot contain any elements of the form $f(v_1,\ldots,v_n)$, and so the condition $f(v_1,\ldots,v_n)\notin\mathcal{I}(se)$ for all $se\in S$ is vacuously true. The last equality again follows from the definition of $\mathcal{I}$. ∎

Proposition 40(b) (Transformation 11) *If $\mathcal{I}$ is an interpretation such that for each j, $\mathcal{I}(\mathcal{V}_{N_j})=\mathcal{I}(a_j\cap\overline{se_j})$, then*

$$
\mathcal{I}(f(a_1,\ldots,a_n)\cap\overline{S}) \;=\; \mathcal{I}\left(\bigcup_{i=1..n} f(a_1,\ldots,a_{i-1},\mathcal{V}_{N_j},a_{i+1},\ldots,a_n)\cap\overline{S'}\right)
$$

Proof: The proof follows from the following chain of equalities:

$$
\begin{aligned}
&\mathcal{I}\left(f(a_1,\ldots,a_n)\cap\overline{S}\right)\\
&=\mathcal{I}\left(f(a_1,\ldots,a_n)\cap\overline{f(se_1,\ldots,se_n)}\cap\overline{S'}\right)\\
&=\{v : v\in\mathcal{I}(f(a_1,\ldots,a_n)) \text{ and } v\notin\mathcal{I}(f(se_1,\ldots,se_n))\}\cap\mathcal{I}(\overline{S'})\\
&=\left\{f(v_1,\ldots,v_n) : \begin{array}{l} f(v_1,\ldots,v_n)\in\mathcal{I}(f(a_1,\ldots,a_n))\\ f(v_1,\ldots,v_n)\notin\mathcal{I}(f(se_1,\ldots,se_n))\end{array}\right\}\cap\mathcal{I}(\overline{S'})\\
&=\left\{f(v_1,\ldots,v_n) : \begin{array}{l} v_i\in\mathcal{I}(a_i) \text{ for } i=1..n, \text{ and}\\ v_j\notin\mathcal{I}(se_j) \text{ for some } j,\ 1\le j\le n\end{array}\right\}\cap\mathcal{I}(\overline{S'})\\
&=\left\{f(v_1,\ldots,v_n) : \begin{array}{l} v_i\in\mathcal{I}(a_i) \text{ for } i=1..n, \text{ and}\\ v_j\in\mathcal{I}(\overline{se_j}) \text{ for some } j,\ 1\le j\le n\end{array}\right\}\cap\mathcal{I}(\overline{S'})\\
&=\bigcup_{j=1..n}\left\{f(v_1,\ldots,v_n) : \begin{array}{l} v_i\in\mathcal{I}(a_i) \text{ for } i\neq j, \text{ and}\\ v_j\in\mathcal{I}(a_j\cap\overline{se_j})\end{array}\right\}\cap\mathcal{I}(\overline{S'})\\
&=\bigcup_{j=1..n}\left\{f(v_1,\ldots,v_n) : \begin{array}{l} v_i\in\mathcal{I}(a_i) \text{ for } i\neq j, \text{ and}\\ v_j\in\mathcal{I}(\mathcal{V}_{N_j})\end{array}\right\}\cap\mathcal{I}(\overline{S'})\\
&=\bigcup_{j=1..n}\mathcal{I}\left(f(a_1,\ldots,a_{j-1},\mathcal{V}_{N_j},a_{j+1},\ldots,a_n)\right)\cap\mathcal{I}(\overline{S'})
\end{aligned}
$$

□

Proposition 41 (Transformation 12) *Let the application of* REDUCE *to* $\mathcal{X}\supseteq\{X : s\in\overline{v}\wedge conj\}$ *result in the constraints* $\mathcal{X}\supseteq se_1,\ldots,\mathcal{X}\supseteq se_n$, *and let* $\mathcal{I}$ *be an interpretation.*

(a) If $v\in\mathcal{I}(\mathcal{Y})$ *and* $\{X : s\dagger a\wedge conj\}$ *is safe with respect to* $\mathcal{I}$ *then* $\mathcal{I}(\{X : s\dagger a\wedge conj\})\supseteq\mathcal{I}(se_i)$, $i=1..n$.

(b) $\mathcal{I}\left(\{X : s\in\overline{v}\wedge conj\right)\subseteq\mathcal{I}(se_1\cup\cdots\cup se_n)$.

Proof: First consider (a). Suppose that $v\in\mathcal{I}(\mathcal{Y})$ and $\{X : s\dagger a\wedge conj\}$ is safe with respect to $\mathcal{I}$ and consider the following chain of implications:

$$\begin{array}{lll} \rho \in \mathcal{I}(s \in \overline{v}) & \text{implies} & \rho(s) \notin \mathcal{I}(v) \\ & \text{implies} & \rho(s) \notin \{v\} \\ & \text{implies} & v \neq \rho(s) \\ & \text{implies} & \exists v'(v' \neq \rho(s) \wedge v' \in \mathcal{I}(a) \\ & \text{implies} & \rho \in \mathcal{I}(s \dagger a). \end{array}$$

This proves that $\mathcal{I}(s \in \overline{v}) \subseteq \mathcal{I}(s \dagger a)$. Moreover, $\rho \rhd (s \in \overline{v})$ iff $\rho \rhd (s \dagger a)$. Combining this with the assumption that $\{X : s \dagger a \wedge conj\}$ is safe with respect to $\mathcal{I}$ establishes the preconditions of Proposition 30. This implies that $\{X : s \in \overline{v} \wedge conj\}$ is safe with respect to $\mathcal{I}$. Hence, Lemma 14 proves that $\mathcal{I}(\{X : s \dagger a \wedge conj\}) \supseteq \mathcal{I}(se_i)$, $i = 1..n$. Now, $\mathcal{I}(s \in \overline{v}) \subseteq \mathcal{I}(s \dagger a)$ also implies that $\mathcal{I}(\{X : s \dagger a \wedge conj\}) \supseteq \mathcal{I}(\{X : s \in \overline{v} \wedge conj\})$. It follows that $\mathcal{I}(\{X : s \dagger a \wedge conj\}) \supseteq \mathcal{I}(se_i)$, $i = 1..n$, and this proves (a).

Finally, part (b) follows immediately from Lemma 14, and this completes the proof of the proposition. $\Box$

Proposition 42 (Transformation 13) *Let $\mathcal{I}$ be an interpretation. If $\{X : s \dagger a \wedge conj\}$ is safe with respect to $\mathcal{I}$ and $\mathcal{I}(a)$ contains more than one element then $\mathcal{I}(\{X : s \dagger a \wedge conj\}) \;=\; \mathcal{I}(\{X : conj\})$*

Proof: If $\rho \rhd s$ then $\rho \in \mathcal{I}(s \dagger a)$ iff $\exists v'(v' \neq \rho(s) \wedge v' \in \mathcal{I}(a))$, but since $\mathcal{I}(a)$ contains more than one element, v' can always be chosen to be different from $\rho(s)$. Hence $\rho \in \mathcal{I}(s \dagger a)$ is true just in case $\rho \rhd s$. Therefore, to prove the proposition it suffices to show that

$$\rho \in \mathcal{I}(conj) \text{ implies } \rho \rhd s$$

and this implication can be proved as follows. Suppose suppose that $\rho \rhd s$ does not hold. This implies that $\rho \notin \mathcal{I}(s \dagger a \wedge conj)$. Since $\{X : s \dagger a \wedge conj\}$ is safe with respect to $\mathcal{I}$, there exists a quantified condition exp_ρ in $s \dagger a \wedge conj$ such that $\rho \rhd s$ and $\rho \notin \mathcal{I}(exp_\rho)$. Clearly exp_ρ cannot be $s \dagger a$ because it has just been shown that $\rho \in \mathcal{I}(s \dagger a)$ iff $\rho \rhd s$. Hence exp_ρ must appear in $conj$ and it follows that $\rho \notin \mathcal{I}(conj)$. $\Box$

Proposition 43 (Transformation 14) *Let $\mathcal{I}$ be an interpretation and let conj be a conjunction of quantified conditions in compaction form. If $\mathcal{I}(\bigcap_{Y} conj)$ is non-empty for each program variable Y appearing in conj, then $\mathcal{I}(\bigcap_{X} conj) = \mathcal{I}(\{X : conj\})$.*

Proof: Let $X_1, \ldots, X_n$ be a listing of the program variables appearing in $\{X : \mathit{conj}\}$ and let X be X_k. Since *conj* is in compaction form, each of its quantified condition is of the form $X_i \in a$ where a is atomic. For each i, let $X_i \in a_{i,1}, \ldots, X_i \in a_{i,n_i}$ be the quantified conditions in *conj* of the form $X_i \in a$. Now consider the following chain of containments and equalities:

$$
\begin{aligned}
\mathcal{I}\Big(\{X : \mathit{conj}\}\Big) &= \{\rho(X_k) : \rho \in \mathcal{I}(\mathit{conj})\} \\
&= \{\rho(X_k) : \forall i\, \forall j\, (\rho(X_i) \in \mathcal{I}(a_{i,j}))\} \\
&= \{\rho(X_k) : \forall i\, (\rho(X_i) \in \mathcal{I}(a_{i,1} \cap \cdots \cap a_{i,n_i}))\} \\
&= \{\rho(X_k) : \forall i\, (\rho(X_i) \in \mathcal{I}(\mathit{conj}_{X_i}))\} \\
&= \Big\{\rho(X_k) : \rho(X_k) \in \mathcal{I}(\mathit{conj}_{X_k})\Big\} \\
&= \mathcal{I}\,(\textstyle\bigcap_{X} \mathit{conj})\,.
\end{aligned}
$$

The fifth equality (which removes the universal quantifier) follows from the fact that each $a_{i,1} \cap \cdots \cap a_{i,n_i}$ is non-empty in $lm(\mathit{explicit}(\mathcal{C}))$ and since $lm(\mathit{explicit}(\mathcal{C})) \subseteq \mathcal{I}$, this implies that $\mathcal{I}(a_{i,1} \cap \cdots \cap a_{i,n_i}) \neq \{\}$. $\square$

We now use these basic propositions to prove that the transformation are sound (that is, they preserve the least model). Note that a number of the propositions have side conditions relating to intersection variables and safeness. We therefore prove soundness in conjunction with two invariants.

Invariant 4 *[Intersection Variable Invariant] $\mathcal{C}$ satisfies the intersection variable invariant if $\mathcal{V}_N \in var(\mathcal{C})$ implies that $lm(\mathcal{C}) \models \mathcal{V}_N = (\bigcap N)$.* $\square$

Invariant 5 (Safeness Invariant) *$\mathcal{C}$ satisfies the safeness invariant if $\mathcal{C}$ is safe with respect to $lm(\mathcal{C})$.* $\square$

The key part of the proof that Δ_2 is sound is the following proposition, which says that the transformations are sound if both invariants are satisfied, and moreover, that the transformations preserve the invariants.

Proposition 44 *Let $\mathcal{C}$ be a collection of constraints that satisfies the intersection variable invariant and the safeness invariant. Then, for each transformation δ in Δ_2,*

1. *$lm(\mathcal{C} \cup \delta(\mathcal{C})) =_{var(\mathcal{C})} lm(\mathcal{C})$;*
2. *$\mathcal{C} \cup \delta(\mathcal{C})$ satisfies the intersection variable invariant, and*
3. *$\mathcal{C} \cup \delta(\mathcal{C})$ satisfies the safeness invariant.*

Proof (No New Variables): This case covers all transformation applications except those that may introduce new variables. Specifically, it excludes Transformation 10 and case (b) of Transformation 11. First consider part (1). Since δ does not introduce new variables, part (1) reduces to $lm(\mathcal{C} \cup \delta(\mathcal{C})) = lm(\mathcal{C})$. It is easy to verify that $lm(\mathcal{C} \cup \delta(\mathcal{C})) \supseteq lm(\mathcal{C})$ (Proposition 50 in Appendix I contains a proof of this in a very general setting). Hence it suffices to show that $lm(\mathcal{C})$ is a model of $\delta(\mathcal{C})$. Let $\mathcal{I}$ denote $lm(\mathcal{C})$, and since $\mathcal{C}$ satisfies the safeness invariant, this implies that each quantified set expression in $\mathcal{C}$ is safe with respect to $\mathcal{I}$. It follows that the preconditions of Propositions 34–38, 40(a) and 41–43 (corresponding to Transformations 5–9, case (a) of Transformation 11, and Transformations 12–14 respectively) are satisfied. These propositions imply that $\delta(\mathcal{C})$ consists of a collection of constraints $\mathcal{X} \supseteq se_1, \ldots, \mathcal{X} \supseteq se_n$, $n \geq 1$, such that there is a constraint of the form $\mathcal{X} \supseteq se$ in $\mathcal{C}$ and $\mathcal{I}(se) \supseteq \mathcal{I}(se_i)$, $i = 1..n$. Now, since $\mathcal{X} \supseteq se$ appears in $\mathcal{C}$, it must be the case that $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$. Hence $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se_i)$, $i = 1..n$, and this proves the $\mathcal{I}$ is a model of $\delta(\mathcal{C})$, and completes the proof of (1).

Consider part (2). Since all intersection variables in $\mathcal{C} \cup \delta(\mathcal{C}))$ appear in $\mathcal{C}$, part (2) immediately follows from part (1) and the fact that $\mathcal{C}$ satisfies the intersection variable invariant.

Consider part (3) of the proposition. By assumption, each quantified set expression in $\mathcal{C}$ is safe with respect to $lm(\mathcal{C})$. Since $lm(\mathcal{C}) = lm(\mathcal{C} \cup \delta(\mathcal{C}))$, it follows that these same quantified set expressions are also safe with respect to $lm(\mathcal{C} \cup \delta(\mathcal{C}))$. It remains to consider the new quantified set expression introduced by $\delta(\mathcal{C})$. The only transformations that can introduced new quantified set expressions are 5, 8, 12 and 13. Let $\mathcal{I}$ denote $lm(\mathcal{C} \cup \delta(\mathcal{C}))$ and consider each of these transformations in turn.

In the case of Transformation 5, $\{X : (s \in \mathcal{Y}) \wedge conj\}$ is safe with respect to $\mathcal{I}$ because $\mathcal{C}$ is assumed to be safe with respect to $\mathcal{I}$. Moreover, since $\mathcal{I}$ is a model of $\mathcal{C}$, $\mathcal{I}(\mathcal{Y}) \supseteq \mathcal{I}(a)$, and so $\mathcal{I}(s \in a) \subseteq \mathcal{I}(s \in \mathcal{Y})$. Combining this with the fact that $\rho \rhd (s \in a)$ iff $\rho \rhd (s \in \mathcal{Y})$, establishes the three preconditions of

Proposition 30. Hence $\{X : s \in a \wedge conj\}$ is safe with respect to $\mathcal{I}$. Finally, by Lemma 14, it follows that $\delta(\mathcal{C}) = \text{REDUCE}(\mathcal{X} \supseteq \{X : (s \in a) \wedge conj\})$ is safe with respect to $\mathcal{I}$.

In the case of Transformation 8, it is clear that the quantified set expression $\{X_i : f(X_1, \ldots, X_n) \in a\}$ is safe with respect to $\mathcal{I}$ because $\rho \rhd f(X_1, \ldots, X_n)$ for all environments ρ. Hence, $\delta(\mathcal{C}) = \text{REDUCE}(\mathcal{X} \supseteq \{X_i : f(X_1, \ldots, X_n) \in a\})$ is safe with respect to $\mathcal{I}$, by Lemma 14.

In the case of Transformation 12, $\{X : (s \dagger a) \wedge conj\}$ is safe with respect to $\mathcal{I}$ because $\mathcal{C}$ is assumed to be safe with respect to $\mathcal{I}$. Also, since $\mathcal{I} \supseteq lm(explicit(\mathcal{C}))$ it follows that $\mathcal{I}(a) \supseteq \mathcal{I}(v)$. Using this, it is easy to verify that $\mathcal{I}(s \in \overline{v}) \subseteq \mathcal{I}(s \dagger a)$. Combining this with the fact that $\rho \rhd (s \in \overline{v})$ iff $\rho \rhd (s \dagger a)$, establishes the three preconditions of Proposition 30. Hence $\{X : s \in \overline{v} \wedge conj\}$ is safe with respect to $\mathcal{I}$. Finally, by Lemma 14, it follows that $\delta(\mathcal{C})$ is safe with respect to $\mathcal{I}$.

Consider Transformation 13. The only new quantified set expression in $\delta(\mathcal{C})$ is $\{X : conj\}$. Since $\mathcal{I} \supseteq lm(explicit(\mathcal{C}))$, it follows that $\mathcal{I}(a)$ must contain at least two elements. If $\rho \rhd s$, then by definition, $\rho \in \mathcal{I}(s \dagger a)$ iff $\exists v'(v' \neq \rho(s) \wedge v' \in \mathcal{I}(a))$, but since $\mathcal{I}(a)$ contains more than one element, v' can always be chosen to be different from $\rho(s)$. Hence, if $\rho \rhd s$ then $\rho \in \mathcal{I}(s \dagger a)$. Now, suppose that $\rho \notin \mathcal{I}(conj)$. Then $\rho \notin \mathcal{I}(s \dagger a \wedge conj)$. Since $\mathcal{C}$ is assumed to be safe with respect to $\mathcal{I}$, it follows that there exists a quantified set expression exp_ρ in $s \dagger a \;\wedge\; conj$ such that $\rho \rhd exp_\rho$ but $\rho \notin \mathcal{I}(exp_\rho)$. If exp_ρ is $s \dagger a$, then $\rho \rhd s$ and $\rho \notin \mathcal{I}(s \dagger a)$, but we have just proved that this is not possible. Hence exp_ρ must appear in $conj$ and this completes the proof that $\delta(\mathcal{C})$ is safe. $\square$

Proof (Intersection Variables): This case covers transformation applications that may introduce new variables, that is, Transformation 10 and case (b) of Transformation 11. The proof for these transformations builds on the a similar proof in Proposition 25 (page 187). Some of the following material is a duplication of material from Proposition 25, however it is repeated because this proof is a key part of the correctness argument.

Consider the intersection variables $\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n}$ mentioned by Transformations 10 and 11. For each $\mathcal{V}_{N_j}$, $j = 1..n$, either $\mathcal{V}_{N_j}$ appears in $var(\mathcal{C})$ or else a constraint $\mathcal{V}_{N_j} \supseteq a_{1,j} \cap \cdots \cap a_{m,j}$ is included in $\delta(\mathcal{C})$ such that $N_j = \bigcup_{i=1..m} \mathcal{N}(a_{i,j})$. Note that each $a_{i,j}$ appears in $\mathcal{C}$. Using this fact, an interpretation $\mathcal{I}$ can be defined as follows:

$$\mathcal{I}(\mathcal{X}) = \begin{cases} lm(\mathcal{C})(a_{1,j} \cap \cdots \cap a_{m,j}) & \text{if } \mathcal{X} \notin var(\mathcal{C}) \text{ and } \mathcal{X} \text{ is } \mathcal{V}_{N_j} \\ lm(\mathcal{C})(\mathcal{X}) & \text{otherwise} \end{cases}$$

That is, $\mathcal{I}$ extends $lm(\mathcal{C})$ to the new intersection variables such that $\mathcal{I}$ satisfies each constraint $\mathcal{V}_{N_j} \supseteq a_{1,j} \cap \cdots \cap a_{m,j}$ that appears in $\delta(\mathcal{C})$. The main part of the proof for this case is that $\mathcal{I} = lm(\mathcal{C} \cup \delta(\mathcal{C}))$.

Consider each of the set expressions $a_{i,j}$. As has already been noted, each $a_{i,j}$ appears in $\mathcal{C}$, and this can be used to prove that each $a_{i,j}$ satisfies the following equation:

$$\mathcal{I}(a_{i,j}) = \mathcal{I}(\bigcap \mathcal{N}(a_{i,j})) \tag{7.30}$$

To prove this, observe that if $a_{i,j}$ is an intersection variable, then (7.30) follows from the intersection variable invariant for $\mathcal{C}$, and if $a_{i,j}$ is not an intersection variable then $\bigcap \mathcal{N}(a_{i,j})$ is just $a_{i,j}$. Equation (7.30) can be used to prove the following chain of equalities

$$\begin{aligned} \mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j}) &= \mathcal{I}(a_{1,j}) \cap \cdots \cap \mathcal{I}(a_{m,j}) \\ &= \mathcal{I}\left(\bigcap \mathcal{N}(a_{1,j})\right) \cap \cdots \cap \mathcal{I}\left(\bigcap \mathcal{N}(a_{m,j})\right) \\ &= \mathcal{I}\left(\bigcap \mathcal{N}(a_{1,j}) \cup \cdots \cup \mathcal{N}(a_{m,j})\right) \\ &= \mathcal{I}\left(\bigcap N_j\right) \end{aligned}$$

which proves that, for all j, $\mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j}) = \mathcal{I}(\cap N_j)$. Now, if $\mathcal{V}_{N_j}$ is introduced by δ, then $\mathcal{I}(\mathcal{V}_{N_j}) = \mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j})$ by definition of $\mathcal{I}$. On the other hand, if $\mathcal{V}_{N_j}$ appears in $\mathcal{C}$ then $\mathcal{I}(\mathcal{V}_{N_j}) = \mathcal{I}(\bigcap N_j)$ because $\mathcal{C}$ satisfies the intersection variable invariant. Hence, for $j = 1..n$,

$$\mathcal{I}(\mathcal{V}_{N_j}) \ = \ \mathcal{I}(\bigcap N_j) \ = \ \mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j}) \tag{7.31}$$

The main use of (7.31) is to prove that $\mathcal{I}$ is a model of $\delta(\mathcal{C})$. Recall the definitions of Transformations 10 and 11. In both cases, there is a constraint $\mathcal{X} \supseteq se$ in $\mathcal{C}$ such that the constraints in $\delta(\mathcal{C})$ are either of the form (a) $\mathcal{X} \supseteq se'$ or (b) $\mathcal{V}_{N_j} \supseteq a_{1,j} \cap \cdots \cap a_{m,j}$ where $\mathcal{V}_{N_j}$ is a new intersection variable. By definition, $\mathcal{I}$ is a model of the constraints in (b). Consider a constraint $\mathcal{X} \supseteq se'$ in (a), and recall Propositions 39, 40(a) and 40(b). It is clear from (7.31) that $\mathcal{I}$ satisfies the preconditions of these transformations. It follows that $\mathcal{I}(se) \supseteq \mathcal{I}(se')$. Moreover, $\mathcal{I}$ is a model of $\mathcal{C}$, and so $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se) \supseteq \mathcal{I}(se')$. Hence $\mathcal{I}$ is a model of $\mathcal{X} \supseteq se$. This completes the proof that $\mathcal{I}$ is a model of $\delta(\mathcal{C})$.

$\mathcal{I}$ is not only a model of $\mathcal{C} \cup \delta(\mathcal{C})$, it is in fact the least model. To see this, let $\mathcal{I}'$ be an arbitrary model of $\mathcal{C} \cup \delta(\mathcal{C})$. If $\mathcal{X}$ is a variable that is different from the $\mathcal{V}_{N_j}$ introduced at this step, then $\mathcal{I}'(\mathcal{X}) \supseteq \mathcal{I}(\mathcal{X})$ because $\mathcal{I}' \supseteq lm(\mathcal{C})$ and $lm(\mathcal{C}) = \mathcal{I}(X)$ by definition of $\mathcal{I}$. On the other hand, for the variables $\mathcal{V}_{N_j}$ introduced at this step, consider the following chain:

$$\mathcal{I}'(\mathcal{V}_{N_j}) \quad \supseteq \quad \mathcal{I}'(a_{1,j} \cap \cdots \cap a_{m,j}) \quad \supseteq \quad \mathcal{I}(a_{1,j} \cap \cdots \cap a_{m,j}) \quad = \quad \mathcal{I}(\mathcal{V}_{N_j}).$$

The first containment follows because $\mathcal{I}'$ is a model of $\delta(\mathcal{C})$. The second is because $a_{1,j} \cap \cdots \cap a_{m,j}$ contains only variables from $\mathcal{C}$, and it has just been proved that $\mathcal{I}'(\mathcal{X}) \supseteq \mathcal{I}(\mathcal{X})$ for variables $\mathcal{X} \in var(\mathcal{C})$. The final equality follows from the definition of $\mathcal{I}$. This completes the proof that $lm(\mathcal{C} \cup \delta(\mathcal{C})) \supseteq \mathcal{I}$. Combining this with $\mathcal{I} \supseteq lm(\mathcal{C} \cup \delta(\mathcal{C}))$ proves that $\mathcal{I} = lm(\mathcal{C} \cup \delta(\mathcal{C}))$.

Now, by definition, $\mathcal{I}$ agrees with $lm(\mathcal{C})$ on $var(\mathcal{C})$, and so $lm(\mathcal{C}) =_{var(\mathcal{C})} lm(\mathcal{C} \cup \delta(\mathcal{C}))$. This proves part (1) of the proposition. To prove part (2), we need to show that $\mathcal{I}(\mathcal{V}_N) = \mathcal{I}(\bigcap N)$ for all intersection variables $\mathcal{V}_N$ appearing in $\mathcal{C}$. Suppose that $\mathcal{V}_N \in var(\mathcal{C})$. Since $\mathcal{C}$ satisfies the intersection invariant and $\mathcal{I}$ agrees with $lm(\mathcal{C})$ on $var(\mathcal{C})$,

$$\mathcal{I}(\mathcal{V}_N) = lm(\mathcal{C})(\mathcal{V}_N) = lm(\mathcal{C})(\bigcap N) = \mathcal{I}(\bigcap N).$$

On the other hand, if $\mathcal{V}_N$ is introduced by $\delta(\mathcal{C})$, then $\mathcal{I}(\mathcal{V}_N) = \mathcal{I}(\bigcap N)$ follows from (7.31).

It remains to show that $\mathcal{C} \cup \delta(\mathcal{C})$ satisfies the safeness invariant. Now, none of the transformations considered introduce new quantified set expressions. Hence each quantified set expression in $\mathcal{C} \cup \delta(\mathcal{C})$ is safe with respect to $lm(\mathcal{C})$. Moreover, these quantified set expression only involve variables from $var(\mathcal{C})$. Since $lm(\mathcal{C})$ and $lm(\mathcal{C} \cup \delta(\mathcal{C})$ agree on $var(\mathcal{C})$, it is easy to verify that $\mathcal{C} \cup \delta(\mathcal{C})$ is safe with respect to $lm(\mathcal{C} \cup \delta(\mathcal{C}))$. Hence $\mathcal{C} \cup \delta(\mathcal{C})$ satisfies the safeness invariant. $\square$

We have already been shown that $\mathcal{C}_0 = \text{REDUCE}(\text{STANDARDIZE}(\mathcal{SC}_P))$ satisfies the safeness invariant. Also, it is clear that $\mathcal{C}_0$ satisfies the intersection variable invariant (since it does not contain any intersection variables). Hence, the previous proposition directly implies that:

Lemma 16 (Soundness) Δ_2 *is sound on each* $\mathcal{C}_i$ *constructed by the algorithm.*

Proof: Clearly $\mathcal{C}_0$ satisfies the safeness invariant and the intersection variable invariant. Repeated application of Proposition 44 proves that each $\mathcal{C}_i$ constructed by the algorithm satisfies both of these invariants. It also proves that each $\mathcal{C}_i$ is sound on $\mathcal{C}_i$. □

We now prove that the algorithm terminates. We begin by showing that Δ_2 is atomically bounded.

Lemma 17 (Atomically Bounded) Δ_2 *is atomically bounded.*

Proof: The proof proceeds by establishing a bound on the atomic set expressions that can be introduced by the algorithm. It has already been established in Proposition 33 that each $\mathcal{C}_i$ constructed by the algorithm satisfies the atomic set expression invariant. This implies that each atomic set expression introduced by the algorithm either:

(i) appears in $atomic(\mathcal{C}_0)$;

(ii) is introduced by an application of Transformation 12;

(iii) is of the form $f(a_1, \ldots, a_n)$ where f is a function symbol appearing in $\mathcal{C}_0$, and each a_i is either an intersection variable or a strict subterm of some atomic set expression that falls into cases (i) or (ii); or

(iv) is of the form $\overline{S}$ such that $S \subseteq atomic(S_1 \cup \cdots \cup S_n)$ for some complement constants $\overline{S_1}, \ldots, \overline{S_k}$ satisfying cases (i) or (ii) of the invariant.

Now, the set of atomic set expressions that satisfy (i) is fixed. Moreover, the atomic set expressions that may satisfy (iii) and (iv) are essentially determined by those that satisfy (i) and (ii). The critical item is therefore (ii), since Transformation 12 can introduce completely new atomic set expressions. When this transformation is applied to reduced form constraints $\mathcal{C}$, the effect is to add constraints $\text{REDUCE}(\mathcal{X} \supseteq \{X : s \in \overline{v} \wedge conj\})$ such that $\mathcal{X} \supseteq \{X : s \dagger a \wedge conj\}$ is a constraint in $\mathcal{C}$ and $lm(explicit(\mathcal{C}))(a) = \{v\}$.

Recall the steps of REDUCE (see Figure 7.5, page 199). Clearly these steps do not affect the quantified conditions in *conj* since *conj* is in reduced form. Hence, the application of REDUCE to $\mathcal{X} \supseteq \{X : s \in \overline{v} \wedge conj\}$ can only involve applications of steps to the quantified condition $s \in \overline{v}$ and any new quantified conditions produced by such steps. In other words, all new quantified conditions in $\text{REDUCE}(\mathcal{X} \supseteq \{X : s \in \overline{v} \wedge conj\})$ can be traced back to $s \in \overline{v}$. Thus the new quantified conditions in $\text{REDUCE}(\mathcal{X} \supseteq \{X : s \in \overline{v} \wedge conj\})$ are the same as the new quantified conditions in $\text{REDUCE}(\mathcal{X} \supseteq \{X : s \in \overline{v}\})$.

Moreover, any new atomic set expressions introduced by this application of Transformation 12 must be contained in the new quantified conditions. It follows that all new atomic set expressions introduced by this application of Transformation 12 are contained in the set $ATM_{s,v}$ defined by:

$$ATM_{s,v} \;=\; \left\{ a \in atomic(a') : \begin{array}{c} s' \in a' \text{ is a constraint in} \\ \text{REDUCE}\,(\mathcal{X} \supseteq \{X : s \in \overline{v}\}) \end{array} \right\}$$

Importantly, if there is another constraint of the form $\mathcal{X} \supseteq \{X : s \dagger a \wedge conj'\}$ in $\mathcal{C}$, then the atomic set expressions introduced by an application of Transformation 12 to this constraint are also contained in $ATM_{s,v}$.

Now, consider the expressions $s \dagger a$. It is easy to verify that no transformation introduces new expressions of this form. Hence the only expressions of the form $s \dagger a$ that appear during the algorithm are those that appear in $\mathcal{C}_0$. This means that the new atomic set expressions introduced by Transformation 12 are dependent on a finite set quantified conditions $s \dagger a$, and the possible values v such that $lm(explicit(\mathcal{C})) = \{v\}$ for some collection of constraints $\mathcal{C}$ constructed during the algorithm. Now, consider the sequence of constraints $\mathcal{C}_0, \mathcal{C}_1, \ldots$ constructed by the algorithm. Since these are an increasing sequence of collections, it follows that $lm(explicit(\mathcal{C}_0)), lm(explicit(\mathcal{C}_1)), \ldots$ is an increasing sequence of interpretations. Hence, for any atomic set expression a, if $lm(explicit(\mathcal{C}_i))(a)$ and $lm(explicit(\mathcal{C}_j))(a)$ are both singleton sets, then it must be the case that $lm(explicit(\mathcal{C}_i))(a) = lm(explicit(\mathcal{C}_j))(a)$. This means that for each a, there is exactly one value of v such that $lm(explicit(\mathcal{C}_i))(a) = \{v\}$. Denote this value (if it exists) by v_a. A key consequence of this is that if an application of Transformation 12 occurs during the algorithm and the application involves the quantified condition $s \dagger a$, then any new atomic set expressions introduced by this application must be contained in $ATM(s, v_a)$ where v_a

is the value associated with a. In summary then, the atomic set expressions introduced by Transformation 12 must be contained in the finite set

$$\{a : a \in ATM(s, v_a) \text{ and } s \dagger a \text{ appears in } \mathcal{C}_0\}$$

Now, consider the union of this set with $atomic(\mathcal{C}_0)$. This combined set contains all of the atomic set expressions that can be introduced during algorithm execution that satisfy part (i) or (ii) of the atomic set expression invariant. Let K be the cardinality of this combined set. Now, the atomic set expressions that satisfy parts (iii) and (iv) of the atomic set expression invariant are essentially expressions that are derived from (i) and (ii), and these can be bounded using the bounds on (i) and (ii) as follows.

First consider the atomic set expressions that satisfy (iii). These are of the form $f(a_1, \ldots, a_n)$ where each a_i appears in (i) or (ii) or is an intersection variable. Now, intersection variables are of the form $\mathcal{V}_N$ such that N is some set of atomic set expressions. Moreover, by inspection of Transformations 10 and 11 (these are the only transformations that introduce intersection variables), N contains only atomic set expressions a_i such that $f(a_1, \ldots, a_n)$ is an atomic set expressions that appears at some stage during the algorithm. It follows that the number of intersection variables is bounded by 2^K. Hence the number of atomic set expressions that satisfy (iii) is bounded by $F.(K')^n$ where F is the number of function symbols appearing in $\mathcal{C}_0$, n is the maximum arity of a function symbol in $\mathcal{C}_0$, and K' is $K + 2^K$.

Finally, consider the atomic set expressions that satisfy (iv). These are of the form $\overline{S}$ such that there are constants $\overline{S_1}, \ldots, \overline{S_k}$ that fall into cases (i) or (ii), and $S \subseteq atomic(S_1 \cup \cdots \cup S_n)$. Now, a finite bound has already been established for the number of atomic set expressions introduced during the algorithm that satisfy parts (i) and (ii) of the atomic set expression invariant. Hence the set

$$atomic(\{a : a \in S \text{ and } \overline{S} \text{ satisfies (i) or (ii)}\})$$

is finite. Let it have cardinality K''. Then the number of atomic set expressions introduced by the algorithm that satisfy (iv) can be bounded by $2^{K''}$. This completes the proof that only a finite number of different atomic set expressions may be introduced by the algorithm. $\square$

Lemma 18 (Termination) *Let $\mathcal{C}_0$ be a collection of constraints in reduced form and standard form. Then the instance of the generic algorithm defined by Δ_2 terminates on $\mathcal{C}_0$.*

Proof: The proof proceeds by establishing a bound on the number of distinct constraints that may be introduced by the algorithm. Since the constraints encountered during the algorithm are in standard form, they must all have one of the following forms:

(a) $\mathcal{X} \supseteq f_{(i)}^{-1}(a)$,

(b) $\mathcal{X} \supseteq \{X : exp_1 \wedge \cdots \wedge exp_m\}$,

(c) $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_n$, $n \geq 2$ or

(d) $\mathcal{X} \supseteq a$,

where $a, a_1, \ldots, a_n$ are atomic set expressions, $\mathcal{X}$ is a set variable, and $f_{(i)}^{-1}$ is a projection symbol, and $exp_1, \ldots, exp_m$ are quantified conditions. Now, consider these four kinds of constraints in turn.

In case (a), note that no transformation introduces constraints involving projection operations. Hence any constraint in this class must in fact appear in $\mathcal{C}_0$, and this places a trivial bound on the number of these kinds of constraints that can be encountered during the algorithm.

Now consider case (b). Each exp_i is either of the form $s \in a$ or $s \dagger a$ such that s is a program term and a is an atomic set expression. A bound has already been established on the number of possible atomic set expressions. Now focus on the program terms s. It has already been observed that the only transformations capable of introducing new quantified set expressions are 5, 8, 12 and 13. Observe that Transformations 5, 12 and 13 only introduce subterms of program terms already appearing. Specifically, when one of Transformations 5, 12 and 13 is applied to constraints $\mathcal{C}$, then any new quantified set expression must be of the form $s \in a$ or $s \dagger a$ such that $\mathcal{C}$ contains a quantified condition $s' \in a'$ and s is a subterm of s'.

On the other hand, Transformation 8 may introduce completely new program terms. Specifically, this transformation introduces a program term $f(X_1, \ldots, X_n)$ for each occurrence of a constraint falling into case (a). As noted previously, it is assumed that the $X_1, \ldots, X_n$ are chosen in some canonical manner (for example, using some fixed listing of VAR) so that this transformation cannot be repeatedly applied to a constraint $\mathcal{X} \supseteq f_{(i)}^{-1}(a)$ to produce REDUCE($\mathcal{X} \supseteq \{X_i : f(X_1, \ldots, X_n) \in a\}$), and then REDUCE($\mathcal{X} \supseteq \{X_i' : f(X_1', \ldots, X_n') \in a\}$), etc. Hence there is exactly one application of Transformation 8 for each occurrence of a constraint that falls into case (a). Clearly the subsequent application of REDUCE (and subsequent applications of other transformation) may introduce subterms of $f(X_1, \ldots, X_n)$. To summarize then, each constraint $\mathcal{X} \supseteq f_{(i)}^{-1}(a)$ in $\mathcal{C}_0$ may introduce a finite number of new program terms, namely $f(X_1, \ldots, X_n)$, $X_1, \ldots, X_n$. It follows that there is a bound on the number of program terms introduced by Transformation 8.

This means that there is a bound on the number of distinct program terms that may be encountered during the algorithm. Combining this with the bound atomic set expressions, proves that the number of quantified conditions $s \in a$ and $s \dagger a$ is bounded. Since each *conj* is maintained in a non-redundant form, it follows that there is a bound on the number of conjunctions $exp_1 \wedge \cdots \wedge exp_m$. This implies a bound on the expressions $\{X : exp_1 \wedge \cdots \wedge exp_m\}$ because the program variable X must either appear in $\mathcal{C}$ or be introduced by an application of Transformation 8. This in turn implies the existence of a bound on the number of constraints of the form $\mathcal{X} \supseteq \{X : exp_1 \wedge \cdots \wedge exp_m\}$ that can be introduced by the algorithm since there is a bound on the number of set variables $\mathcal{X}$ (set variables are atomic set expressions).

Finally, consider cases (c) and (d). Since each constraint of the form $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_n$ is such that $\mathcal{C}_0$ contains an intersection operator of arity $m \geq n$, the bound on atomic set expressions implies a bound on the number of such constraints. Similarly, a bound may be established on the number of constraints of the form $\mathcal{X} \supseteq a$.

This completes the proof that the algorithm can only introduce a finite number of distinct constraints, and since $\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \ldots$ is an increasing sequence of constraints, it follows that at some stage it must be the case that $\mathcal{C}_{n+1} = \mathcal{C}_n$, and so the algorithm terminates. $\square$

It remains to prove that Δ_2 is complete.

Lemma 19 (Completeness) Δ_2 *is complete.*

Proof: The proof of completeness has the same basic structure as the completeness proof for the intersection-projection algorithm. Some of the cases (particularly those dealing with intersection) are just adaptations of this previous proof. Let $\mathcal{C}$ be the result of exhaustively applying Δ_2 to a collection of constraints. This implies that $\delta(\mathcal{C}) \subseteq \mathcal{C}$ for all transformations δ in Δ_2. Adopting the notation of the generic algorithm, let the sequence of constraints obtained by this exhaustive application be $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_i$ where $\mathcal{C}_i = \mathcal{C}$. Let $\mathcal{D}$ denote the subset of constraints in $\mathcal{C}$ of form $\mathcal{X} \supseteq a$ where a is a non-variable atomic set expression. Clearly $\mathcal{D} \subseteq \mathit{explicit}(\mathcal{C}) \subseteq \mathcal{C}$, and so $lm(\mathcal{D}) \subseteq lm(\mathit{explicit}(\mathcal{C})) \subseteq lm(\mathcal{C})$. The remainder of the proof shows that $lm(\mathcal{D}) \supseteq lm(\mathcal{C})$, and it is clear that this implies $lm(\mathit{explicit}(\mathcal{C})) = lm(\mathcal{C})$, as required by the definition of completeness.

To prove that $lm(\mathcal{D}) \supseteq lm(\mathcal{C})$, we shall show that $lm(\mathcal{D})$ is a model of $\mathcal{C}$. Let $\mathcal{I}_{\mathcal{D}}$ denote $lm(\mathcal{D})$. Proposition 17 provides the following characterization of $\mathcal{I}_{\mathcal{D}}$:

$$v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X}) \quad \text{iff} \quad \begin{array}{l} v \in \mathcal{I}_{\mathcal{D}}(a) \text{ for some constraint } \mathcal{X} \supseteq a \text{ in } \mathcal{C} \\ \text{where } a \text{ is non-variable atomic set expression} \end{array} \tag{7.32}$$

The remainder of the proof uses this fact to show that $\mathcal{I}_{\mathcal{D}}$ is a model of $\mathcal{C}$. Consider each possible constraint in $\mathcal{C}$ in turn:

Case (i): Consider a constraint of the form $\mathcal{X} \supseteq a$ where a is an atomic set expression. The proof here is identical to that in Lemma 13 (completeness of the intersection-projection algorithm).

Case (ii): Consider a constraint of the form $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$ where each a_i is an atomic set expression and $m \geq 2$. The proof is by induction on v and the induction hypothesis is: for all values v and for all constraints $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$, $m \geq 2$, appearing in $\mathcal{C}$,

(a) $v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m)$ implies $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$, and

(b) if $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$ is introduced by an application of Transformation 10 or 11 then $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$ implies $v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m)$.

Let v be a value such that the induction hypothesis holds for all values with fewer function symbols that v. Before considering (a) and (b), it is convenient to first prove the following statement: if v' has fewer symbols that v and $a_1, \ldots, a_k$ appear in $\mathcal{C}_i$ then

$$v' \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_k) \quad \text{iff} \quad \bigwedge_{a \in N} v' \in \mathcal{I}_{\mathcal{D}}(a) \text{ where } N = \bigcup_{j=1..k} \mathcal{N}(a_j) \quad (7.33)$$

This is proved by a secondary induction on i. Suppose that (7.33) holds for all $i' < i$. Let $a_1, \ldots, a_k$ appear in $\mathcal{C}_i$ and consider the following chain of propositions.

$$\begin{aligned} v' \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m) \quad & \text{iff} \quad \bigwedge_{j=1..m} v' \in \mathcal{I}_{\mathcal{D}}(a_j) \\ & \text{iff} \quad \bigwedge_{j=1..m} \bigwedge_{a \in \mathcal{N}(a_j)} v' \in \mathcal{I}_{\mathcal{D}}(a) \\ & \text{iff} \quad \bigwedge_{a \in N} v' \in \mathcal{I}_{\mathcal{D}}(a) \text{ where } N = \bigcup_{j=1..k} \mathcal{N}(a_j) \end{aligned}$$

The first step is just an expansion of $\cap$. For the second step, take each a_j in turn and consider two cases. If a_j is not an intersection variable then $\mathcal{N}(a_j) = \{a_j\}$ and so the second step is trivial. On the other hand, suppose that a_j *is* an intersection variable, say $\mathcal{V}_{N_j}$. Corresponding to $\mathcal{V}_{N_j}$, there exists a constraint $\mathcal{V}_{N_j} \supseteq a'_1 \cap \cdots \cap a'_l$, $l \geq 2$, that is introduced by Transformation 10 or 11. Moreover, this constraint must appear in $\mathcal{C}_{i-1}$ and $N_j = \mathcal{N}(a'_1) \cup \cdots \cup \mathcal{N}(a'_l)$. Now, since v' is smaller than v and $\mathcal{C}_{i-1}$ is constructed before $\mathcal{C}_i$, the main induction hypothesis and the secondary induction hypothesis respectively imply that

$$\begin{aligned} & v' \in \mathcal{I}_{\mathcal{D}}(a'_1 \cap \cdots \cap a'_l) \quad \text{iff} \quad v' \in \mathcal{I}_{\mathcal{D}}(\mathcal{V}_{N_j}), \qquad \text{and} \\ & v' \in \mathcal{I}_{\mathcal{D}}(a'_1 \cap \cdots \cap a'_l) \quad \text{iff} \quad \bigwedge_{a \in N_j} v' \in \mathcal{I}_{\mathcal{D}}(a) \end{aligned}$$

and the second step follows immediately. The final step in the chain follows from the definition of N. This completes the inductive proof of (7.33). The following key property is an immediate corollary of (7.33): if v' has fewer symbols than v, and $\mathcal{V}_N, a_1, \ldots, a_k$ appear in $\mathcal{C}_i$ where $N = \mathcal{N}(a_1) \cup \cdots \cup \mathcal{N}(a_k)$, then

$$v' \in \mathcal{I}_{\mathcal{D}}(\mathcal{V}_N) \quad \text{iff} \quad v' \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_k) \qquad (7.34)$$

Now consider part (a) of the main induction hypothesis. Assume that $v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m)$. It follows that $v \in a_i$, $i = 1..m$. Now, if one of the a_i is a set variable, say $\mathcal{Y}$, then (7.32) implies that there exists a constraint $\mathcal{Y} \supseteq a$ in $\mathcal{C}$ where a is a non-variable atomic set expression such that $v \in \mathcal{I}_{\mathcal{D}}(a)$.

This means that the preconditions of Transformation 6 are satisfied, and so the constraint $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_{i-1} \cap a \cap a_{i+1} \cap \cdots \cap a_m$ must appear in $\mathcal{C}$.

This argument may be repeated if necessary, and it follows that $\mathcal{C}$ must contain a constraint of the form $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$ where each a_i is a non-variable atomic set expression and $v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m)$. Now, Transformation 9 can be applied to this constraint, and since this application does not produce any new constraints, it must be the case that $\mathcal{C}$ contains a constraint of the form $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m \cap \overline{S}$ such that $v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_m \cap \overline{S})$ and each a_i is of the an atomic set expression that is not a set variable or a complement constant. Let v be $f(v_1, \ldots, v_n)$. It follows that each a_i must be of the form $f(a_{i,1}, \ldots, a_{i,n})$ such that $v_j \in \mathcal{I}_{\mathcal{D}}(a_{i,j})$, $i = 1..m$, $j = 1..n$. This implies that $v_j \in \mathcal{I}_{\mathcal{D}}(a_{1,j} \cap \cdots \cap a_{m,j})$. Since $\mathcal{C}$ contains all constraints generated by Transformation 10, it follows that $\mathcal{C}$ contains $\mathcal{X} \supseteq f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n}) \cap \overline{S}$ such that $N_j = \mathcal{N}(a_{1,j}) \cup \cdots \cup \mathcal{N}(a_{m,j})$, $j = 1..n$. By (7.34), $v_j \in \mathcal{I}_{\mathcal{D}}(\mathcal{V}_{N_j})$. Hence $v \in \mathcal{I}_{\mathcal{D}}(f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n}) \cap \overline{S})$.

Hence, $\mathcal{C}$ must contain a constraint of the form $\mathcal{X} \supseteq f(a_1, \ldots, a_n) \cap \overline{S}$ such that $f(v_1, \ldots, v_n) \in \mathcal{I}(f(a_1, \ldots, a_n) \cap \overline{S})$. In fact $\mathcal{C}$ may contain a number of constraints of this form. Pick the constraint that minimizes the cardinality of the set S. Now, Transformation 10 can be applied to this constraint. Suppose that S contains an element of the form $f(se_1, \ldots, se_n)$. This implies that $\mathcal{C}$ must contain the constraints

$$\mathcal{X} \supseteq f(a_1, \ldots, a_{j-1}, \mathcal{V}_{N_j}, a_{j+1}, \ldots, a_n) \cap \overline{S'}, j = 1..n$$

where S' is $S - \{f(se_1, \ldots, se_n)\}$ and N_j is $\mathcal{N}(a_j) \cup \{\overline{se_j}\}$, $j = 1..n$. Since $f(v_1, \ldots, v_n) \in \mathcal{I}_{\mathcal{D}}(\overline{S})$, it must be the case that

$$f(v_1, \ldots, v_n) \notin \mathcal{I}_{\mathcal{D}}(f(se_1, \ldots, se_n)).$$

Hence, for some l, $v_l \notin \mathcal{I}_{\mathcal{D}}(se_l)$. We shall now argue that

$$f(v_1, \ldots, v_n) \in \mathcal{I}_{\mathcal{D}}(f(a_1, \ldots, a_{l-1}, \mathcal{V}_{N_l}, a_{l+1}, \ldots, a_n) \cap \overline{S'}) \qquad (7.35)$$

Clearly $f(v_1, \ldots, v_n) \in \mathcal{I}_{\mathcal{D}}(\overline{S'})$ and so it suffices to prove that $f(v_1, \ldots, v_n) \in \mathcal{I}_{\mathcal{D}}(f(a_1, \ldots, a_{l-1}, \mathcal{V}_{N_l}, a_{l+1}, \ldots, a_n))$. Since $v_l \in \mathcal{I}_{\mathcal{D}}(a_l)$ and $v_l \in \overline{se_l}$, it follows from (7.34) that $v_l \in \mathcal{I}_{\mathcal{D}}(\mathcal{V}_{N_l})$. Moreover, $v_i \in \mathcal{I}_{\mathcal{D}}(a_i)$, $i = 1..n$. Hence $f(v_1, \ldots, v_n) \in \mathcal{I}_{\mathcal{D}}(f(a_1, \ldots, a_{l-1}, \mathcal{V}_{N_l}, a_{l+1}, \ldots, a_n))$ and this completes the proof of (7.35).

Now, S' is smaller than S and this violates the assumption that the constraint $\mathcal{X} \supseteq f(a_1, \ldots, a_n) \cap \overline{S}$ minimizes the cardinality of the set S. Hence, the assumption that S contains an element of the form $f(se_1, \ldots, se_n)$ must not be valid. This implies that an application of Transformation 10 to the constraint $\mathcal{X} \supseteq f(a_1, \ldots, a_n) \cap \overline{S}$ must in fact produce the constraint $\mathcal{X} \supseteq f(a_1, \ldots, a_n)$. Moreover, $\mathcal{C}$ must already contain this constraint. Hence $\mathcal{D}$ contains $\mathcal{X} \supseteq f(a_1, \ldots, a_n)$, and it follows that $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$, and this completes the proof of (a).

To prove (b), suppose that $\mathcal{X} \supseteq se$ is introduced by an application of Transformation 11 or 10. By inspection of Transformations 11 and 10, $\mathcal{X}$ must be an intersection variable. The first part of the proof shall establish that if $\mathcal{X} \supseteq se'$ appears in $\mathcal{C}_i$ then

$$v \in \mathcal{I}_{\mathcal{D}}(se') \text{ implies } v \in \mathcal{I}_{\mathcal{D}}(se) \tag{7.36}$$

The proof proceeds by induction. Suppose that (7.36) holds for all $i' < i$ and let $\mathcal{X} \supseteq se'$ be a constraint in $\mathcal{C}_i$ and let v be a value in $\mathcal{I}_{\mathcal{D}}(se')$. Now, either $\mathcal{X} \supseteq se'$ appears for the first time in $\mathcal{C}_i$, or else $\mathcal{C}_{i-1}$ contains a constraint of the form $\mathcal{X} \supseteq se'$. In the first case, (7.36) is vacuously true. Now consider the second case. Clearly the only transformations that could add the constraint $\mathcal{X} \supseteq se'$ are 6, 9, 10 and 11. We consider each of these in turn.

If Transformation 6 is used to obtain $\mathcal{X} \supseteq se'$, then $\mathcal{C}_{i-1}$ must contain constraints $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_{i-1} \cap \mathcal{Y} \cap a_{i+1} \cap \cdots \cap a_n$ and $\mathcal{Y} \supseteq a$ such that $n \geq 2$, a is an atomic set expression that is not a set variable, and se' is $a_1 \cap \cdots \cap a_{i-1} \cap a \cap a_{i+1} \cap \cdots \cap a_n$. Clearly $\mathcal{Y} \supseteq a$ appears in $\mathcal{D}$. Now, suppose that $v \in \mathcal{I}_{\mathcal{D}}(se')$. Thus $v \in \mathcal{I}(a_j)$, $j \neq i$, and $v \in \mathcal{I}(a)$. Since $\mathcal{Y} \supseteq a$ appears in $\mathcal{D}$, $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{Y})$, and so

$$v \in \mathcal{I}_{\mathcal{D}}(a_1 \cap \cdots \cap a_{i-1} \cap \mathcal{Y} \cap a_{i+1} \cap \cdots \cap a_n)$$

Since this constraint appears in $\mathcal{C}_{i-1}$ and $\mathcal{C}_{i-1}$ satisfies (7.36), it follows $v \in \mathcal{I}_{\mathcal{D}}(se')$ and this proves that (7.36) holds for i.

If Transformation 9 is used to obtain $\mathcal{X} \supseteq se'$, then $\mathcal{C}_{i-1}$ must contain a constraint $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m$, $m \geq 2$, such that se' is $\mathcal{X} \supseteq a'_1 \cap \cdots \cap a'_{m-n} \cap \overline{S}$ where $\overline{S_1}, \ldots, \overline{S_n}$ and $a'_1, \ldots, a'_{m-n}$ are subsequences of $a_1, \ldots, a_m$ such that the first subsequence contains the complement constants in $a_1, \ldots, a_m$, and the second contains the remaining atomic set expressions, and $S = S_1 \cup \cdots \cup$

S_n. Now, suppose that $v \in \mathcal{I}_\mathcal{D}(se')$. It is easy to verify that this implies $v \in \mathcal{I}_\mathcal{D}(a_i)$, $i = 1..m$, and so $v \in \mathcal{I}_\mathcal{D}(a_1 \cap \cdots \cap a_m)$. Since this constraint appears in $\mathcal{C}_{i-1}$ and $\mathcal{C}_{i-1}$ satisfies (7.36), it follows $v \in \mathcal{I}_\mathcal{D}(se')$ and this proves that (7.36) holds for i.

If Transformation 10 is used to obtain $\mathcal{X} \supseteq se'$, then $\mathcal{C}_{i-1}$ must contain a constraint $\mathcal{X} \supseteq a_1 \cap \cdots \cap a_m \cap \overline{S}$ such that $m \geq 2$, each a_i is of the form $f(a_{i,1}, \ldots, a_{i,n})$, and se' is $\mathcal{X} \supseteq f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n}) \cap \overline{S}$ where $N_j = \bigcup_{i=1..m} \mathcal{N}(a_{i,j})$, $j = 1..n$. Now, suppose that $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(se')$. This implies that $v_j \in \mathcal{I}_\mathcal{D}(\mathcal{V}_{N_j})$, $j = 1..n$, and $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(\overline{S})$. Since each v_j has fewer symbols than v, property (7.34) can be applied to show that $v_j \in \mathcal{I}_\mathcal{D}(a_{1,j} \cap \cdots \cap a_{m,j})$. It follows that $v_j \in \mathcal{I}_\mathcal{D}(a_{i,j})$, $j = 1..n$, $i = 1..m$, and so $v \in \mathcal{I}_\mathcal{D}(f(a_{i,1}, \ldots, a_{i,n}))$, $i = 1..m$. Hence $v \in \mathcal{I}_\mathcal{D}(a_1 \cap \cdots \cap a_m \cap \overline{S})$. Since this constraint appears in $\mathcal{C}_{i-1}$ and $\mathcal{C}_{i-1}$ satisfies (7.36), it follows $v \in \mathcal{I}_\mathcal{D}(se')$ and this proves that (7.36) holds for i.

Finally, if Transformation 11 is used to obtain $\mathcal{X} \supseteq se'$, then $\mathcal{C}_{i-1}$ must contain a constraint $\mathcal{X} \supseteq f(a_1, \ldots, a_n) \cap \overline{S}$ such that $f(\top, \ldots, \top) \notin S$ and either (i) se' is $\mathcal{X} \supseteq f(a_1, \ldots, a_n)$ and $f'(\cdots) \in S$ implies $f \neq f'$, or else (ii) se' is $\mathcal{X} \supseteq f(a_1, \ldots, a_{k-1}, \mathcal{V}_{N_k}, a_{k+1}, \ldots, a_n) \cap \overline{S'}$, for some k, $1 \leq k \leq n$, such that $f(se_1, \ldots, se_n) \in S$, S' is $S - \{f(se_1, \ldots, se_n)\}$ and N_k is $\mathcal{N}(a_k) \cup \{\overline{se_k}\}$. Now, suppose that $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(se')$. In case (i), it is immediate that $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(f(a_1, \ldots, a_n))$. It is also easy to verify that $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(\overline{S})$, and so $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(f(a_1, \ldots, a_n) \cap \overline{S})$. In case (ii), $v_j \in \mathcal{I}_\mathcal{D}(a_j)$, $j \neq k$, $v_k \in \mathcal{I}_\mathcal{D}(\mathcal{V}_{N_k})$ and $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(\overline{S'})$. Applying (7.34) proves that $v_j \in \mathcal{I}_\mathcal{D}(a_j \cap \overline{se_j})$. Hence $v_j \in \mathcal{I}_\mathcal{D}(a_j)$. Also, $v_j \in \mathcal{I}_\mathcal{D}(\overline{se_j})$, and it is easy to verify that this implies $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(\overline{f(se_1, \ldots, se_n)})$. Combining this with $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(\overline{S'})$ proves that $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(\overline{S})$. Thus, in either case, $f(v_1, \ldots, v_n) \in \mathcal{I}_\mathcal{D}(se')$. Since this constraint appears in $\mathcal{C}_{i-1}$ and $\mathcal{C}_{i-1}$ satisfies (7.36), it follows $v \in \mathcal{I}_\mathcal{D}(se')$ and this proves that (7.36) holds for i.

This completes the proof of (7.36), and (b) can now be proved as follows. If $v \in \mathcal{I}_\mathcal{D}(\mathcal{X})$ then there exists a constraint $\mathcal{X} \supseteq a$ in $\mathcal{D}$ such that $v \in \mathcal{I}_\mathcal{D}(a)$. Since $\mathcal{X} \supseteq a$ also appears in $\mathcal{C}$, it follows from (7.36) that $v \in \mathcal{I}_\mathcal{D}(a_1 \cap \cdots \cap a_m)$.

Case (iii): Consider a constraint of the form $\mathcal{X} \supseteq \{X : conj\}$. Suppose that $v \in \mathcal{I}_\mathcal{D}(\{X : conj\})$. This implies that there is an environment ρ such that $\rho \in \mathcal{I}_\mathcal{D}(conj)$ and $\rho(X) = v$. Now, consider the following cases of $conj$:

Case (iii)(a): Suppose that *conj* is in compaction form. Since $\rho \in \mathcal{I}_{\mathcal{D}}(\{X : conj\})$, it follows that $\mathcal{I}_{\mathcal{D}}(\lceil Y \rceil\, conj)$ is non-empty for each Y appearing in *conj*, and hence each $lm(explicit(\mathcal{C}))(\lceil Y \rceil\, conj)$ is also non-empty. By Proposition 43, $\mathcal{I}_{\mathcal{D}}(\{X : conj\}) = \mathcal{I}_{\mathcal{D}}(\lceil X \rceil\, conj)$. It follows that $v \in \mathcal{I}_{\mathcal{D}}(\lceil X \rceil\, conj)$. Moreover, since each $lm(explicit(\mathcal{C}))(\lceil Y \rceil\, conj)$ is non-empty, the preconditions of Transformation 14 are satisfied, and so $\mathcal{C}$ must contain the constraint $\mathcal{X} \supseteq \lceil X \rceil\, conj$. This constraint falls either into case (i) or (ii) considered above. Since these cases have already been established for v, it follows that $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$.

Case (iii)(b): Suppose that *conj* does not contain any quantified conditions of the form $s \dagger a$. Since *conj* is in reduced form, each condition in *conj* is either of the form $X \in a$ or $s \in \mathcal{X}$ where X is a program variable, s is a program term consisting of program variables and function symbols, a is an atomic set expression and $\mathcal{X}$ is a set variable. Now, let $s_1 \in a_1, \ldots, s_n \in a_n$ be a listing of the quantified conditions in *conj* that do not contain set variables (such conditions must be of the form $s \in a$ where a is ground). Define $\nabla(conj)$ to be the number of function symbols appearing in $s_1, \ldots, s_n$. The proof for this case shall be argued by induction on ∇. In the base case where $\nabla(conj)$ is 0, each quantified condition in *conj* must have the form $X \in a$ where X is a program variable. Hence *conj* is in compaction form, and so the proof for the base case follows from case (iii)(a).

For the induction case, suppose that $\nabla(conj) \leq j$ implies that $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$, and consider *conj* such that $\nabla(conj) = j + 1$. Since $\nabla(conj) > 0$, it must be the case that *conj* is of the form $conj' \wedge s \in a$ such that s contains some function symbols. This implies that a must be a variable, say $\mathcal{Y}$. Moreover, ρ is such that $\rho(s) \in \mathcal{I}_{\mathcal{D}}(\mathcal{Y})$. Hence, there exists a constraint $\mathcal{Y} \supseteq a'$ in $\mathcal{C}$ such that a' is a non-variable atomic expression and $\rho(s) \in \mathcal{I}_{\mathcal{D}}(a')$. Hence Transformation 5 is applicable and it follows that $\mathcal{C}$ must contain $\textsc{reduce}(\mathcal{X} \supseteq \{X : conj' \wedge s \in a'\})$. Clearly $v \in \mathcal{I}_{\mathcal{D}}(\{X : conj' \wedge s \in a'\})$. Moreover, Lemma 14 proves that there exists a constraint $\mathcal{X} \supseteq se'$ in $\textsc{reduce}(\mathcal{X} \supseteq \{X : conj' \wedge s \in a'\})$ such that $v \in \mathcal{I}_{\mathcal{D}}(se')$. By inspection of the steps that make up REDUCE, it is clear that se' is in fact of the form $\{X : conj''\}$. It remains to show that $\nabla(conj'') < \nabla(conj)$.

Now, it is easy to verify that each step of REDUCE does not increase ∇. In fact, except for step (iv), all steps replace a quantified set expression by a (possibly empty) collection of quantified set expressions such that each new quantified set expression contains fewer function symbols in its program terms than the original quantified set expression. In other words, for these steps, the new quantified set expression have a strictly smaller ∇ than the original quantified set expression. Step (iv), on the other hand, may introduce function symbols (by duplicating a program term), but the quantified conditions involved are of the form $s'' \in a''$ where a'' is ground, and hence they do not contribute to ∇, and so step (iv) leaves ∇ unchanged. Now, consider the quantified conditions $s \in a'$ constructed by Transformation 5. Clearly at least one step of REDUCE is applicable to $s \in a$ (since s is not a program variable, and a is not a set variable). Now, consider two cases. If a' is ground, then $\nabla(conj' \wedge s \in a') < \nabla(conj' \wedge s \in a)$, and since REDUCE does not increase ∇, it follows that $\nabla(conj'') < \nabla(conj)$. If a' is not ground, then one of the steps other than (iv) is applicable, and again it follows that $\nabla(conj'') < \nabla(conj)$. This completes the inductive proof of case (iii)(b).

Case (iii)(c): Suppose that *conj* contains an apartness condition $s \dagger a$. The proof for this case is by induction on the number of apartness conditions in *conj*. If *conj* does not contain any such conditions, then it falls into case (iii)(b), and this proves the base case. For the induction case, suppose that *conj* has less than $k \geq 1$ apartness conditions then $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$, and consider the case where *conj* has exactly k apartness conditions. Clearly *conj* has at least one apartness condition, say $s \dagger a$. Since $\rho \in \mathcal{I}_{\mathcal{D}}(conj)$, it follows that $\rho \in \mathcal{I}_{\mathcal{D}}(s \dagger a)$. Hence there is some value v' in $\mathcal{I}_{\mathcal{D}}(a)$ such that $\rho(s) \neq v'$. Since $\mathcal{I}_{\mathcal{D}} \subseteq lm(explicit(\mathcal{C}))$, this implies that either Transformation 12 or 13 is applicable, depending on whether v' is the only value in $lm(explicit(\mathcal{C}))(a)$. In either case, Proposition 41 or Proposition 42 can be applied to show that there is a constraint $\mathcal{X} \supseteq \{X : conj'\}$ in $\mathcal{C}$ such that $v \in \mathcal{I}_{\mathcal{D}}(\{X : conj'\})$ and $conj'$ contains exactly one fewer apartness condition than *conj*. Hence $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$ follows from the induction hypothesis.

Case (iv): Consider a constraint of the form $\mathcal{X} \supseteq f_{(i)}^{-1}(a)$. Suppose that $v \in \mathcal{I}_{\mathcal{D}}(f_{(i)}^{-1}(a))$. Clearly Transformation 8 is applicable, and so $\mathcal{C}$ contains the constraint $\mathcal{X} \supseteq \{X_i : f(X_1, \ldots, X_n) \in a\}$. By Proposition 37, $\mathcal{I}_{\mathcal{D}}(\{X : conj\}) = \mathcal{I}_{\mathcal{D}}(f_{(i)}^{-1}(a))$. It follows that $v \in \mathcal{I}_{\mathcal{D}}(\{X : conj\})$. Now, since $\mathcal{X} \supseteq \{X_i : f(X_1, \ldots, X_n) \in a\}$ appears in $\mathcal{C}$, case (iv) can be applied to prove that $v \in \mathcal{I}_{\mathcal{D}}(\mathcal{X})$, and this completes the proof for this case. $\square$

Combining the above lemmas with Theorem 7 proves that:

Theorem 9 (Correctness of Quantified Set Expression Algorithm) *Let $\mathcal{SC}_P$ be the set constraints corresponding to a program P, and let $\mathcal{C}_0$ = $\text{REDUCE}(\text{STANDARDIZE}(\mathcal{SC}_P))$. When input with $\mathcal{C}_0$, the instance of the generic algorithm defined by Δ_2 terminates and outputs explicit form constraints $\mathcal{C}_{out}$ such that $lm(\mathcal{C}_{out}) =_{var(\mathcal{SC}_P)} lm(\mathcal{SC}_P)$.*

We note that many decisions about the design of the algorithm were made to simplify presentation at the expense of efficiency. We now outline a number of these. First, consider the transformation involving substitution into quantified set expressions (Transformation 5). If $\mathcal{C}$ contains

$$\begin{array}{l} \mathcal{X} \supseteq \{X : X \in \mathcal{Y} \wedge f(X) \in \mathcal{Z}\} \\ \mathcal{Y} \supseteq f(\mathcal{U}) \\ \mathcal{Y} \supseteq f(\mathcal{W}) \end{array}$$

then Transformation 5 adds the constraints

$$\begin{array}{l} \mathcal{X} \supseteq \{X : X \in f(\mathcal{U}) \wedge f(X) \in \mathcal{Z}\}, \\ \mathcal{X} \supseteq \{X : X \in f(\mathcal{W}) \wedge f(X) \in \mathcal{Z}\}. \end{array} \tag{7.37}$$

These constraints are unnecessary because the compaction transformation only requires that the left hand side of a quantified condition be a variable; the right hand side does not have to be a non-variable expression. Moreover, these extra constraints can lead to further redundant constraints, and may introduce unnecessary intersection variables. To illustrate this, suppose that the only lower bound for $\mathcal{Z}$ is $Z \supseteq f(\mathcal{Y})$. Substituting this constraint into the constraints (7.37) eventually leads to the constraint $\mathcal{X} \supseteq f(\mathcal{U}) \cap f(\mathcal{W})$, and this may introduce a new intersection variable. Note that substituting $Z \supseteq f(\mathcal{Y})$ into $\mathcal{X} \supseteq \{X : X \in \mathcal{Y} \wedge f(X) \in \mathcal{Z}\}$ leads to $\mathcal{X} \supseteq \{X : X \in \mathcal{Y}\}$, which via compaction yields $\mathcal{X} \supseteq \mathcal{Y}$, and by further substitution to $\mathcal{X} \supseteq f(\mathcal{U})$ and $\mathcal{X} \supseteq f(\mathcal{W})$. To avoid such redundant substitution steps, Transformation 5 can be modified so that s is required to be a non-variable program term.

Second, consider the redundancies inherent in the original constraints $\mathcal{SC}_P$. In particular, $\mathcal{SC}_P$ contains many groups of constraints of the form

$$\mathcal{X}^1 \ \supseteq \ \{X_1 : conj\}, \ldots, \mathcal{X}^n \ \supseteq \ \{X_n : conj\} \tag{7.38}$$

Now, during execution of the algorithm, the occurrences of *conj* are treated in an identical manner in the sense that if a transformation is applied to one occurrence, then it can be applied to another. Hence, the work of "solving" *conj* is duplicated for each occurrence in the initial constraints. It is therefore appropriate to consider grouping these constraints together into a form such as

$$(\mathcal{X}^1, \ldots, \mathcal{X}^n) \supseteq \{(X_1, \ldots, X_n) : conj\}.$$

whose meaning is identical to the constraints (7.38). The only main change required for this new kind of constraint involves the compaction transformation. Specifically, this transformation becomes:

> If $\mathcal{C}$ contains $(\mathcal{X}^1, \ldots, \mathcal{X}^n) \supseteq \{(X_1, \ldots, X_n) : conj\}$ such that $conj$ is in compaction form and $lm(explicit(\mathcal{C}))(\bigcap_X conj)$ is non-empty, for each $X \in \{\mathcal{X}^1, \ldots, \mathcal{X}^n\}$, then output the constraints $\mathcal{X} \supseteq \bigcap_X conj$, for each $X \in \{\mathcal{X}^1, \ldots, \mathcal{X}^n\}$.

We also observe that the bounds described in the termination argument (see Lemma 18) can be significantly tightened. By doing so, it is fairly straightforward to obtain an EXPTIME bound on the execution of the algorithm. We note that EXPTIME bounds have been reported by Fruhwirth, Shapiro, Vardi and Yardeni [18] for a related class of set constraints.

7.7 Related Work

We conclude this chapter with a discussion of the literature related to set constraint algorithms. Early work by Reynolds [63] describes a simplification algorithm for set constraints involving projection. The motivation for this work was the inference of data type definitions in a first order functional language. Subsequently, a similar algorithm was independently developed by Jones and Muchnick [32]. In essence, these algorithms consist of Transformations 1, 2 and 3. Further work by [30, 31, 50, 56] has extended the basic approach to higher order functions, binding time analysis and analysis for compile-time garbage collection and globalization of function parameters. Again, projection is the only operator employed.

Constraints involving notions of both intersection and projection were first used by Mishra [48] to approximate the success set of a logic program. Specifically, the constraints contained intersection and a form of projection.

However, for algorithmic reasons, an approximate form of union was used (see the discussion of $\uplus$ in Section 5.6, page 134). In essence, this restriction ensured that the set of atoms and terms that could be defined were *tuple-distributive* in the sense that they were closed under the $\star$ operator defined on page 134. Moreover, only partial algorithms were given.

The first decidability results for set constraints were obtained by Heintze and Jaffar in [21, 22]. In [21], an algorithm was presented for solving constraints involving quantified expressions of the form $\{X : s \in se\}$ where s is a program term constructed from program variables and function symbols. The purpose of this was to obtain a simple and decidable approximation to the success set of a logic program (the approximation defined is equivalent to bottom-up sba_P). In [22] the set constraint calculus was formalized and studied them in an abstract setting (most of our definitions and notation for set constraints are taken from this paper). Its main result of was a decision procedure to determine the satisfiability of *definite* set constraints, which are constraints of the form $a \supseteq se$ where a is a set expression that contains no set operators and se is a set expression whose set operators are projections and intersections. Collections of definite constraint have least models whenever they are satisfiable.

Soon after writing [21, 22], we discovered an alternative proof of the results therein using a reduction to a result by Filé [16]. To motivate this reduction, first note that for some programs P, bottom-up set based analysis is exact in the sense that, using bottom-up semantics, $sba_P = lm(\mathcal{EC}_P)$. A syntactic characterization of a class of programs with this property is given in [21]; call this class EXACT(sba). Now, the main result of [21] essentially shows that bottom-up sba_P is decidable and is a regular set. As a corollary, the success set of all programs in EXACT(sba) is decidable and regular. Moreover, using the transformations described in [25], an arbitrary program P can be transformed to a program P' in EXACT(sba) such that $sba_P = sba_{P'}$ (for bottom-up semantics). This means that the problem of computing bottom-up sba_P is equivalent to the problem of computing $lm(\mathcal{EC}_P)$ for programs in EXACT(sba). Now, in [16], Filé defines a subclass of logic programs based on an extended notion of tree automata called *pattern replacing automata*, and shows that the success set of all programs in this class is decidable and regular. The key step for using this result to prove the decidability and regularity of bottom-up sba_P is a transformation that maps any program in EXACT(sba) into an equivalent program in Filé's class.

We now review some of the main works subsequent to ours. In [18], Fruhwirth, Shapiro, Vardi and Yardeni provided another proof of the decidability of sba_P. Their proof uses a technique very similar to the above reduction to Filé's result, although they were unaware of his result and essentially gave an alternative proof of it. In [5] Aiken and Murphy presented an algorithm for set constraints involving intersection and complement but omitting projection.

Very recently, Bachmair, Gandzinger and Waldmann [8] have obtained an elegant proof of the decidability of the satisfaction problem for a large subclass of set constraints involving complementation, intersection and projection (in particular, the class they consider properly contains the definite constraints considered in [22] and the constraints considered in [5]). The basis of their result is a translation from set constraints into predicate calculus formulas constructed from monadic predicates, variables and quantifiers (note that there are no function symbols). We briefly outline the approach.

It has long been recognized that there are close connections between set constraints and various fragments of logic (for example, see [22], where it is observed that results by Rabin [62] prove the decidability of monadic set constraints). Such relationships exists because set based reasoning can be expressed in the predicate calculus by regarding a monadic predicate as the set of values on which it is true. Hence, a set constraint $\mathcal{X} \supseteq f(\mathcal{X}) \cup a$ can be translated into the formula $P_{\mathcal{X}}(a) \wedge (P_{\mathcal{X}}(x) \Rightarrow P_{\mathcal{X}}(f(x))$, where $P_{\mathcal{X}}$ is the predicate introduced to capture the set variable $\mathcal{X}$.

The key idea of [8] is the use of skolemization to establish a correspondence between set constraints and a class of formulas that was shown decidable by Löwenheim [42] (see [2] for a somewhat simpler proof). In essence, set constraints are equivalent to predicate calculus formulas that are the result of skolemizing a formula with monadic predicates that is in prenex normal form. For example, consider the set constraint $\mathcal{X} \supseteq f(\mathcal{X})$ where $\Sigma = \{a, f\}$. This can be written in the predicate calculus as

$$P_{\mathcal{X}}(x) \Rightarrow P_{f(\mathcal{X})}(x) \quad \wedge \quad P_{f(\mathcal{X})}(f(x)) \Leftrightarrow P_{\mathcal{X}}(x) \quad \wedge \quad P_{f(\mathcal{X})}(a) \Leftrightarrow \mathit{false}$$

where the predicates $P_{\mathcal{X}}$ and $P_{f(\mathcal{X})}$ capture the values of $\mathcal{X}$ and $f(\mathcal{X})$ respectively. Now, this formula is just the result of skolemizing[2]

[2]Note that this formula contains only one occurrence of the variable f, and this occurrence appears in the expression $P_{f(\mathcal{X})}(f)$, where $P_{f(\mathcal{X})}$ is a predicate symbol and f is the bound variable

$$\exists a \forall x \exists f \left(P_{\mathcal{X}}(x) \Rightarrow P_{f(\mathcal{X})}(x) \quad \wedge \quad P_{f(\mathcal{X})}(f) \Leftrightarrow P_{\mathcal{X}}(x) \quad \wedge \quad P_{f(\mathcal{X})}(a) \Leftrightarrow \mathit{false} \right)$$

which is in prenex normal form and consists of only variables and monadic predicates symbols. The specific subclass of set constraints considered by [8] can be characterized as constraints of the form $se_1 \supseteq se_2$ such that se_1 does not contain the projection symbol. The general set constraint problem posed in [22] (that is, arbitrary constraints involving complementation, intersection and projection) is still open.

We now provide an algorithmic comparison between this chapter and other works in the literature. The set constraint algorithm presented in this chapter is based on algorithms developed by Heintze and Jaffar [21, 22, 23]. The main difference is that we consider more general quantified set expressions. In particular, [21] was restricted to quantified expressions that are of the form $\{X : \mathit{conj}\}$ such that each quantified condition in conj is of the form $s \in se$ where s is constructed from function symbols and program variables, and se is a set expression. [23] essentially considered quantified conditions of the the form $X \in \mathcal{X}$ and $X \dagger \mathcal{X}$ where X is a program variable and $\mathcal{X}$ is a set variable. In contrast, the algorithm presented in this chapter deals with quantified conditions of the form $s \in se$ and $s \dagger se$, where s may contain function symbols, projections and program variables, and se is a set expression[3]. It also deals with complement constants. The extensions to quantified expressions are necessary because the imperative language considered in this thesis is much more general than that used in [23]. We note that it is the appearance of projections in quantified conditions that necessitates the use of the safeness invariant.

There are two main alternatives to the approach we have adopted for computing set program approximations. The first is based on the transformation of set constraints into the class considered by Filé [16], and the second is based on the transformation of set constraints into the monadic predicate formulas considered by Bachmair, Gandzinger and Waldmann [8]. While these approaches involve simpler correctness proofs, the approach of [22], which we have adapted, has a number of important advantages. Specifically, it is more direct than the other methods and remains entirely within the framework of set constraints. Moreover, the algorithms involved are

in question.

[3]Strictly speaking, for decidability reasons, the program term s in $s \in se$ cannot contain completely arbitrary combinations of function symbols, projections and variables. See page 197 for further details.

simpler and more intuitive (although the proofs are not). Moreover, it provides greater flexibility, yields an explicit representation of the least model the constraints, and appears to be more amenable to implementation. We now expand on these last three points.

The algorithm in [22] is very flexible in the sense that it can be extended in a number of ways to deal with a variety of set operators arising from the analysis of different programming languages (as has been exploited in this thesis). In contrast, the other approaches typically involve translations into (decidable subclasses of) other formal systems, and intuitive extensions to the set constraints do not usually map into intuitive extensions in these formal systems. This is either because the transformations themselves do not make sense on the extensions, or else the translation of the extended constraints gives rise to formulas in the formal system that does not satisfy the relevant syntactic criteria required for decidability. For example, there does not appear to be any way to extend [8] so that the reverse skolemization transformation can be applied to constraints that contain quantified set expressions. Similarly, although [16] can be used to solve set constraints involving a restricted form of quantified set expressions, this method cannot be extended to solve quantified set expressions involving apartness conditions or quantified conditions of the form $s \in se$ where s contains projection symbols.

An explicit representation of the least model of the constraints is particularly important for program analysis applications. This representation provides the characterization of the structure of possible run-time values that is needed for many compile-time code improvements. Such an explicit representation is computed by the algorithm in [22] and the algorithm obtained using [16]. In contrast, although the algorithm of [8] provides a method of answering questions about the least model (including membership and non-emptiness), it does not provide any notion of explicit representation of the least model.

The set constraints algorithms based on [8] and [16] are complex and highly combinatorial in nature and do not appear to provide a basis for implementation. In contrast, the algorithm in [22] is very simple and appears to be better suited to implementation. In particular, it can be reformulated in such a way that simple operations such as projection can be treated specially and implemented cheaply. More generally, because this algorithm is very direct, it is easy to take advantage of the structural properties of set constraints that arise from typical programs. This appears to be crucial for practical implementation of set based analysis. On the other hand, because the approaches based on [8] and [16] use involved transformations into other formal systems, such properties are more difficult to exploit.

Chapter 8

Implementation

The algorithm described in Chapter 7 focussed on the issue of decidability of set based analysis. In particular, numerous aspects of the algorithm were designed for clarity rather than efficiency. As a result, a straightforward implementation of this algorithm gives very poor performance. This chapter describes the design and implementation of a prototype system for practical set based analysis. In particular, we show that substantial progress can be made by redesigning the algorithms, employing appropriate representation techniques, and removing various forms of redundancy. We provide empirical evidence to show that practical set based analysis is within reach.

8.1 Introduction

This chapter is a progress report on an ongoing effort to incorporate set based analysis in an experimental compiler, and focuses on one of the main uncertainties of set based analysis: its computational cost. It is clear that solving set constraints can be expensive in the worst case, and this is due to the exponential behavior of the intersection operation (see [18] for a formal account of the exponential behavior of one class of set constraints). However it is not clear whether worst case behavior is a good indication of the practicality of set based analysis, since programs rarely exhibit the extremes of behavior used in worst case analysis. For example, in the worst case, the arity of predicate and function symbols may increase linearly with the size of a program, but this is rarely the case in practice. Moreover, many programs have a very hierarchical structure and mutual recursion rarely extends beyond a small number of predicates.

We address the question of practicality by developing and evaluating an implementation of the set constraint simplification algorithm. For simplicity, we shall restrict our attention to intersection-projection constraints (see Section 7.5), and for convenience we shall write these constraints as equalities. Specifically, the constraints considered in the implementation are of the form $\mathcal{X}_1 = se_1, \ldots, \mathcal{X}_n = se_n$ where $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are distinct set variables, and each se_i is constructed from union, function symbols, set variables and intersection and projection operators.

Corresponding to a program P, constraints of this form can be constructed to analyze P, in a manner similar to the construction of $\mathcal{SC}_P$. The differences between these constraints and $\mathcal{SC}_P$ are relatively minor, although in general the constraints $\mathcal{SC}_P$ are slightly more accurate. Most importantly, the constraints used here provide a similar uniform treatment of structures, and represent the core part of the more complex constraints. Also, the proofs of correctness of the constraints can be adapted from the correctness of $\mathcal{SC}_P$. Since the focus of this chapter is on solving set constraints, and not on the specific construction of constraints from a program, we shall omit the full details of this construction, and instead give some examples. Throughout this chapter we shall focus on logic programs because they tend to yield constraints that are more difficult to solve.

The method for constructing constraints from a program is essentially

	$Ret_p = p(\mathcal{X})$
$p(X) \leftarrow q(X), r(X).$	$Ret_q = q(a) \cup q(f(\mathcal{Y}))$
$q(a).$	$Ret_r = r(f(\mathcal{Z}))$
$q(f(Y)) \leftarrow q(Y).$	$\mathcal{X} = q^{-1}(Ret_q) \cap r^{-1}(Ret_r)$
$r(f(Z)).$	$\mathcal{Y} = q^{-1}(Ret_q)$
	$\mathcal{Z} = \top$

Figure 8.1: **Bottom-Up Set Constraints**

the same as that outlined in Chapter 2 and then formalized in Chapter 4. First, set variables are introduced to capture the sets of values of each program variable at each point in the program. Then constraints are written between these sets to safely approximate the local consistency conditions of the program. Figure 8.1 illustrates the construction of constraints to approximate the bottom-up semantics of a logic program. The main difference between these constraints and $\mathcal{SC}_P$ is that, for convenience, we have used variables Ret_p, Ret_q and Ret_r to capture the sets of ground atoms in the success set corresponding to the predicates p, q and r respectively. As before, set variables $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ are used to capture the sets of values for the program variables X, Y and Z respectively. For example, the constraint $\mathcal{X} = q^{-1}(Ret_q) \cap r^{-1}(Ret_r)$ indicates that the set of values for the program variable X consists of those values v such that $q(v)$ is in Ret_q and $r(v)$ is in Ret_r.

Figure 8.2 shows how constraints may be constructed for the analysis of a logic program under a top-down left-to-right execution strategy. Recalling the notation for program points, note that program point 3 indicates program execution just before $q(X)$ is called in the body of the rule $p(X) \leftarrow q(X), r(X)$, point 4 indicates execution just before $r(X)$, and point 5 indicates execution after both body atoms have succeeded. As in $\mathcal{SC}_P$, a set variable is introduced to describe the values of each program variable at each program point. The set variables $\mathcal{X}^3$, $\mathcal{X}^4$, and $\mathcal{X}^5$ respectively denote the values of $\mathcal{X}$ at points 3, 4 and 5. The variables $Call_p$, $Call_q$ and $Call_r$ have been introduced to capture the possible calls to p, q and r. For example, the constraint $\mathcal{X}^5 = p^{-1}(Call_p) \cap q^{-1}(Ret_q)$ indicates that the values of X at point 3 consists of all v such that $p(v)$ is in $Call_p$ and $q(v)$ is in Ret_q. Additional initial goals can be accommodated by appropriately modifying the equations for $Call_p$, $Call_q$ and $Call_r$. Figure 8.3 contains another ex-

	$\mathcal{W}^1 = \top$
	$\mathcal{W}^2 = p^{-1}(Ret_p)$
	$\mathcal{X}^3 = p^{-1}(Call_p)$
	$\mathcal{X}^4 = p^{-1}(Call_p) \cap q^{-1}(Ret_q)$
2. $\leftarrow p(W)^1.$	$\mathcal{X}^5 = p^{-1}(Call_p) \cap q^{-1}(Ret_q) \cap\ r^{-1}(Ret_r)$
5. $p(X) \leftarrow q(X)^3,\ r(X)^4.$	$\mathcal{Y}^7 = r^{-1}(Call_r)$
6. $q(a).$	$Ret_p = p(\mathcal{X}^5)$
7. $r(Y).$	$Ret_q = q(a)$
	$Ret_r = r(\mathcal{Y}^7)$
	$Call_p = p(\mathcal{W}^1)$
	$Call_q = q(\mathcal{X}^3)$
	$Call_r = r(\mathcal{X}^4)$

Figure 8.2: **Top-Down Set Constraints**

2. $\leftarrow app(cons(b, nil), cons(c, nil), V)^1.$
3. $app(nil, W, W).$
5. $app(cons(X, L), Y, cons(X, Z)) \leftarrow app(L, Y, Z)^4.$

$$
\begin{array}{rcl}
\mathcal{V}^1 &=& \top \\
\mathcal{V}^2 &=& app_{(3)}^{-1}(Ret_{app}) \\
\mathcal{W}^3 &=& app_{(2)}^{-1}(Call_{app}) \cap app_{(3)}^{-1}(Call_{app}) \\
\mathcal{X}^4 &=& cons_{(1)}^{-1}(app_{(1)}^{-1}(Call_{app})) \cap cons_{(1)}^{-1}(app_{(3)}^{-1}(Call_{app})) \\
\mathcal{L}^4 &=& cons_{(2)}^{-1}(app_{(1)}^{-1}(Call_{app})) \\
\mathcal{Y}^4 &=& app_{(2)}^{-1}(Call_{app}) \\
\mathcal{Z}^4 &=& cons_{(2)}^{-1}(app_{(3)}^{-1}(Call_{app})) \\
\mathcal{X}^5 &=& cons_{(1)}^{-1}(app_{(1)}^{-1}(Call_{app})) \cap cons_{(1)}^{-1}(app_{(3)}^{-1}(Call_{app})) \\
\mathcal{L}^5 &=& cons_{(2)}^{-1}(app_{(1)}^{-1}(Call_{app})) \cap app_{(1)}^{-1}(Ret_{app}) \\
\mathcal{Y}^5 &=& app_{(2)}^{-1}(Call_{app}) \cap app_{(2)}^{-1}(Ret_{app}) \\
\mathcal{Z}^5 &=& cons_{(2)}^{-1}(app_{(3)}^{-1}(Call_{app})) \cap app_{(3)}^{-1}(Ret_{app}) \\
Ret_{app} &=& app(nil, \mathcal{W}^3, \mathcal{W}^3) \cup app(cons(\mathcal{X}^5, \mathcal{L}^5), \mathcal{Y}^5, cons(\mathcal{X}^5, \mathcal{Z}^5)) \\
Call_{app} &=& app(cons(b, nil), cons(c, nil), \mathcal{V}^1) \cup app(\mathcal{L}^4, \mathcal{Y}^4, \mathcal{Z}^4)
\end{array}
$$

Figure 8.3: **The Append Program and Its Top-Down Constraints**

ample of the construction of the top-down constraints, this time involving the append program.

8.2 The Basic Set Constraint Algorithm

We present a reformulation of the algorithm for intersection-projection constraints from Section 7.5. The main difference from the algorithm in Section 7.5 is in this presentation we have attempted to minimize the number of transformation steps that have to be performed by combining the substitution, projection simplification and intersection simplification transformations.

The first step of the algorithm is a preprocessing stage that puts the constraints in a convenient form. This form essentially corresponds to a restriction of standard form. Define that an *intersector* is of the form $a_1 \cap \cdots \cap a_n$ where the a_i are atomic set expressions. A *projector* is of the form $f_{(i)}^{-1}(a)$ where a is an atomic set expression. A constraint is in *restricted standard form* if it is of the form $\mathcal{X} = se_1 \cup \cdots \cup se_n$ where at most one of the se_i is either a projector or an intersector, and the remaining se_i are atomic expressions. A collection of constraints $\mathcal{X}_1 = se_1, \ldots, \mathcal{X}_m = se_m$ is in *restricted standard form* if $\mathcal{X}_1, \ldots, \mathcal{X}_m$ are distinct set variables, and each constraint is in restricted standard form.

The method for converting constraints to restricted standard form involves repeatedly identifying an occurrence of a set expression that does not satisfy the restricted standard form definition, replacing it by a new variable and then adding a new equation between the new variable and the replaced set expression. The details are very similar to the conversion of constraints to standard form in Section 7.3, page 174. Importantly, given a collection $\mathcal{C}$ of constraints of the form $\mathcal{X}_1 = se_1, \ldots, \mathcal{X}_m = se_m$ where $\mathcal{X}_1, \ldots, \mathcal{X}_m$ are distinct variables, the resulting constraints, call them $\mathcal{C}_0$, are in restricted standard form, and $lm(\mathcal{C}) =_{var(\mathcal{C})} lm(\mathcal{C}_0)$.

The bulk of the algorithm consists of the exhaustive application of four transformations to restricted standard form constraints. Before presenting these transformations, it is convenient to represent constraints using an array indexed by set variables. Specifically, let $\mathcal{C}$ be a collection of restricted standard form constraints, and define $rhs_{\mathcal{C}}$ as follows

$$rhs(\mathcal{X}) \stackrel{\text{def}}{=} \begin{cases} \{se_1, \ldots, se_n\} & \text{if } \mathcal{C} \text{ contains } \mathcal{X} = se_1 \cup \cdots \cup se_n \\ \{\} & \text{otherwise} \end{cases}$$

where $\mathcal{X}$ ranges over the set variables in $\mathcal{C}$. It is convenient to extend this notation to atomic set expressions, so that $rhs(a)$ denotes $rhs(\mathcal{X})$ if a is the set variable $\mathcal{X}$, and $rhs(a)$ denotes $\{a\}$ if a is not a set variable. We can now describe the first three transformation steps.

Transformation 15 *If $\mathcal{Y} \in rhs(\mathcal{X})$ and $a \in rhs(\mathcal{Y})$ is an atomic expression then add a to $rhs(\mathcal{X})$.* □

Transformation 16 *If $f^{-1}_{(i)}(a) \in rhs(\mathcal{X})$ and $rhs(a)$ contains $\top$ then add $\top$ to $rhs(\mathcal{X})$.* □

Transformation 17 *If $f^{-1}_{(i)}(a) \in rhs(\mathcal{X})$ and $rhs(a)$ contains $f(a_1, \ldots, a_n)$ then add a_i to $rhs(\mathcal{X})$ if $lm(explicit(\mathcal{C}))(f(a_1, \ldots, a_n)) \neq \{\}$.* □

The last transformation simplifies intersectors. Again we use the $\mathcal{V}_N$ naming scheme for any new variables introduced during the simplification of intersections.

Transformation 18 *If $a_1 \cap \cdots \cap a_m \in rhs(\mathcal{X})$ and there exists a sequence $a'_1, \ldots, a'_m$ such that $a'_i \in rhs(a_i)$, $i = 1..m$, and for some $f \in \Sigma$, each a'_i has the form $f(\cdots)$ or $\top$, then let $a''_1, \ldots, a''_k$ be the elements of $a'_1, \ldots, a'_m$ of the form $f(\cdots)$ and*

- *if $k = 0$ then add $\top$ to $rhs(\mathcal{X})$;*
- *if $k > 0$, let the arity of f be n, let a''_i be $f(a_{i,1}, \ldots, a_{i,n})$, $i = 1..k$, let $N_j = \mathcal{N}(a_{1,j}) \cup \cdots \cup \mathcal{N}(a_{k,j})$, $j = 1..n$, and*
  - *add $f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n})$ to $rhs(\mathcal{X})$, and*
  - *for each j, if $rhs(\mathcal{V}_{N_j}) = \{\}$ then add $a_{1,j} \cap \cdots \cap a_{k,j}$ to $rhs(\mathcal{V}_{N_j})$.*

□

$$\begin{array}{rcl} \mathcal{W} & = & \mathcal{X} \cup f^{-1}(f^2(\mathcal{W})), \\ \mathcal{X} & = & f(\mathcal{Y}) \cap \mathcal{Z}, \\ \mathcal{Y} & = & a, \\ \mathcal{Z} & = & f(a) \cup g(a), \end{array}$$

A. Add $f(\mathcal{W})$ to $rhs(\mathcal{W})$;
B. Add $f(\mathcal{V}_{\{\mathcal{Y},a\}})$ to $rhs(\mathcal{X})$, and set $rhs(\mathcal{V}_{\{\mathcal{Y},a\}})$ to $\{\mathcal{Y} \cap a\}$;
C. Add $f(\mathcal{V}_{\{\mathcal{Y},a\}})$ to $rhs(\mathcal{W})$;
D. Add a to $rhs(\mathcal{V}_{\{\mathcal{Y},a\}})$;

Figure 8.4: **Example Execution of the Reformulated Algorithm**

The reformulated algorithm can now be stated as: exhaustively apply those instances of Transformations 15–18 that change *rhs*, and on termination output the explicit form constraints resulting from the deletion of all intersectors, projectors and variables from *rhs*. The main difference between this algorithm and the intersection-projection algorithm in Chapter 7 is that the op-substitution, projection simplification and intersection simplification transformations (Transformations 1, 3 and 4 respectively) have been combined in an effort to minimize the number of transformation applications performed during the algorithm. The proofs of termination and correctness of the reformulated algorithm are similar to those for the intersection-projection algorithm. In summary,

Theorem 10 *Let $\mathcal{C}$ be a collection of constraints in restricted standard form. Then, the exhaustive application of Transformations 15–18 terminates. Moreover, the resulting constraints $\mathcal{C}'$ are such that $lm(\mathcal{C}) =_{var(\mathcal{C})} lm(explicit(\mathcal{C}'))$.* $\Box$

Figure 8.4 contains an example execution of the reformulated algorithm. When input with the constraints shown below on the left column, the reformulated algorithm performs steps A, B, C and D, corresponding to applications of Transformations 16, 18, 15 and 18 respectively. The final output of the algorithm consists of the constraints $\mathcal{W} = f(\mathcal{W}) \cup f(\mathcal{V}_{\{\mathcal{Y},a\}})$, $\mathcal{X} = f(\mathcal{V}_{\{\mathcal{Y},a\}})$, $\mathcal{Y} = a$, $\mathcal{Z} = f(a) \cup g(a)$, and $\mathcal{V}_{\{\mathcal{Y},a\}} = a$.

8.3 Outline of Implementation

The implementation of the preprocessing phase is fairly straightforward. We shall instead concentrate on the implementation of the core part of the algorithm. We first classify the kinds of expressions that can appear in the sets $rhs(\mathcal{X})$. First, there are non-variable atomic expressions, and we refer to these as *constructed expressions* and denote them by ce. In essence, these are the central objects that are manipulated by the algorithm. Second, there are the projectors, and these can be viewed as operators on constructed expressions. For example, if $rhs(\mathcal{X})$ contains the constructed expression $f(a_1, a_2)$, then $f_{(2)}^{-1}(\mathcal{X})$ produces a_2 (which must either be a constructed expression or a variable). Specifically, corresponding to each projector of the form $f_{(i)}^{-1}(\cdots)$, define a partial function $proj_{f,i}$ as follows

$$proj_{f,i}(ce) \;=\; \begin{cases} a_i & \text{if } ce \text{ is } f(a_1, \ldots, a_n) \\ \top & \text{if } ce \text{ is } \top \\ \textit{undefined} & \text{otherwise} \end{cases}$$

The third class of expressions consists of the intersectors. Again these can be viewed as operators on constructed expressions, and corresponding to each m-ary intersection expression, we define a partial function $intersect_m$ as follows. Let $ce_1, \ldots, ce_m$ be a sequence of constructed expressions such that, for some $f \in \Sigma$, each ce_i has the form $f(\cdots)$ or $\top$, let $a_1, \ldots, a_k$ be the elements of $ce_1, \ldots, ce_m$ of the form $f(\cdots)$, let each a_i be $f(a_{i,1}, \ldots, a_{i,n})$, $i = 1..k$, let N_j is $\mathcal{N}(a_{1,j}) \cup \cdots \cup \mathcal{N}(a_{k,j})$, $j = 1..n$, and define that

$$intersect_m(ce_1, \ldots, ce_m) \;=\; \begin{cases} \top & \text{if } k = 0 \\ f(\mathcal{V}_{N_1}, \ldots, \mathcal{V}_{N_n}) & \text{if } k \geq 1 \end{cases}$$

and that $intersect_m(ce_1, \ldots, ce_m)$ is undefined if the sequence $ce_1, \ldots, ce_m$ is not of the specified form. The final kind of expressions that can occur in rhs are simply variables, and their main effect in the algorithm is to directly transfer constructed expressions from one equation to another.

On the basis of this classification, we represent the array rhs as three separate arrays, con, var and op. Let $con(\mathcal{X})$ denote the set of constructed expressions in $rhs(\mathcal{X})$, let $var(\mathcal{X})$ denote the set of variables in $rhs(\mathcal{X})$ and let $op(\mathcal{X}_i)$ denote the set of projectors and intersectors in $rhs(\mathcal{X})$. For example, the equation $\mathcal{X} = f^2(\mathcal{X}) \cup f(\mathcal{X}) \cup f^{-1}(\mathcal{X}) \cup \mathcal{Y}$ is represented as $con(\mathcal{X}) = \{f^2(\mathcal{X}), f(\mathcal{X})\}$, $op(\mathcal{X}) = \{f^{-1}(\mathcal{X})\}$, and $var(\mathcal{X}) = \{\mathcal{Y}\}$. Note that the preprocessing phase ensures that initially $op(\mathcal{X}_i)$ is either empty or a singleton set, and that the algorithm never alters $op(\mathcal{X}_i)$. Hence we shall treat $op(\mathcal{X}_i)$ as either the empty set, or a projector or an intersector. Corresponding to the convention that $rhs(a)$ is just $\{a\}$ if a is not a set variable, we shall similarly define that $con(a)$ is $\{a\}$ if a is not a set variable. Using this new notation, Transformations 15–18 can be rephrased as:

- If $op(\mathcal{X})$ is $f_i^{-1}(\mathcal{Y})$, $ce \in con(\mathcal{Y})$ and $nonempty(t)$, then if $proj_{f,i}(ce)$ is a variable, add it to $var(\mathcal{X})$, and otherwise add it to $con(\mathcal{X})$.
- If $op(\mathcal{X})$ is $a_1 \cap \cdots \cap a_m$ and $ce_i \in con(a_i)$, $i = 1..m$, then add $intersect_m(ce_1, \ldots, ce_m)$ to $con(\mathcal{X})$, and construct appropriate equations for any new intersection variables introduced.
- If $\mathcal{Y} \in var(\mathcal{X})$ and $ce \in con(\mathcal{Y})$, then add ce to $con(\mathcal{X})$.

where "add" is used informally to mean that if the operation is defined then add the resulting atomic expression to the specified set if it does not already appear there, and $nonempty(ce)$ denotes a function that determines whether ce is non-empty in $lm(explicit(\mathcal{C}))$ (we shall address the implementation of *nonempty* later in this section).

Our implementation is essentially based on the above formulation of transformations. Two key observations about this formulation motivate a number of major implementation design decisions. First, a frequent operation in the algorithm involves comparison of constructed expressions, and in particular, the determination of whether a new constructed expression is already an element of a set. Second, at any particular instance, there may be many possible steps, but only a few of these are likely to be productive. It is therefore important to be able to focus on those steps that are likely to be productive. We now elaborate.

Representation of Constructed Expressions

To provide the cheap comparison for constructed expressions, we code each such expression as an integer in the following simple manner. As each variable is encountered, a unique positive integer is allocated for it. Notationally we shall write $\#\mathcal{X}$ to denote the integer associated with the variable $\mathcal{X}$. Function and constant symbol are also identified by positive integers; we write $\#f$ to denote the integer for the function symbol f. The coding of constructed expressions is essentially performed by incrementally building a mapping ζ from sequences of integers into negative integers (ζ is initially empty). Specifically, the coding $\#ce$ of a constructed expression ce is achieved as follows. Let ce be $f(a_1, \ldots, a_n)$ and first compute the sequence $(\#f, \#a_1, \ldots, \#a_n)$. Then, return $\zeta(\#f, \#a_1, \ldots, \#a_n)$ if it is already defined, and otherwise allocate a new negative integer say j, update ζ appropriately and then return j. An array is also maintained to map each coding into the sequence it represents. This is used particularly by the projection operation.

Coding constructed expressions as integers provides a very compact representation since there is maximal sharing between constructed expressions that have common subexpressions. It also allows the set $con(\mathcal{X})$ to be represented efficiently. Specifically, we represent the relationship $a \in con(\mathcal{X})$ using ordered pairs $(\#a, \#\mathcal{X})$, and this in turn is implemented using a hash table. Similar comments apply to a number of other operations of the algorithm, and hashing techniques are heavily used throughout. Although fairly simple methods have proved effective for moderate sized problems, it is likely that specialized hashing techniques will be important for handling very large analysis problems.

However the use of the coding represents a tradeoff. Whilst operations such as comparison of constructed expressions are dramatically improved, the operation of projection becomes slightly more expensive and also there are overheads in initializing and maintaining the data structures used. Thus far, these costs appear to be insignificant compared to the substantial improvements over a very early implementation using an explicit PROLOG-style term representation.

Dependency Directed Updating

We now address the issue of focusing on productive steps. Note that once a particular instance of a step has been applied, it never needs to be applied again. For example, if $op(\mathcal{X})$ is the projector $f^{-1}_{(1)}(\mathcal{Y})$ and the constructed expression $f(b, c)$ appears in $con(\mathcal{Y})$, then once b has been added to $con(\mathcal{X})$, we never again need to apply $op(\mathcal{X})$ to $f(b, c)$. We exploit this by using a dependency directed updating scheme. In other words we only apply operations to new constructed expressions. Specifically, we define an array dep such that $dep(\mathcal{X})$ is the set consisting of

$$
\begin{aligned}
\mathcal{X} &= b \cup f(\mathcal{X}) \cup \mathcal{Z} \\
\mathcal{Y} &= c \cup (\mathcal{X} \cap \mathcal{Z}) \\
\mathcal{Z} &= d \cup f^{-1}_{(1)}(\mathcal{X})
\end{aligned}
$$

	con	*var*	*op*	*dep*
$\mathcal{X}$	$b, f(\mathcal{X})$	$\mathcal{Z}$		$\cap\_dep(\mathcal{Y}), proj\_dep(\mathcal{Z})$
$\mathcal{Y}$	c		$\mathcal{X} \cap \mathcal{Z}$	
$\mathcal{Z}$	d		$f^{-1}_{(1)}(\mathcal{X})$	$var\_dep(\mathcal{X}), \cap\_dep(\mathcal{Y})$

Figure 8.5: **Example Constraints and Their Representation**

$$
\begin{aligned}
& \{var\_dep(\mathcal{Y}) : \mathcal{X} \in var(\mathcal{Y})\} \\
\bigcup\ & \{proj\_dep(\mathcal{Y}) : f^{-1}_{(i)}(\mathcal{X}) \in op(\mathcal{Y})\} \\
\bigcup\ & \{\cap\_dep(\mathcal{Y}) : (a_1 \cap \cdots \cap a_n) \in op(\mathcal{Y}) \text{ where some } a_i \text{ is } \mathcal{X}\ \}
\end{aligned}
$$

We refer to items of the form $var\_dep(\mathcal{Y})$, $proj\_dep(\mathcal{Y})$ or $\cap\_dep(\mathcal{Y})$ as *dependencies*. The set of dependencies for a variable $\mathcal{X}$ indicates where any new constructed expressions for $\mathcal{X}$ have to be propagated. Figure 8.5 contains an example collection of constraints and its representation in terms of arrays *con*, *var op* and *dep*. Using this representation, the algorithm can now be described in Figure 8.6. The algorithm is initiated by inspecting each variable $\mathcal{X}$ and calling $add(proj_{f,i}(ce), \mathcal{X})$ for each $ce \in rhs(a)$ if $op(\mathcal{X})$ is $f^{-1}_{(i)}(a)$, and calling $add(intersect_n(ce_1, \ldots, ce_n))$ for each sequence $(ce_1, \ldots, ce_n)$ such that $ce_i \in rhs(a_i)$, if $op(\mathcal{X})$ is $a_1 \cap \cdots \cap a_n$.

Intersection Variables and Non-Empty

We conclude by discussing intersection variables and the function *nonempty*. The generation and management of intersection variables is straightforward. Corresponding to the function $\mathcal{N}$, an array is used to record the set of atomic set expressions corresponding to each intersection variable. The mapping from a set of atomic set expressions $\{a_1, \ldots, a_n\}$ into an intersection variable is performed by first constructing $\mathcal{N}(a_1) \cup \cdots \cup \mathcal{N}(a_n)$. The elements of this set are then sorted and the resulting sequence is looked up in a hash table, where each entry of this table consists of a sorted list of atomic expressions and the corresponding intersection variable. If the sequence is found in the

update$(ce, dep, \mathcal{X})$ {
if *dep* is $var\_dep(\mathcal{Y})$ **then** $add(ce, \mathcal{Y})$;
if *dep* is $proj\_dep(\mathcal{Y})$ and $op(\mathcal{Y})$ is $f_{(i)}^{-1}(\mathcal{X})$ **then** $add(proj_{f,i}(ce), \mathcal{Y})$;
if *dep* is $\cap\_dep(\mathcal{Y})$ and $op(\mathcal{Y})$ is $a_1 \cap \cdots \cap a_m$ **then**
  let $\mathcal{X}$ be a_j; /* note that $\mathcal{X}$ must appear in $a_1 \ldots, a_m$ */
  foreach sequence $(ce_1, \ldots, ce_{i-1}, ce, ce_{i+1}, \ldots, ce_n)$
      such that $ce_j \in rhs(a_j)$, $j \neq i$
    $add(intersect_n(ce_1, \ldots, ce_{i-1}, ce, ce_{i+1}, \ldots, ce_n), \mathcal{Y})$;
    construct appropriate equations for any new intersection variables;
}

$add(ce, \mathcal{X})$ {
if $ce \notin con(\mathcal{X})$ **then**
  $con(\mathcal{X}) := con(\mathcal{X}) \cup \{ce\}$;
  foreach $dep' \in dep(\mathcal{X})$
    $update(ce, dep', \mathcal{X})$;
}

Figure 8.6: **Dependency-Based Algorithm**

table, the appropriate intersection variable is returned. If the sequence is not found, then a new variable $\mathcal{V}_N$ is generated and the table is updated appropriately. Also a new equation is generated by setting $var(\mathcal{V}_N) := \{\}$, $con(\mathcal{V}_N) := \emptyset$ and $op(\mathcal{V}_N) := \{a_1 \cap \cdots \cap a_n\}$ where N is $\mathcal{N}(a_1) \cup \cdots \cup \mathcal{N}(a_n)$, and updating the dependencies for each variable $\mathcal{X}$ in $\{a_1, \ldots, a_n\}$ using $dep(\mathcal{X}) := dep(\mathcal{X}) \cup \{op\_dep(\mathcal{V}_N)\}$, and then finally, if each a_i is a non-variable atomic expression, calling $add(intersect_n(a_1, \ldots, a_n), \mathcal{V}_N)$

We now address the issue of the non-empty condition in the projector step. The algorithm used is essentially a variation of the algorithm to determine non-emptiness described in Section 7.1. However the previous algorithm is modified so that it incrementally computes the mapping *nonempty*, instead of recomputing it each time it is needed. As before, let *nonempty* be an array that maps each atomic expression a to a boolean value that is *true* if a is currently known to be non-empty. At the start of the algorithm, each entry in *nonempty* is set to *true* iff a is a ground atomic expression that is non-empty under all interpretations. The value of $nonempty(f(a_1, \ldots, a_n))$ is updated to *true* if $nonempty(a_i)$ is *true* for all $1 \leq i \leq n$. The value of $nonempty(\mathcal{X})$ for a variable $\mathcal{X}$ is updated to *true* if $con(\mathcal{X})$ contains a non-empty atomic expression. This updating is managed using lists of dependencies. For each atomic expression a, we maintain the list of the atomic expressions directly dependent on $nonempty(a)$. When $nonempty(a)$ changes from *false* to *true*, the atomic expressions dependent on a are recomputed. In essence, the computing of *nonempty* is merged into the rest of the algorithm. Hence, whenever we need to determine whether a is currently non-empty, we just inspect $nonempty(a)$; there is no explicit call to the *nonempty* function.

We now address the issue of how to incorporate the non-empty condition in the projector updating step. Recall that the step is: if $op(\mathcal{X})$ is $f^{-1}_{(i)}(\mathcal{Y})$, $ce \in con(\mathcal{Y})$ and $nonempty(ce)$, then add $proj_{f,i}(ce)$ to $con(\mathcal{X})$ or $var(\mathcal{X})$. Now, there is no difficulty if we find that $nonempty(ce)$ is *true*, since we can then proceed with the updating step. However, if $nonempty(ce)$ is *false* then we need to ensure that if $nonempty(ce)$ ever becomes *true*, then the projector updating step is eventually completed. This is achieved by adding a new kind of dependency to the non-empty dependency lists.

8.4 Intersectors

The previous section outlined a rudimentary implementation, which although a significant improvement over a naive implementation, is still impractical. The main reason for this is intersectors, which are perhaps the most problematic part of the implementation. Not only are they responsible for introducing new variables, but updating intersectors has a combinatorial nature. For example, consider an intersector $a_1 \cap \cdots \cap a_n$, and suppose that ce has been freshly added to $con(a_i)$. Then, in updating this intersector using this constructed expression, we must consider all possible combinations of $ce_j \in con(a_j)$, $j \neq i$.

A first step in dealing with this problem is to partition the constructed expressions in $con(\mathcal{X})$ according to their principal function symbol. For example, consider updating the intersector $\mathcal{X} \cap \mathcal{Y} \cap \mathcal{Z}$ using a newly constructed expression $f(\mathcal{Y})$ for $\mathcal{Z}$. If $con(\mathcal{X}) = \{f(\mathcal{X}), f(\mathcal{W}), g(\mathcal{X})\}$ and $con(\mathcal{Y}) = \{f(\mathcal{Z}), g(\mathcal{Y})\}$, then clearly we only need to consider the intersections $f(\mathcal{X}) \cap f(\mathcal{Z}) \cap f(\mathcal{Y})$ and $f(\mathcal{W}) \cap f(\mathcal{Z}) \cap f(\mathcal{Y})$, and we can ignore intersections such as $f(\mathcal{X}) \cap g(\mathcal{Y}) \cap f(\mathcal{Y})$, since they will always be empty. Although this very simple modification is useful, more fundamental changes are required to implement intersection efficiently. In essence we need to exploit the substantial redundancy that is typically present in set constraints.

Minimal DNF Expansion

Consider the intersector $\mathcal{X} \cap \mathcal{Y} \cap \mathcal{Z}$ where $con(\mathcal{X})$, $con(\mathcal{Y})$ and $con(\mathcal{Z})$ are respectively $\{A, B, C\}, \{A, C, D\}$ and $\{A, B, D\}$, where A, B, C and D are distinct constructed expressions. In essence, we wish to compute

$$(A \cup B \cup C) \cap (A \cup C \cup D) \cap (A \cup B \cup D).$$

A naive expansion of this expression would lead to 27 intersections: $A \cap A \cap A$, $A \cap A \cap C$, etc. However for this example, we only need to compute the expressions A, $B \cap D$, $C \cap B$ and $C \cap D$, and so we can reduce 27 intersections to just three. The problem of minimizing the number of expressions that must be computed can be viewed as a special case of computing a minimal DNF form, given an expression in CNF. A number of different approaches were tested. The essence of the current approach is to precompute information about the pattern of occurrences of constructed terms that appear in more than one "disjunction", and then to use this information to sequentially build up minimal conjunctions.

We now outline how this approach to intersection is incorporated in the implementation. It is clear that updating intersectors is very expensive, and should therefore be done as infrequently as possible. This can be achieved by giving precedence to updates involving projectors. Specifically, the updating is split into two phases. The first consists of the exhaustive updating using projectors. The second consists of updating using intersectors. The algorithm then proceeds by repeatedly alternating these two phases. As a side effect, this organization of updating steps also enhances the performance of the DNF minimization algorithm, since by delaying its application, the amount of redundancy increases. The algorithm can now be described as

repeat
  exhaustively apply all possible projector steps;
  recompute each intersector;
until no change;

Approximating Subsumption Relationships

We conclude this section by describing an enhancement the performance of intersector recomputation. This modification is perhaps the most important described so far. Consider an intersector $\mathcal{X} \cap \mathcal{Y}$. If $con(\mathcal{X}) = \{A, B\}$ and $con(\mathcal{Y}) = \{C, D\}$ and in the least model of the equations, $A \subseteq B$ and $C \subseteq D$, then clearly the recomputation of the intersector can be reduced to computing $B \cap D$. However, establishing whether subsumption relationships such as $A \subseteq B$ hold in the least model can only be done, in general, once the least model is known. The approach used here involves approximating the least model using the information contained in the array *con*. Specifically, this array defines an equation for each variable $\mathcal{X}$:

$\mathcal{X} = a_1 \cup \cdots \cup a_n$, where $con(\mathcal{X}) = \{a_1, \ldots, a_n\}$.

Hence *con* can be considered to define a collection of equations in explicit form, and this in turn defines an interpretation, call it $\mathcal{I}_{con}$. This interpretation represents the part of the least model that has been computed by the algorithm so far. If $\mathcal{I}_{lm}$ denotes the least model of the constraints input to the algorithm, then $\mathcal{I}_{con}(\mathcal{X}) \subseteq \mathcal{I}_{lm}(\mathcal{X})$ for all variables $\mathcal{X}$ appearing in the input constraints.

Importantly, $\mathcal{I}_{con}$ provides a useful approximate of the subsumption relationships that hold in $\mathcal{I}_{lm}$, and can be used to simplify the intersector computation as follows. First, we use $\mathcal{I}_{lm}$ to compute the non-redundant components of the sets $con(\mathcal{X})$. Let S be a set of constructed expressions, let $\mathcal{I}$ be an interpretation and define that ce is a *maximal element* of S *with respect to* $\mathcal{I}$ if $ce \in S$ and for each $se' \in S$, $\mathcal{I}(se') \not\supset \mathcal{I}(se)$. In other words, there is no expression in S that is strictly larger than se. Then, given the sets $con(\mathcal{X})$, define for each $\mathcal{X}$ the set of maximal constructed expressions as follows:

$$max\_con(\mathcal{X}) \stackrel{\text{def}}{=} \{ce \in con(\mathcal{X}) : ce \text{ is maximal wrt } \mathcal{I}_{con}\} \tag{8.39}$$

If $con(\mathcal{X})$ contains several maximal elements that are equal under $\mathcal{I}_{con}$, then we put one of them in $max\_con(\mathcal{X})$ and discard the others. Finally, the updating of each intersector is carried out using the constructed expressions in max_con instead of con.

In general, there is no formal connection between the subsumption relationships that hold in $\mathcal{I}_{lm}$ and $\mathcal{I}_{con}$, and the sets $max\_con(\mathcal{X})$ are essentially arbitrary subsets of $con(\mathcal{X})$. However, in practice the correlation between the subsumption relationships that hold in $\mathcal{I}_{lm}$ and $\mathcal{I}_{con}$ is close, especially after the early stages of the algorithm. In essence this is because $\mathcal{I}_{con}$ increases quickly in the early stages of the algorithm, and for most of the algorithm's execution it closely approximates $\mathcal{I}_{lm}$. We shall provide some empirical evidence to support this claim in the next section.

We now briefly outline the correctness of this modified intersector recomputation. Since the modified intersector recomputation serves to reduce the number of constructed expressions that would otherwise be generated, the main change in the proof from the basic intersection-projection algorithm relates to completeness. In essence, the key to the proof involves showing that the modified algorithm produces sufficient constructed expressions so that $\mathcal{I}_{con}$ is a model of the constraints. Modifications to the previous completeness proof are needed when the definitions of transformations are used to argue that, on termination, certain constraints are present. The main observation required is that when max_con is computed from con using (8.39), the explicit form constraints described by max_con, call them $\mathcal{C}_{max\_con}$, are such that $lm(\mathcal{C}_{max\_con}) = \mathcal{I}_{con}$.

Additional Improvements

The preceding sections contain only the major design decisions of our implementation. However, there are number of minor modifications which, in sum, have an important impact on the systems performance. We now very briefly summarize a number of these. During the execution of the algorithm, many variables have the same *con* sets. Typically the number of distinct sets $con(\mathcal{X})$, where $\mathcal{X}$ ranges over all set variables, is approximately a third of the total number of set variables. Hence a number of operations only need to be computed once for each distinct set $con(\mathcal{X})$, and this can lead to important savings, particularly in the subsumption component of the algorithm. The computation of intersectors is another place where many improvements can be made. First, it is worthwhile to store the sequences of constructed expressions that have been considered at each intersector. Then, when recomputing an intersector, any sequence that has been generated previously can be ignored. Second, an intersector only needs to be recomputed if the variables on which it depends have changed since the last time it was recomputed. By keeping track of the changes to these variables, the number of intersector recomputations can be reduced by up to 50%.

8.5 Empirics

The implementation described in this paper has been developed over a period of three years. A very early version used an explicit representation for constructed expressions. However the comparison of constructed expressions was prohibitively expensive and only very small programs could be analyzed. The next version employed integer term representation as well as an early form of the modifications for intersector simplification described in the previous section. Most of the basic notions described in this paper were contained in this version. The current version re-implements these notions more efficiently and makes greater use of dependency directed updating. The subsumption algorithm was completely redesigned using bit vectors. The system consists of approximately 4,500 lines of Standard ML (this includes programs to construct the set constraints corresponding to a

	V1 (Nov 89)		V2 (Nov 90)		V3 (Nov 91)	
	time	eqns	time	eqns	time	eqns
nrev-bu	0	18	0.03	14	0.01	10
nrev-td	3.7	50	0.48	48	0.21	35
imp-bu	280	400	4.7	233	0.9	128
imp-td	??	??	21	459	5.3	372
dnf-bu	>2600	>511	2.1	142	0.65	96
dnf-td	??	??	210	1002	12.59	440

Table 8.1: **Impact of Design Decisions**

program as well as the set constraint solver and its front end).

Table 8.5 compares these three versions of the algorithm. The benchmarks used in this table are based on three small logic programs. The first is the standard naive reverse program for reversing lists program (which consists of four rules, two of them facts). The second is an interpreter for a simple imperative programming language similar to the language outlined in Chapter 3. This program consists of 55 rules, 25 of them facts. The third program is a program to convert a propositional logic formula into a formula in disjunctive normal form. It contains 32 rules (10 facts and 22 non-facts with an average of about 2 body atoms per non-fact rule) and contains a large number of mutually recursive calls. For all of these program, we consider the set constraints corresponding to both bottom-up and top-down left-to-right execution. For each version of the algorithm and each benchmark, we give the time taken to run the benchmark and the number of equations generated. All timings are on a Sun Sparc 1+ (24MB) running Mach and using version 0.75 of Standard ML of New Jersey. Note that for the larger examples, the difference between the first and third implementations is in excess of one thousand. In fact timings for a number of benchmarks could not be obtained using the first implementation.

We now give more timings for the third implementation, with particular emphasis on the design decisions outlined in the previous sections. Table 8.5 presents a breakdown of the time and space behavior of the algorithm. Two additional benchmarks are used in this table. The constraints for these benchmarks are given in Figure 8.7, where $f^{11}(\mathcal{X})$ denotes 11 applications of f to $\mathcal{X}$ and $f^{-33}(\mathcal{X})$ denotes 33 applications of the projector $f^{-1}_{(1)}$ to $\mathcal{X}$. These are essentially pathological cases designed for testing the robustness of the intersector and projector parts of the algorithm respectively. Neither

	time				space			
	total (secs)	proj (%)	sub (%)	$\cap$ (%)	eqns	$\cap$	con exps	model size
nrev-bu	0.01	0	100	0	10+0	0+0	11	10
nrev-td	0.21	10	76	14	20+15	8+5	20	26
imp-bu	0.9	8	85	7	120+138	13+12	119	473
imp-td	5.3	8	80	12	246+126	246+126	231	1610
dnf-bu	0.65	29	68	3	88+8	3+4	82	159
dnf-td	12.59	6	59	35	156+284	53+254	313	1739
lcm	13.86	0	99	1	3+142	3+142	171	148
hcf	27.54	1	99	0	1+33	0+0	148	4

Table 8.2: **Time and Space Measurements for V3**

lcm	hcf
$\mathcal{X} = a \cup f^{11}(\mathcal{X})$ $\mathcal{Y} = a \cup f^{13}(\mathcal{Y})$ $\mathcal{Z} = \mathcal{X} \cap \mathcal{Y}$	$\mathcal{X} \ = \ a \cup f^{-33}(\mathcal{X}) \cup f^{144}(\mathcal{X})$

Figure 8.7: **The lcm and hcf Benchmarks**

benefits form the subsumption component of the algorithm.

For each benchmark, Table 8.5 gives timing information, equation counts and an indication of the space used. The first column of the table is total time, and the next three columns break this down into time spent on projectors, inferring subsumption relationships and intersectors, each expressed as a percentage of total time. Column 5 gives the total number of equations in the form $x + y$ where x is the number of equations in the benchmark, and y is the number of additional equations generated during execution. Column 6 give the number of equations whose right hand side contains an intersector. Again entries are of the form $x + y$ where x is the number of intersectors in the input equations, and y is the number of intersectors introduced during execution. Columns 7 is the total number of distinct constructed expressions. Column 8 provides a measure of the complexity of the least model constructed by the algorithm, and is obtained as follows. First, list all of the sets $max\_con(\mathcal{X})$, for each set variable $\mathcal{X}$. Some of these sets will be the same (in which case the corresponding variables are equal in the least model). The complexity measure is then obtained by summing

	subsumption				no subsumption			
	time (secs)	total eqns	con exps	table size	time (secs)	total eqns	con exps	table size
nrev-bu	0.01	10	11	16	0	10	11	16
nrev-td	0.21	35	20	75	0.11	35	20	75
dnf-bu	0.65	96	82	1689	1.01	184	126	5309
dnf-td	12.59	440	313	9317	≈ 218	2669	1510	280K
lcm	13.86	145	171	292	0.26	145	171	292
hcf	27.54	34	148	1644	0.31	34	148	1644

Table 8.3: **Effects of the Subsumption Modification**

the cardinalities of the distinct sets that appear in this list. These results show that bottom-up constraints are substantially easier to solve than top-down left-to-right constraints. This general relationship holds because, by their nature, top-down constraints are more complicated and more accurate than the bottom-up constraints. The subsumption approximation part of the algorithm accounts for the majority of execution time. The lcm and hcf examples show extreme behavior. In the case of lcm, the output for $\mathcal{X}$ is $\mathcal{X} = a \cup f^{143}(\mathcal{X})$, and for hcf the output is $\mathcal{X} = a \cup f^{3}(\mathcal{X})$.

The driving example used during the development of this implementation was the top-down left-to-right constraints for the dnf program. A number of features of this program lead to top-down constraints that are unusually expensive to solve. In particular, it was this program that motivated the subsumption approximation part of the implementation. Table 8.5 illustrates the effects of the subsumption component of the algorithm on all of the example constraints. For the analysis of nrev, and for the lcm and hcf constraints, there is little redundancy and the subsumption component is expensive and provides no direct benefit. However for the more substantial dnf program, it is clearly of benefit, and in the case of top-down analysis, it is crucial. Specifically, in this case the difference in time is nearly a factor of 20, and for one measure of memory usage (the number of entries in the central hash table of the implementation), the difference is a factor of 30. (In fact the measurements for this entry in the table had to be made on a 64MB DECstation 5000, and then converted to equivalent Sparc 1+ times).

Table 8.5 illustrates the dynamic behavior of the algorithm by giving cumulative measures of the consumption of resources during execution of the top-down dnf constraints. One "iteration" corresponds to an exhaustive

	Iterations								
	1	2	3	4	5	6	7	8	9
% time	2	8	18	31	47	61	81	91	100
% $\cap$ eqns	19	24	40	67	85	96	100	100	100
% model	23	43	58	68	78	86	93	100	100

Table 8.4: **Cumulative Consumption of Resources During Execution.**

application of the projector transformation followed by a recomputation of each intersector. The measure of the amount of the least model computed at each stage is obtained as follows. Let $max\_con(\mathcal{X})|_{final}$ denote the value of the set $max\_con(\mathcal{X})$ on termination of the algorithm. Let $\delta(\mathcal{X})$ denote the proportion of the elements in $max\_con(\mathcal{X})|_{final}$ that currently appear in $con(\mathcal{X})$, and this represents a pessimistic estimate of the proportion of $\mathcal{X}$ that has been currently computed. Finally, the measure of the amount of the least model computed is just the average of $\delta(\mathcal{X})$ over all variables $\mathcal{X}$ currently appearing in the equations. The behavior of this measure indicates that the model grows quickly in the early stages, and that most of the time is spent obtaining the last few components of the model. This supports the notion that during most of the execution of the algorithm, $max\_con(\mathcal{X})$ provides a fairly good approximation to the least model for the purposes of obtaining subsumption relationships.

To illustrate the effects of the minimal dnf expansions component of intersector simplification, we again use the top-down dnf constraints. When applied to these constraints, the implementation constructs 1270 intersections. In comparison, a simple expansion of these formulas would have led to 1921 intersections. Although this only represents a direct saving of 649 intersections, or about a third, the indirect savings, in terms of intersection variables that do not need to be introduced, is more significant.

In summary then, we have worked with a moderate sized program (dnf) that exhibits properties that are problematic for analysis, such as substantial mutual recursion and intersection. Very significant progress has been made and this example can now be analyzed within a reasonable time and space bound. While this provides strong evidence that set based analysis can be made practical, much work remains. Our experience suggests that a number of components of the algorithm can be further improved. Currently the major expense is the subsumption algorithm, and one reason for this is that it is not by nature incremental. We are investigating ways of overcoming this, including trading some of its effectiveness for the ability to reuse results from previous iterations. Another avenue for improvement lies in the use of specialized hashing techniques.

Thus far we have focussed extensively on efficiency aspects of the analysis. However we have now reached a stage where moderate sized programs can be analyzed. We therefore plan to use this implementation to investigate set based analysis, with particular emphasis on the quality of the information obtained[1] and its relevance to compilation. Practical program analysis must also deal with operations such as call, assert and retract. Preliminary work suggests that it may be possible to directly modeled these operations in a set constraint framework.

We conclude with a discussion of related work. Most of the work on implementation of analysis for logic programs is based on abstract interpretation and the algorithms used are fundamentally different from those for solving set constraints. Our work is more closely connected to work on types for logic programs in which types are defined by ignoring inter-variable or inter-argument dependencies. Implementations have been reported in [55] and [68], but the former does not focus on practical issues, and the latter is not directly comparable to our work since it deals with type checking rather than type inference. The most closely related work to ours is [4], which describes an implementation of type inference for the functional language FL. In very general terms their observations about the complexity of the intersection operation are similar to ours. However, the two algorithms are completely different in nature. In particular their algorithm does not include the projection operation, and this appears to substantially alter the tradeoffs and design decisions that are made. Furthermore, our implementation has been specifically designed to deal with constraints where there is substantial mutual dependency between variables. Such constraints are typical in the constraints generated for the top-down analysis of logic programs.

[1] As an example of the output of the algorithm, the top-down analysis of the dnf program constructs the following set $\mathcal{X}$ to describe the set of terms that are the result of putting an arbitrary input formula into dnf: $\mathcal{X} = or(\mathcal{X}, \mathcal{X}) \cup and(\mathcal{X}, \mathcal{X}) \cup not(\mathcal{C})$ where $\mathcal{C}$ describes the set of propositional constants.

Part III

Extensions and Future Work

The developments of Parts I and II have focussed on approximating the run-time values of program variables in imperative and logic programs. The main reason for this focus was to provide a concrete setting for the formalization of set based analysis. However, the underlying ideas of set based analysis are by no means restricted to this kind of analysis. We now show how they can be extended to capture information other than variable values, reason about inter-variable dependencies, and analyze functional languages. The style of presentation for this part shall emphasize examples and intuition rather than formal details.

We begin by outlining how set constraints may be used for computing program properties such as modes (in logic programming) and structure sharing. We then address the issue of inter-variable dependencies. It is clear that many kinds of analysis benefit from information about inter-variable dependencies, and in this respect set based analysis is deficient. We therefore develop an approach to program analysis that combines the accuracy of reasoning about program structures inherent in set based analysis, with the ability of abstract interpretation to reason about inter-variable dependencies. This is carried out in the context of logic programs, and leads to a hybrid logic program analysis engine that is more accurate than either abstract interpretation or set constraints. We conclude by outlining how set based analysis may be applied to functional languages. We focus on Standard ML, and illustrate connections with type systems, particularly those based on simple subtypes.

Chapter 9

Modes and Structure Sharing

The core part of this thesis focuses on the possible run-time values of program variable. However, the basic process of constructing set constraints from a program and then solving these set constraints preserves numerous structural properties of a program. It is therefore possible, with only minor modifications to the set constraint algorithm, to compute approximations to a variety of other program properties. We illustrate this with two kinds of analysis. First, we outline the computation of instantiation levels of variables during execution of a logic program (mode analysis). Second, we describe the use of set based analysis to obtain information about the sharing of structures between program variables. This chapter provides only an informal description of what is involved for these two extensions. Its purpose is to demonstrate that the techniques of set based analysis are not tied to a specific kind of analysis, just as they are not tied to a specific operational model.

3. $\leftarrow p(X)^1,\ q(X)^2.$
4 $p(f(b,Y)).$
5. $q(f(Z,c)).$

$$
\begin{array}{rcl}
\mathcal{X}^1 & = & \top \\
\mathcal{X}^2 & = & p_{(1)}^{-1}(Ret_p) \\
\mathcal{X}^3 & = & p_{(1)}^{-1}(Ret_p) \cap q_{(1)}^{-1}(Ret_q) \\
\mathcal{Y}^4 & = & f_{(2)}^{-1}(p_{(1)}^{-1}(Call_p)) \\
\mathcal{Z}^5 & = & f_{(1)}^{-1}(p_{(1)}^{-1}(Call_p)) \\
Call_p & = & p(\mathcal{X}^1) \\
Call_q & = & p(\mathcal{X}^2) \\
Ret_p & = & p(f(b,\mathcal{Y}^4)) \\
Call_p & = & q(f(\mathcal{Z}^5,c))
\end{array}
$$

Figure 9.1: **Mode Analysis Example**

9.1 Mode Analysis

Mode analysis involves determining information about the instantiation of variables during the execution of a program. For example, at some point during program execution, a variable X could be uninstantiated (the values of X are not constrained), instantiated to some fixed value (there is only one possible value for X), or else partially determined (some of the structure of X is fixed, but the value of X is not uniquely determined). In the first case X is said to be *free*, in the second case X is said to be *ground*, and in the third case X is said to be *partially instantiated*. Consider the (labeled) program in Figure 9.1, and suppose that the goal $\leftarrow p(X),\ q(X)$ is executed in a left-to-right manner. When $p(X)$ is selected from the goal, X is free. When $q(X)$ is selected, X is partially instantiated. Finally, when the goal has completed execution, X is ground. Note how the instantiation of X changes as the goal executes.

To illustrate the use of set constraints to compute mode information, we shall focus on groundness information. That is, we shall seek to infer when a program variable is ground. Throughout this chapter, we shall use set constraints of the form $\mathcal{X}_1 = se_1, \ldots, \mathcal{X}_n = se_n$ where $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are distinct set variables, and each se_i is constructed from union, function symbols, set variables and the intersection and projection operators. Recall that in Chapter 8 we outlined how such constraints could be obtained from a program (see Section 8.1, page 259). Such constraints are more compact but slightly less accurate than $\mathcal{SC}_P$. Consider the program and its constraints shown in Figure 9.1. The output of the set constraint algorithm on these constraints is:

$$\begin{array}{lcl} \mathcal{X}^1 & \supseteq & \top \\ \mathcal{X}^2 & \supseteq & f(b, \mathcal{Y}^4) \\ \mathcal{X}^3 & \supseteq & f(b, c) \\ \mathcal{Y}^4 & \supseteq & \top \\ \mathcal{Z}^5 & \supseteq & \top \end{array}$$

Now, if we interpret $\top$ as the set of *all* terms (not just ground terms), then the least model of the above constraints is

$$\begin{array}{lcl} \mathcal{X}^1 & \mapsto & \{\text{all terms}\} \\ \mathcal{X}^2 & \mapsto & \{f(b, s) : \ s \text{ is an arbitrary term}\} \\ \mathcal{X}^3 & \mapsto & \{f(b, c)\} \\ \mathcal{Y}^4 & \mapsto & \{\text{all terms}\} \\ \mathcal{Z}^5 & \mapsto & \{\text{all terms}\} \end{array}$$

and this correctly describes information about the groundness of $\mathcal{X}$ during program execution. This process can be formalized by re-interpreting set constraints so that they map into arbitrary sets of terms, rather than sets of ground terms. The details are considerable and are beyond the scope of the thesis. We instead give some intuition and some further examples.

The underlying reason why set constraints can be used to obtain mode information is that they correspond closely to the operational behavior of a program. Broadly speaking, union is used to model rule choices, projections are used to model the matching of one atom against another, intersection is used to model unification, and top corresponds to an uninstantiated variable. Moreover, the set constraint algorithm preserves this correspondence. In particular, the simplification of the intersection operator (conservatively) models the behavior of intersection. For example, the set expression $f(b) \cap \top$ is essentially simplified into $f(b)$, corresponding to the unification of $f(b)$ with an uninstantiated variable. Note that during the algorithm $\top$ cannot be (safely) interpreted as an uninstantiated variable. Instead it must be interpreted as an unknown structure. This is essentially because aliasing effects are ignored.

Consider the append program in Figure 9.2. When such a program is analyzed, we typically wish to infer properties such as: what are the possible "modes" of calls to append, given that it is called with first and second

2. $\leftarrow app(\star, \star, \top)^1$.
3. $app(nil, W, W)$.
5. $app(cons(X, L), Y, cons(X, Z)) \leftarrow app(L, Y, Z)^4$.

$$\begin{array}{rcl}
\mathcal{V}^1 & = & \top \\
\mathcal{V}^2 & = & app_{(3)}^{-1}(Ret_{app}) \\
\mathcal{W}^3 & = & app_{(2)}^{-1}(Call_{app}) \cap app_{(3)}^{-1}(Call_{app}) \\
\mathcal{X}^4 & = & cons_{(1)}^{-1}(app_{(1)}^{-1}(Call_{app})) \cap cons_{(1)}^{-1}(app_{(3)}^{-1}(Call_{app})) \\
\mathcal{L}^4 & = & cons_{(2)}^{-1}(app_{(1)}^{-1}(Call_{app})) \\
\mathcal{Y}^4 & = & app_{(2)}^{-1}(Call_{app}) \\
\mathcal{Z}^4 & = & cons_{(2)}^{-1}(app_{(3)}^{-1}(Call_{app})) \\
\mathcal{X}^5 & = & cons_{(1)}^{-1}(app_{(1)}^{-1}(Call_{app})) \cap cons_{(1)}^{-1}(app_{(3)}^{-1}(Call_{app})) \\
\mathcal{L}^5 & = & cons_{(2)}^{-1}(app_{(1)}^{-1}(Call_{app})) \cap app_{(1)}^{-1}(Ret_{app}) \\
\mathcal{Y}^5 & = & app_{(2)}^{-1}(Call_{app}) \cap app_{(2)}^{-1}(Ret_{app}) \\
\mathcal{Z}^5 & = & cons_{(2)}^{-1}(app_{(3)}^{-1}(Call_{app})) \cap app_{(3)}^{-1}(Ret_{app}) \\
Ret_{app} & = & app(nil, \mathcal{W}^3, \mathcal{W}^3) \cup app(cons(\mathcal{X}^5, \mathcal{L}^5), \mathcal{Y}^5, cons(\mathcal{X}^5, \mathcal{Z}^5)) \\
Call_{app} & = & app(cons(b, nil), cons(c, nil), \mathcal{V}^1) \cup app(\mathcal{L}^4, \mathcal{Y}^4, \mathcal{Z}^4)
\end{array}$$

Figure 9.2: **The Append Program and Its Top-Down Constraints**

arguments ground. To express this compactly, it is convenient to introduce a new constant $\star$ to denote the set of ground terms. That is, $\top$ denotes all terms and $\star$ denotes just ground terms. Corresponding to this new constant, we have some new simplification rules:

$$f_{(i)}^{-1}(\star) = \star$$
$$\star \cap \top = \star$$
$$\star \cap f(s_1, \ldots, s_n) = (\star \cap s_1, \ldots, \star \cap s_n)$$

Note that $\star \cap t$ cannot just be simplified into t since t may contain non-ground terms, but $\star \cap t$ is always ground. For example, $\star \cap f(\top)$ is not the same as $f(\top)$.

The implementation described in Chapter 8 has been extended to provide mode information using the methods just outlined. Although it adds a number of special cases, particular during the computation of intersectors, the overhead of these modifications appears to be negligible. In fact a number of the benchmarks used in Chapter 8 employ an initial goal involving $\star$ and $\top$. For example, the top-down benchmarks involving the naive reverse program, the imperative language interpreter and the dnf program used the initial goals $\leftarrow nrev(\star, \top)$, $\leftarrow eval(\star, \top)$ and $\leftarrow dnf(\star, \top)$ respectively. In each case, the modes computed are the obvious ones (for example, all calls to nrev have first argument ground, and all calls to append have first and second arguments ground).

To give some intuition about the accuracy of the mode information generated, consider a simple abstract interpretation algorithm that uses the approximate values *any* and *ground* and represents substitutions as mappings from program variables into $\{any, ground\}$. The information computed by the set constraint approach will be strictly more accurate than this simple abstract interpretation algorithm. In particular, it will perform better on programs involving structural information. For example, the set constraint approach determines that after execution of the goal $\leftarrow p(f(X, b))$ in the first program of Figure 9.3, X will be ground, whereas the abstract interpretation algorithm will be unable to determine this fact.

However, the set constraint approach ignores inter-variable dependencies, and these are clearly important for mode analysis. For example, in the second program of Figure 9.3, the set constraint approach is not able to determine that after execution of $\leftarrow eq(X, Y), r(Y)$, the variable X is ground.

$\leftarrow p(f(X,b))$.	$\leftarrow eq(X,Y), r(Y)$.	$\leftarrow p(Y), r(Y,X)$.
$p(U) \leftarrow q(U)$.	$eq(U,U)$.	$p(U) \leftarrow q(U)$.
$q(f(a,V))$.	$r(a)$.	$q(f(a,V))$.
		$r(f(W1,W2),W1)$.

Figure 9.3: **Three Programs Illustrating Accuracy of Mode Analysis**

$\leftarrow p(X,Y)$.	$\leftarrow p(X,Y)$.
$p(Z,Z) \leftarrow q(Z)$.	$p(f(b),f(b))$.
$q(f(b))$.	

Figure 9.4: **Sharing of Structures Between Variables**

In contrast, abstract interpretation approaches have been developed to capture information about the aliasing and sharing behavior of variables; see [15, 27, 44, 51, 59] for example. These *dependency-based* approaches infer that both X and Y are ground for this program. They are, however, rather weak in reasoning about partially instantiated structures. For example, in the first program, the dependency-based approaches cannot infer that X is ground since this requires reasoning about the structure of the terms to which U can be bound.

In summary, the set constraint approach is more accurate on some programs because of its superior ability to reason about term structure, and the abstract interpretation approach is more accurate on some programs because of its ability to capture information about inter-variable dependencies. Note that neither approach is able to determine that X is ground after the execution of the goal $\leftarrow p(Y), r(Y,X)$ in the third program of Figure 9.3. This motivates the development of hybrid approaches that incorporate set based techniques (for reasoning about term structure) and abstract interpretation techniques (for reasoning about dependencies). We shall consider such an approach in Chapter 10.

9.2 Sharing Analysis

We now outline how set constraints can be used to obtain information about the sharing of structures between program variables. Consider the two programs in Figure 9.4. While the declarative semantics of both programs is the same (the answer substitution for $\leftarrow p(X,Y)$ is $[X \mapsto f(b), Y \mapsto f(b)]$ in both cases), there are important differences between these programs at the implementation level. In particular, after execution of the goal in the first program, it is usually the case that X and Y are bound to the same "copy" of the structure $f(b)$. In contrast, after execution of the goal in the second program, X and Y are typically bound to different "copies" of the structure $f(b)$.

Such differences become important when one considers optimizations involving reuse of structures. To illustrate such optimizations, consider the following rule

$$p(X,Y,L) \leftarrow q(X,LX), \quad r(Y,LY), \quad app(LX,LY,L).$$

where *app* denotes the standard list append predicate (which appears in Figure 9.2). Suppose that it has already been determined that whenever this rule is called, X and Y are both ground and do not contain any occurrences of the list constructor *cons*, and L is free. Now, the rule executes by calling q and r, which generate lists LX and LY from the input values of X and Y, and then combining the results. If we can determine after execution of $q(X,LX)$ and $r(Y,LY)$ that LX and LY are both ground and do not share any *cons* symbols, then the operation of appending LX and LY to give L can be significantly improved by modifying the execution of *app* so that the structure of LX is reused. In effect, this modified append operation destructively updates the end of the list LX to point to the list LY.

We now show how set constraints can be adopted to compute the kind of sharing information needed for structure reuse optimizations. The set constraints corresponding to the two programs in Figure 9.4 are given in Figure 9.5. After substituting the definitions of $Call_p$, Ret_p and performing some obvious simplifications of the projection symbols, these constraints become:

$$\begin{array}{ll}
\mathcal{X}^1 = \top & \\
\mathcal{Y}^1 = \top & \mathcal{X}^1 = \top \\
\mathcal{X}^2 = \mathcal{Z}^4 & \mathcal{Y}^1 = \top \\
\mathcal{Y}^2 = \mathcal{Z}^4 & \mathcal{X}^2 = f(b) \\
\mathcal{Z}^3 = \top & \mathcal{Y}^2 = f(b) \\
\mathcal{Z}^4 = \top \cap \top \cap f(b) &
\end{array}$$

2. $\leftarrow p(X,Y)^1$.
4. $p(Z,Z) \leftarrow q(Z)^3$.
5. $q(f(b))$.

2. $\leftarrow p(X,Y)^1$.
3. $p(f(b), f(b))$.

$$\begin{aligned}
\mathcal{X}^1 &= \top \\
\mathcal{Y}^1 &= \top \\
\mathcal{X}^2 &= p^{-1}_{(1)}(Ret_p) \\
\mathcal{Y}^2 &= p^{-1}_{(2)}(Ret_p) \\
\mathcal{Z}^3 &= p^{-1}_{(1)}(Call_p) \cap p^{-1}_{(2)}(Call_p) \\
\mathcal{Z}^4 &= p^{-1}_{(1)}(Call_p) \cap p^{-1}_{(2)}(Call_p) \cap q^{-1}_{(1)}(Ret_q) \\
Call_p &= p(\mathcal{X}^1, \mathcal{Y}^1) \\
Call_q &= q(\mathcal{Z}^3) \\
Ret_p &= p(\mathcal{Z}^4, \mathcal{Z}^4) \\
Ret_q &= q(f(b))
\end{aligned}$$

$$\begin{aligned}
\mathcal{X}^1 &= \top \\
\mathcal{Y}^1 &= \top \\
\mathcal{X}^2 &= p^{-1}_{(1)}(Ret_p) \\
\mathcal{Y}^2 &= p^{-1}_{(2)}(Ret_p) \\
Call_p &= p(\mathcal{X}^1, \mathcal{Y}^1) \\
Ret_p &= p(f(b), f(b))
\end{aligned}$$

Figure 9.5: **Set Constraints for Programs in Figure 9.4**

Consider giving a unique label to each occurrence of a function symbol in the set constraints constructed from a program. Using labels ①,② etc., the above constraints become

$$\begin{aligned}
\mathcal{X}^1 &= \top \\
\mathcal{Y}^1 &= \top \\
\mathcal{X}^2 &= \mathcal{Z}^4 \\
\mathcal{Y}^2 &= \mathcal{Z}^4 \\
\mathcal{Z}^3 &= \top \cap \top \\
\mathcal{Z}^4 &= \top \cap \top \cap f^{①}(b^{②})
\end{aligned}$$

$$\begin{aligned}
\mathcal{X}^1 &= \top \\
\mathcal{Y}^1 &= \top \\
\mathcal{X}^2 &= f^{①}(b^{②}) \\
\mathcal{Y}^2 &= f^{③}(b^{④})
\end{aligned}$$

Now, the definition of interpretation of set constraints can be extended to deal with labeled constraints as follows. First define that a *labeled term* is a term in which some of the function symbols are labeled. Now define that a *labeled interpretation* is a mapping from set variables into labeled terms. Such an interpretation is extended to map from *labeled* set expressions into sets of labeled terms. For example $\mathcal{I}(se_1 \cup se_2) = \mathcal{I}(se_1) \cup \mathcal{I}(se_2)$, $\mathcal{I}(f^{-1}_{(i)}(se)) = \{s_i : f(s_1, \ldots, s_n) \in \mathcal{I}(se) \text{ or } f^{\lambda}(s_1, \ldots, s_n) \in \mathcal{I}(se)\}$ where λ ranges over labels, and $\mathcal{I}(\top)$ is the set of all unlabeled terms. Models and least models are defined in the expected way. For example, in the least models of the above constraints, the sets for $\mathcal{X}_2$ and $\mathcal{Y}_2$ are respectively:

$$\begin{array}{ll} \mathcal{X}^2 = \{f^{①}(b^{②})\} & \qquad \mathcal{X}^2 = \{f^{①}(b^{②})\} \\ \mathcal{Y}^2 = \{f^{①}(b^{②})\} & \qquad \mathcal{Y}^2 = \{f^{③}(b^{④})\} \end{array}$$

Now, in the least model of the first constraints, the sets for $\mathcal{X}^2$ and $\mathcal{Y}_2$ contain terms with common labels, and this indicates the possibility of sharing of structures between these two variables. In contrast, in the least model of the second constraints, the sets for $\mathcal{X}^2$ and $\mathcal{Y}_2$ do not contain terms with common labels, and this reflects the fact that there can be no sharing of structures between these two variables.

Note that we have defined $\mathcal{I}(\top)$ to be the set of all unlabeled terms. Otherwise we could not obtain any useful information from the least model since it would indicate possible sharing between $\mathcal{X}^1$ and $\mathcal{Y}^1$. Thus far we have not considered how intersection is interpreted, and this represents the most difficult part of the extension. Intuitively, intersection operations corresponds to unification operations during program execution. Now, suppose that a variable X is bound to the term $f^{①}(s_1)$ and consider unifying this with $f^{②}(s_2)$. Although this unification step may change the structure of s_1, it cannot change the binding of X. That is, after unification, X is still bound to an occurrence of f with label ①. To reflect this, we first define a partial function $\wedge$ for combining labeled terms such that the leftmost label is preserved. Specifically, let $t_1, \ldots, t_m$ be labeled terms such that, for some function symbol f, each t_i is either of the form $f^{\lambda_i}(t_{i,1}, \ldots, t_{i,n})$ or $f(t_{i,1}, \ldots, t_{i,n})$. Define that $t_1 \wedge \cdots \wedge t_m$ is $f^{\lambda_j}(s_1, \ldots, s_m)$ where $j \geq 1$ is the smallest index such that t_j is of the form $f^{\lambda_i}(t_{i,1}, \ldots, t_{i,n})$, and each $s_k = t_{1,k} \wedge \cdots \wedge t_{m,k}$, $k = 1..n$. If j does not exist, then $t_1 \wedge \cdots \wedge t_m$ is $f(s_1, \ldots, s_m)$. Finally, $\mathcal{I}(se_1 \cap \cdots \cap se_n)$ can be defined as follows:

$$\{s_1 \wedge \cdots \wedge s_m : s_1 \in \mathcal{I}(se_1), \ldots, s_m \in \mathcal{I}(se_m)\}.$$

This means that $\cap$ is no longer a commutative operation, and reflects the complexity of reasoning about implementation behavior.

Corresponding to the changes in the interpretation of labeled set constraints, the transformations of the set constraint algorithm are appropriately modified. For example, the transformation for simplifying projections ignores labels, and simplifies both $X \supseteq f^{-1}_{(i)}(f^{\lambda}(s_1, \ldots, s_n))$ and $X \supseteq f^{-1}_{(i)}(f(s_1, \ldots, s_n))$ into $\mathcal{X} \supseteq s_i$. The transformation for simplifying intersection preserves the leftmost label. For example, $\mathcal{X} \supseteq \top \cap f^{\lambda_1}(s) \cap f^{\lambda_2}(t)$ is simplified into $\mathcal{X} \supseteq f^{\lambda_1}(\mathcal{V}_N)$ where $N = \mathcal{N}(s) \cup \mathcal{N}(t)$.

We remark that to prove such an analysis correct, we would need to start with a much richer notion of operational semantics than has been used to date. In particular, we would need an operational semantics that characterizes the construction of term structures during unification in logic programming implementations. We also note that a number of details have been omitted in the analysis we have just outlined. For example, consider the following program, and its (somewhat simplified) set constraints:

3. $\leftarrow p(X,Y)^1,\ r(Y)^2.$
4. $p(W,W).$
5. $r(a).$

$$
\begin{aligned}
\mathcal{X}^1 &= \top \\
\mathcal{Y}^1 &= \top \\
\mathcal{X}^2 &= \top \cap \mathcal{W}^4 \\
\mathcal{Y}^2 &= \top \cap \mathcal{W}^4 \\
\mathcal{X}^3 &= \top \cap \mathcal{W}^4 \\
\mathcal{Y}^3 &= \top \cap \mathcal{W}^4 \cap a \\
\mathcal{W}^4 &= \top
\end{aligned}
$$

Now, because of the aliasing of X to Y, there is sharing of structure between X and Y after the execution of the goal $\leftarrow p(X,Y), r(Y)$. However this is not directly reflected in the set constraints for the program. This issue is addressed through the use of labels on the $\top$ constants introduced for the two occurrences of $\mathcal{W}$. More generally, extensions are needed to deal with multiple occurrences of variables in the heads of rules.

Chapter 10

The Unfolding Engine

In order to define a simple, intuitive notion of approximation, set based analysis ignores all inter-variable dependencies. In terms of accuracy, set based analysis offers a superior ability to reason about data structures. The disadvantage is a complete inability to reason about inter-variable dependencies, and it is clear that many kinds of analysis benefit from information about such dependencies. This motivates an investigation of ways to re-incorporate restricted forms of inter-variable dependency into set based analysis. Since abstract interpretation offers a limited ability to reason about inter-variable dependencies, it is natural to consider combinations of set based analysis techniques (for accurate reasoning about data structures) and abstract interpretation (for reasoning about inter-variable dependencies). In this chapter, we present a hybrid algorithm for analysis of logic programs that combines these techniques in way that enhances the effectiveness of both. In particular, we describe a parameterized algorithm, called the *unfolding engine*, that is strictly more accurate than either abstract interpretation or set constraint approaches. Importantly, the underlying approach of unfolding semantic constraints can be easily adapted to other programming languages.

10.1 Motivation

In Section 9.1 we discussed the use of set constraints for mode analysis. The advantage of the set constraint approach lies in its ability to accurately and uniformly reason about structure, and this is particular important for reasoning about partially instantiated terms. However, the set based approach ignores all inter-variable dependencies, and such dependencies are important for reasoning about modes. On the other hand, abstract interpretation approaches to mode analysis are able to capture some information about inter-variable dependencies, but are limited in their ability to reasoning about term structure. These differences are illustrated by the three programs in Figure 9.3 on page 286. For the first of these programs, the set constraint approach performs better than standard abstract interpretation approaches. For the second, abstract interpretation approaches are superior. The third program in the figure is

$$\begin{array}{l} \leftarrow p(Y), r(Y,X). \\ p(U) \leftarrow q(U). \\ q(f(a,V)). \\ r(f(W1,W2),W1). \end{array}$$

and for this program, neither approach is able to determine that X is ground after execution of the goal $\leftarrow p(Y), r(Y,X)$. In this chapter we describe an approach to analysis that addresses these deficiencies and is able to determine that X is ground.

Recall that abstract interpretation approaches to analysis can be viewed as consisting of two main components: an abstract domain and an implementation of an iterative fixed point computation. Now, the key differences between these algorithms usually pertain to the design of the abstract domain and its associated algorithms. Differences between the fixed point computations are relatively minor (often algorithms differ because *ad hoc* approximations are introduced for efficiency reasons). For this reason it is possible to speak of an idealized generic algorithm or "engine" that encompasses the essence of the fixed point computations.

In this chapter we develop a new engine for logic program analysis. As instances of this engine, we obtain analysis algorithms that combine the ability of set based analysis to reason about structural information with the ability of abstract interpretation to reason about dependencies between variables.

The new engine is based upon the use of unfold transformations. While the standard engine iterates over fixed semantic equations, this *unfolding engine* iterates over dynamically changing equations. The main result shows that the unfolding engine is uniformly more accurate than the standard engine in the sense that, given an abstract domain, the output of the unfolding engine, for any program, is more accurate than that of the standard engine. We remark that this chapter contains essentially the same material as [26].

10.2 Collecting Semantics

Most program analysis start with what is essentially a representation of an *approximation* of a program's collecting semantics. This is essentially obtained by starting with semantic equations that characterize the collection semantics, and then replacing exact operations in these equations by approximate operations. The analysis then proceeds by using these *approximate* semantic equations. In contrast, the unfolding engine starts with a representation of the collecting semantics that is *exact*, and *changes* the representation during its execution.

We begin by describing the initial representation of the collecting semantics. Again, the representation employs constraints rather than equations. These constraints are similar to the environment constraints described in Chapter 4. However the environment constraints provide a characterization of the collecting semantics in terms of mappings from program variables to values (*ground* terms), and so they are not well suitable to studying properties such as the instantiation level of variables. Moreover, the environment constraints are based on a very abstract notion of operational model that does not directly incorporate any notion of equation solving or unification. Hence properties specific to the equation solving process cannot be inferred.

The new representation of collecting semantics commits to a specific operational model based on unification (more generally, it would be appropriate to employ operational models parameterized by some notion of *constraint solving*). We begin by consider constraint corresponding to bottom-up execution. Consider Figure 10.1, which contains the standard logic program for appending two lists. Now, to construct constraints to capture the computed answer substitutions obtained from executing the body of each rule in this program, first introduce set variables Ψ^1 and Ψ^2 to represent the sets

1. $app(nil, W, W).$
2. $app(X.Xs, Y, X.Zs) \leftarrow app(Xs, Y, Zs).$

Figure 10.1: **Append Program**

of answer substitutions for rules 1 and 2 respectively. Then construct the following constraints:

$$\Psi^1 \supseteq \{mgu()\}\Big|_{S_1}$$
$$\Psi^2 \supseteq \Big\{mgu\Big(app(Xs, Y, Zs) = \Big\| \theta(app(nil, W, W)) \Big\|_{S_2}\Big) : \theta \in \Psi^1\Big\}\Big|_{S_2}$$
$$\Psi^2 \supseteq \Big\{mgu\Big(app(Xs, Y, Zs) = \Big\| \theta(app(X.Xs, Y, X.Zs)) \Big\|_{S_2}\Big) : \theta \in \Psi^2\Big\}\Big|_{S_2}$$

where S_1 and S_2 denote the sets of variables in rules 1 and 2 respectively. The function mgu denotes the unification algorithm at hand (we shall assume a fixed unification algorithm throughout this chapter), which maps a set of equations to its most general unifier. If the set of equations is empty, then the identity substitution is returned. Where S is a set of program variables, the function $|_S$ restricts the domain of a set of substitutions to the variables S. For example, if θ maps X into b, Y into c and maps all other variables Z into Z, then $\{\theta\}|_{\{X\}}$ consists of the single substitution that maps X into b and is the identity on all other variables. Finally, where t is a term (constructed from predicates and function symbols), $\|t\|_S$ denotes the renaming of the variables in t into new variables so that $var(\|t\|_{\mathcal{V}}) \cap S = \{\}$.

A *model* of such constraints maps both Ψ^1 and Ψ^2 into a set of substitutions, such that each constraint is satisfied. Models are ordered componentwise, so that $Model_1 \leq Model_2$ iff $Model_1(\Psi) \subseteq Model_2(\Psi)$ for each variable Ψ. As an example,

$$\Psi^1 \;\mapsto\; \{(W \mapsto W)\}$$
$$\Psi^2 \;\mapsto\; \left\{ \begin{bmatrix} Xs \mapsto nil \\ Y \mapsto W \\ Zs \mapsto W \end{bmatrix}, \begin{bmatrix} Xs \mapsto X'.nil \\ Y \mapsto W \\ Zs \mapsto X'.W \end{bmatrix}, \begin{bmatrix} Xs \mapsto X'.X''.nil \\ Y \mapsto W \\ Zs \mapsto X'.X''.W \end{bmatrix}, \ldots \right\}$$

where $X', X'', \ldots$ are new variables, is a model of the above constraints for the append program. It is in fact the least such model (up to renaming of new variables). It coincides exactly with the program's meaning in the sense that the set for Ψ^i consists of the computed answer substitutions obtained when the body of the i^{th} rule is executed as a goal.

More formally, let P be a logic program and introduce variables Ψ^α for each rule label in P. Recall that if α is a rule label, then $var(\alpha)$ denotes the set of variables in the rule with label α. The constraints for P are defined as follows.

Definition 19 (Bottom-up Semantic Constraints) *For each rule $R^\alpha \in P$ with body $A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$, construct the constraints*

$$\Psi^\alpha \supseteq \left. \left\{ mgu \begin{pmatrix} A_1 = \left\| \theta_1(B_1) \right\|_{var(\alpha)} \\ \cdots \\ A_n = \left\| \theta_n(B_n) \right\|_{var(\alpha)} \end{pmatrix} : \theta_1 \in \Psi^{\beta_1} \wedge \cdots \wedge \theta_n \in \Psi^{\beta_n} \right\} \right|_{var(\alpha)}$$

where the β_i, $i \geq 1$, range over rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible. $\square$

Before providing more examples, we first introduce an abbreviation for the right hand sides of these constraints. Specifically, we shall abbreviate expressions of the form

$$\left. \left\{ mgu\left(s_1 = \left\| \theta_1(t_1) \right\|_{var(\alpha)}, \ldots, s_n = \left\| \theta_n(t_n) \right\|_{var(\alpha)} \right) : \begin{array}{c} \theta_i \in \Psi^{\beta_i} \\ i = 1..n \end{array} \right\} \right|_{var(\alpha)}$$

by $\left([s_1 = t_1]_{\beta_1}, \ldots, [s_n = t_n]_{\beta_n} \right)_\alpha$. Such an expression is called a *cluster*. For example, the constraints for the append program (Figure 10.1) can be written as follows. Note that the right hand side of the first constraint is an empty cluster.

$$\begin{array}{lcl} \Psi^1 & \supseteq & ()_1 \\ \Psi^2 & \supseteq & ([app(Xs, Y, Zs) = app(nil, W, W)]_1)_2 \\ \Psi^2 & \supseteq & ([app(Xs, Y, Zs) = app(X.Xs, Y, X.Zs)]_2)_2 \end{array}$$

In general, expressions such as $[s = t]_\alpha$ contained in clusters are called *groups*, and the individual equations contained in groups are called *group equations*. (Note that group equations are not pure equations; thus $[s = t]_\alpha$ is different from $[t = s]_\alpha$.) The subscript α indicates the rule α, and is called the *dependency* of the group.

1. $p(f(X)) \leftarrow p(X), q(X), r(Y).$
2. $p(b).$
3. $q(W).$
4. $r(c).$

$$
\begin{array}{lcl}
\Psi^1 & \supseteq & \Big(\; [p(X) = p(f(X))]_1 \, , \; [q(X) = q(W)]_3 \, , \; [r(Y) = r(c)]_4 \; \Big)_1 \\
\Psi^1 & \supseteq & \Big(\; [p(X) = p(b)]_2 \, , \; [q(X) = q(W)]_3 \, , \; [r(Y) = r(c)]_4 \; \Big)_1 \\
\Psi^2 & \supseteq & ()_2 \\
\Psi^3 & \supseteq & ()_3 \\
\Psi^4 & \supseteq & ()_4
\end{array}
$$

Figure 10.2: **Example Clusters Containing More Than One Group**

Figure 10.2 contains another program and its semantic constraints, illustrating clusters that contain more than one group.

In the least model, Ψ^1 is $\{(X \mapsto b, Y \mapsto c),\ (X \mapsto f(b), Y \mapsto c),\ (X \mapsto f(f(b)), Y \mapsto c), \ldots\}$.

We conclude by describing the semantic constraints for top-down left-to-right execution. Again, let P be a logic program and introduce variables Ψ^α for each label α in P. Recall that if α is a rule label, then $var(\alpha)$ denotes the set of variables in the rule with label α, and if α is a body atom label, then $var(\alpha)$ denotes the set of variables in the rule that contains the body atom with label α. Also recall that goals are treated as rules without heads. Using the abbreviated notation for clusters, the top-down constraints for P can be defined as follows.

Definition 20 (Top-Down Semantic Constraints) *For each rule $R^\alpha \in P$ with body $A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$ and head A_0 (if it exists), $\mathcal{EC}_P$ contains the constraints*

$$\begin{array}{rcl}
\Psi^{\alpha_1} & \supseteq & \Big([A_0 \in B_0]_{\beta_0}\Big)_\alpha \\
\Psi^{\alpha_2} & \supseteq & \Big([A_0 \in B_0]_{\beta_0}, [A_1 \in B_1]_{\beta_1}\Big)_\alpha \\
\vdots & \vdots & \quad\vdots \qquad\qquad\quad \vdots \\
\Psi^{\alpha_n} & \supseteq & \Big([A_0 \in B_0]_{\beta_0}, [A_1 \in B_1]_{\beta_1}, \ldots, [A_{n-1} \in B_{n-1}]_{\beta_{n-1}}\Big)_\alpha \\
\Psi^{\alpha} & \supseteq & \Big([A_0 \in B_0]_{\beta_0}, [A_1 \in B_1]_{\beta_1}, \ldots, [A_n \in B_n]_{\beta_n}\Big)_\alpha
\end{array}$$

where β_0 ranges over body atom labels such that $B_0^{\beta_0}$ is a body atom in P and A_0 and B_0 are compatible, and the β_i, $i \geq 1$, range over rule labels such that $B_i^{\beta_i}$ is a head atom in P and A_i and B_i are compatible. □

If the rule R is a goal, then A_0 does not exist, and the group $[A_0 \in B_0]_{\beta_0}$ is simply deleted from each expression. Figure 10.3 illustrates the construction of the top-down constraints using the append program.

Consider the third rule of the program. Recalling the notation for program points, point 4 denotes the point just before $app(Xs, Y, Zs)$ is called, and point 5 denotes the point after the body of the third rule has completed execution. There are six constraints corresponding to this rule. Two of them define the set of substitutions encountered at point 4 (and correspond the possibility that this rule can be called from two places in the program), and four of them define the set of substitutions at point 5 (and correspond to the four combinations of possible calls to this rule and returns from $app(Xs, Y, Zs)$). Note that, strictly speaking, the second and fourth constraints in this collection are omitted because $app(nil, W, W)$ and $app(b.nil, c.nil, V)$ are not compatible.

10.3 Unfolding Semantic Equations

A key idea behind our engine is the substitution of one cluster into another. Such a substitution step may generate a new cluster. The engine is essentially an exhaustive application of this step. We now elaborate.

2. $\leftarrow app(b.nil, c.nil, V)^1$
3. $app(nil, W, W).$
5. $app(X.Xs, Y, X.Zs) \leftarrow app(Xs, Y, Zs)^4.$

$$
\begin{array}{lll}
\Psi^1 & \supseteq & \Big(\,\Big)_1 \\
\Psi^2 & \supseteq & \Big([app(b.nil, c.nil, V) = app(nil, W, W)]_3\Big)_2 \\
\Psi^2 & \supseteq & \Big([app(b.nil, c.nil, V) = app(X.Xs, Y, X.Zs)]_5\Big)_2 \\
\Psi^3 & \supseteq & \Big([app(nil, W, W) = app(b.nil, c.nil, V)]_1\Big)_3 \\
\Psi^3 & \supseteq & \Big([app(nil, W, W) = app(Xs, Y, Zs)]_4\Big)_3 \\
\Psi^4 & \supseteq & \Big([app(X.Xs, Y, X.Zs) = app(b.nil, c.nil, V)]_1\Big)_3 \\
\Psi^4 & \supseteq & \Big([app(X.Xs, Y, X.Zs) = app(Xs, Y, Zs)]_4\Big)_3 \\
\Psi^5 & \supseteq & \left(\begin{array}{c} [app(X.Xs, Y, X.Zs) = app(b.nil, c.nil, V)]_1 \\ [app(Xs, Y, Zs) = app(nil, W, W)]_3 \end{array}\right)_3 \\
\Psi^5 & \supseteq & \left(\begin{array}{c} [app(X.Xs, Y, X.Zs) = app(b.nil, c.nil, V)]_1 \\ [app(Xs, Y, Zs) = app(X.Xs, Y, X.Zs)]_5 \end{array}\right)_3 \\
\Psi^5 & \supseteq & \left(\begin{array}{c} [app(X.Xs, Y, X.Zs) = app(Xs, Y, Zs)]_4 \\ [app(Xs, Y, Zs) = app(nil, W, W)]_3 \end{array}\right)_3 \\
\Psi^5 & \supseteq & \left(\begin{array}{c} [app(X.Xs, Y, X.Zs) = app(Xs, Y, Zs)]_4 \\ [app(Xs, Y, Zs) = app(X.Xs, Y, X.Zs)]_5 \end{array}\right)_3
\end{array}
$$

Figure 10.3: **Top-Down Semantics Constraints for Append**

Consider a group $\mathcal{G}$ whose dependency α is a fact, that is, the rule with label α has no body. Since the equation for rule α is of the form $\Psi^\alpha \supseteq ()$, any model of the semantic constraints must map Ψ^α to the singleton set consisting of the identity substitution. Hence $\mathcal{G}$ has a fixed interpretation in any model, and can be thought of as representing a fixed set of substitutions. Call such groups $\mathcal{G}$ *terminal.* A group is in *substitution form* if the left hand side of each equation therein is a variable; a cluster is in substitution form if each of its groups is in substitution form.

We illustrate informally how such clusters are used as substitutions by returning to the bottom-up constraints for the append program (Figure 10.1). By rewriting equations such as $p(s_1, \ldots, s_n) = p(t_1, \ldots, t_n)$ into $s_1 = t_1, \ldots, s_n = t_n$, these constraints may be simplified as follows.

$$\begin{array}{lll}
\Psi^1 & \supseteq & ()_1 \\
\Psi^2 & \supseteq & \left(\left[\begin{array}{rcl} Xs & = & nil \\ Y & = & W \\ Zs & = & W \end{array} \right]_1 \right)_2 \\
\Psi^2 & \supseteq & \left(\left[\begin{array}{rcl} Xs & = & X.Xs \\ Y & = & Y \\ Zs & = & X.Zs \end{array} \right]_2 \right)_2
\end{array}$$

Denote the two clusters in the constraints for Ψ^2 by cl_1 and cl_2 respectively. cl_1 contains a terminal group $\mathcal{G}_1$ that represents the substitution $[Xs \mapsto nil,\ Y \mapsto W,\ Zs \mapsto W]$. cl_2 contains a group $\mathcal{G}_2$ that is non-terminal. By substituting cl_1 into cl_2, we obtain the new cluster:

$$\left(\left[\begin{array}{rcl} Xs & = & X.nil \\ Y & = & W \\ Zs & = & X.W \end{array} \right]_1 \right)_2 \tag{10.40}$$

and this can be used to construct a new constraint for Ψ^2:

$$\Psi^2 \supseteq \left(\left[\begin{array}{rcl} Xs & = & X.nil \\ Y & = & W \\ Zs & = & X.W \end{array} \right]_1 \right)_2 \tag{10.41}$$

The correctness of a substitution step follows from the fact that it is essentially an "unfolding" operation using the definition of Xs, Y, and Zs. That is, the new expression (10.40) is subsumed by cl_2 in the following sense:

in all models $\mathcal{I}$ of the original constraints, $\mathcal{I}((10.40)) \subseteq \mathcal{I}(cl_2)$. Hence the addition of the constraint (10.41) does not change the least model of the constraints.

However, the new equation is more explicit in the sense that it displays more elements of the set Ψ^2 (as terminal groups). Clearly the process of unfolding of constraints can be repeated indefinitely to obtain new clusters $([Xs = X.X'.nil, Y = W, Zs = X.X'.W]_1)_2$, …, etc. Note that each such cluster represents an answer substitution for the body of the second rule of the append program. To obtain a terminating analysis algorithm, this process of adding new clusters must be curtailed by using a notion of approximation. Consider approximating the new groups constructed during the substitution step. Each such group is terminal and hence defines a fixed set of substitutions. Hence any traditional means for approximating substitution sets can be employed (see, for example, [36]). We present one formalization as follows.

Definition 21 *An abstract domain $\mathcal{D}$ is a set of abstract formulas together with (a) an abstraction function* ABS *that maps a set of variables and a set of substitutions into an abstract formula, and (b) a concretization function* CONC *that maps an abstract formula into a set of substitutions such that* $\text{CONC}(\text{ABS}(S, \Theta)) \supseteq \Theta$ *for all Θ and finite sets S of variables. (Typically, some further algebraic conditions are specified, but these will not concern us here.) The definition of* ABS *and* CONC *induces a notion of abstract unification*[1]

$$\mathcal{D}\text{-}unify(S, s, t, \mathcal{A}) \stackrel{\text{def}}{=} \text{ABS}\Big(S, \Big\{mgu\Big(s = \|\theta(t)\|_{var(s)}\Big) : \theta \in \text{CONC}(\mathcal{A})\Big\}\Big)$$

where s and t are terms, S is a set of program variables and $\mathcal{A} \in \mathcal{D}$. □

As an example, consider the propositional formula domain used by Marriot and Sondergaard to capture relationships between groundness of variables [44]. Specifically, let $\mathcal{D}_{prop}$ consist of all propositional formulas over program variables. For example, X, $X \wedge Y$, $X \vee Y$, $X \iff Y$ are three formulas in $\mathcal{D}_{prop}$. In essence, each formula $\mathcal{A}$ denotes the set of substitutions that satisfies $\mathcal{A}$. If $\mathcal{A}$ is a variable, say X, then θ satisfies $\mathcal{A}$ if $\theta(X)$ is

[1] For pragmatic reasons, some analysis algorithms do not implement $\mathcal{D}$-*unify* exactly, but instead use a conservative approximation.

a ground term. The definition of satisfies is extended to non-variable formulas using the interpretation of the propositional connectives. For example, the proposition $X \vee Y$ is *true* in θ if θ grounds *either* X or Y. Similarly $X \wedge Y$ is *true* if θ grounds *both* X and Y, and $X \iff Y$ is true if θ either grounds both X and Y or grounds neither of them. CONC and ABS can now be defined as follows:

$$\begin{array}{rcl}
\textsc{conc}(\mathcal{A}) & \stackrel{\text{def}}{=} & \{\theta : \theta \text{ satisfies A}\} \\
\textsc{abs}(S, \Theta) & \stackrel{\text{def}}{=} & \left(\begin{array}{l} \text{the strongest proposition } \mathcal{A} \text{ such that} \\ var(\mathcal{A}) \subseteq S \text{ and } \mathcal{A} \text{ is true for all } \theta \in \Theta \end{array}\right)
\end{array}$$

For instance, $mgu(X = f(a, Y))$ can be approximated with respect to variables $\{X, Y\}$ by the formula $X \iff Y$; similarly $mgu(Xs = X.nil, Y = W, Zs = X.W)$ with respect to $\{Xs, Y, Zs\}$ can be approximated by $(Xs \wedge Y \iff Zs)$.

Returning to the bottom-up semantic constraints for append, we recall that the unfolding process may generate an unbounded number of new clusters. This can be curtailed using the abstract domain $\mathcal{D}_{prop}$ in a straightforward way: approximate any new groups $\mathcal{G}$ constructed during the unfolding step using ABS. Hence, instead of producing the constraint (10.41), the first unfolding step now produces the constraint

$$\Psi^2 \supseteq \left(\left[\ Xs \wedge Y \iff Zs\ \right]_1\right)_2$$

Note, however, that this approximation of the unfolding step loses a significant amount of structural information. In fact, when a group $\mathcal{G}_1$ is substituted into a group $\mathcal{G}_2$, the resulting cluster loses all information about these two groups except for that contained in the $\mathcal{D}$-approximation. A key observation of this paper is that the group $\mathcal{G}_2$ can in fact be retained. Specifically, we define that the substitution of $\mathcal{G}_1$ into $\mathcal{G}_2$, which originally yielded a single group $[Xs \wedge Y \iff Zs]$, now yields two groups:

$$\left[\ Xs \wedge Y \iff Zs\ \right]_1 \quad \text{and} \quad \left[\begin{array}{rcl} Xs & = & X.Xs \\ Y & = & Y \\ Zs & = & X.Zs \end{array}\right]_2^{\text{®}}$$

In other words, the group $\mathcal{G}_2$ remains after the substitution, and is marked as a *residual group* (indicated with the superscript with ®). It is these residual groups that provide our engine with the ability to reason about structures.

10.4 The Engine

We begin with some definitions. We write $\tilde{s}$ to denote a sequence of terms $s_1, \ldots, s_n$. Similarly, if $\tilde{s}$ denotes $s_1, \ldots, s_n$ and $\tilde{t}$ denotes $t_1, \ldots, t_n$, we write $\tilde{s} = \tilde{t}$ to denote the sequence of equations $s_1 = t_1, \ldots, s_n = t_n$. An *exact group* is of the form $[s_1 = t_1, \ldots, s_n = t_n]_\alpha$ where the s_i and t_i are program terms (constructed from function and predicate symbols and program variables). We shall frequently write exact groups using the sequence notation as $[\tilde{s} = \tilde{t}]_\alpha$. An exact group is *terminal* if the rule with label α is a fact. Some exact groups may be marked *residual* during execution of the engine. An *approximate group* is defined with respect to a given abstract domain $\mathcal{D}$; it is simply of the form $[\mathcal{A}]_\alpha$ where $\mathcal{A} \in \mathcal{D}$. A *group* is an exact or approximate group. A *cluster* is of the form $(\mathcal{G}_1, \ldots, \mathcal{G}_n)_\alpha$ where each $\mathcal{G}_i$ is a group. In the previous section, semantic constraints were constructed from a program, and these form the starting point of the unfolding engine. At all times, corresponding to each rule R^α, there is a collection of constraints of the form $\Psi^\alpha \supseteq cl$ where cl is a cluster and Ψ^α is the distinguished set variable corresponding to rule label α.

Let $\mathcal{G}$ be the exact group $[\tilde{s} = \tilde{t}]_\alpha$. $\mathcal{G}$ is in *substitution form* if $\tilde{s}$ is a sequence of variables. Now, suppose that $\mathcal{G}$ is in substitution form and let $X_1, \ldots, X_m$ be a listing of the variables in $var(\tilde{s})$. Now define $subs(\mathcal{G})$ to be the set of all substitutions of the form $[X_1 \mapsto u_1, \ldots, X_m \mapsto u_m]$ such that $X_i = u_i$ appears in $\mathcal{G}$, $i = 1..m$. In essence, $subs(\mathcal{G})$ consists of all substitutions defined by taking subsets of the equations in $\mathcal{G}$ such that each subset contains exactly one equation for each variable X_i, $i = 1..m$. A cluster is in *substitution form* if each group therein is either an approximate group, or a terminal group in substitution form, or is marked as a residual group. An exact group can be made into substitution form by deleting certain equations:

$$\widehat{[\mathcal{E}]}_\alpha \stackrel{\text{def}}{=} [X = t : X = t \in \mathcal{G} \text{ and } X \text{ is a variable }]_\alpha.$$

This operator will be used exclusively in the construction of residual groups.

The basic operation of the unfolding engine is the notion of substituting a cluster in substitution form *into* a group. What results is a new cluster. First define the composition of an exact group with an approximate group as follows:

$$[\tilde{s} = \tilde{t}]_\alpha \circ [\mathcal{A}]_\beta \quad \stackrel{\text{def}}{=} \quad [\mathcal{D}\text{-}unify(S, \tilde{s}, \tilde{t}, \mathcal{A})]_\beta$$

where S denotes $var(s_1, \ldots, s_n)$, and $\mathcal{D}$-*unify*$(S, \tilde{s}, \tilde{t}, \mathcal{A})$ denotes the extension of $\mathcal{D}$-*unify* to sequences:

$$\mathcal{D}\text{-}unify(S, \tilde{s}, \tilde{t}, \mathcal{A}) \stackrel{\text{def}}{=} \text{ABS}\left(S, \left\{mgu\left(\tilde{s} = \left\| \theta(\tilde{t}) \right\|_{var(\tilde{s})}\right) : \theta \in \text{CONC}(\mathcal{A})\right\}\right).$$

Here $\|\theta(\tilde{t})\|_{var(\tilde{s})}$ renames the sequence $\theta(\tilde{t})$ so that it has no variables in common with $var(\tilde{s})$.

Similarly, the composition of an exact group with an exact group in substitution form can be defined:

$$[\mathcal{E}]_\alpha \circ [\tilde{X} = \tilde{u}]_\beta \quad \stackrel{\text{def}}{=} \quad \left[s = t\theta : \ (s = t) \in \mathcal{E} \text{ and } \theta \in subs\left(\tilde{X} = \left\| \tilde{u} \right\|_S\right)\right]_\beta$$

where S denotes the variables in the right hand sides of the equations in $\mathcal{E}$, that is $S = \bigcup_{(s=t)\in\mathcal{E}} var(t)$. Intuitively, the renaming operation $\|\tilde{u}\|_S$ in this definition is required because there is a renaming operation implicit in the interpretation of both of the exact groups $[\mathcal{E}]_\alpha$ and $[\tilde{X} = \tilde{u}]_\beta$, but there is only one such operation in the resulting group. In particular, without this renaming, $[X = f(W, Y)] \circ [Y = W]$, would incorrectly yield $[X = f(W, W)]$.

We also define an approximate notion of the composition of an exact group with an exact group in substitution form. Specifically, where $[\mathcal{E}_1]_\alpha$ is an exact group and $[\mathcal{E}_2]_\beta$ is an exact group in substitution form, define that the *approximate composition* $[\mathcal{E}_1]_\alpha \diamond [\mathcal{E}_2]_\beta$ consist of two groups. The first group is simply the $\mathcal{D}$-approximation of $[\mathcal{E}_1]_\alpha \circ [\mathcal{E}_2]_\beta$:

$$[\text{ABS}\,(S, mgu\,([\mathcal{E}_1]_\alpha \circ [\mathcal{E}_2]_\beta))]_\beta$$

where $\mathcal{E}_1$ is $\tilde{s} = \tilde{t}$ and S is $var(\tilde{s})$. The second group is a subset of $[\mathcal{E}_1]_\alpha \circ [\mathcal{E}_2]_\beta$ selected by the function *select*. Specifically, if $\mathcal{E}_2$ is $\tilde{X} = \tilde{u}$, then *select* $([\mathcal{E}_1]_\alpha, [\mathcal{E}_2]_\beta)$ is defined to be

$$\left[s = t\theta : \begin{array}{l} (s = t) \in \mathcal{E}_1 \text{ and } \theta \in subs\left(\tilde{X} = \|\tilde{u}\|_S\right) \\ \text{and either } t \text{ is a variable or } |t\theta| = |t| \end{array}\right]_\beta$$

where S denotes the variables in the right hand sides of the equations in $\mathcal{E}_1$, and $|t|$ is the number of symbols in the term t. Intuitively, *select* chooses equations from $[\mathcal{E}_1]_\alpha \circ [\mathcal{E}_2]_\beta$ in such a way that the size of any term therein does not exceed the size of the terms in the original equations $[\mathcal{E}_1]_\alpha$ and $[\mathcal{E}_2]_\beta$. For example, if $[\mathcal{E}_1]_\alpha$ and $[\mathcal{E}_2]_\beta$ are $[X = W_1, Y = f(W_1), Z = f(W_2)]_\alpha$ and $[W_1 = g(a), W_2 = V]_\beta$ respectively, then $[\mathcal{E}_1]_\alpha \circ [\mathcal{E}_2]_\beta$ is $[X = g(a), Y = f(g(a)), Z = f(V)]_\beta$. However $select([\mathcal{E}_1]_\alpha, [\mathcal{E}_2]_\beta)$ will contain the first and third equations, but not the second because $f(g(a))$ is too big. As we see later, it is possible to consider other definitions of *select*.

We are now in a position to define the main step of our engine.

Definition 22 (Unfolding Step) *Let cl_1 be a cluster containing a non-terminal group $\mathcal{G}$. Let cl_2 be a cluster in substitution form. Construct a new cluster by replacing $\mathcal{G}$ in cl_1 by all of the following groups:*

- $\widehat{\mathcal{G}}$, *and this group is marked as residual;*
- $\mathcal{G} \circ [\mathcal{A}]$, *for all approximate groups $[\mathcal{A}] \in cl_2$, and*
- $\mathcal{G} \diamond [\mathcal{E}]_\alpha$, *for all exact groups $[\mathcal{E}]_\alpha \in cl_2$.*

We say that this new cluster is the result of <u>substituting</u> cl_2 <u>into</u> cl_1 <u>at</u> $\mathcal{G}$.
□

For example, let cl_1 and cl_2 be the following two clusters (where, for simplicity, we have omitted the subscripts on clusters and groups):

$$\Big(\big[\ X = f(Y)\ \big] \Big) \qquad \left(\begin{array}{c} \left[\begin{array}{l} Y = U \\ Y = g(b) \\ Z = c \end{array} \right] \\ \big[\ Y = V\ \big] \end{array} \right)$$

The substitution of cl_2 into cl_1 at the group $[X = f(Y)]$, using the abstract domain $\mathcal{D}_{prop}$, results in the following cluster:

$$\left(\begin{array}{c} \big[\ X = f(Y)\ \big]^{®} \\ \big[\ X = f(U)\ \big] \\ \big[\ X\ \big] \\ \big[\ X = f(V)\ \big] \\ \big[\ true\ \big] \end{array} \right)$$

The first group $[X = f(Y)]$ in this cluster is residual. The second group $[X = f(U)]$ is obtained from *select*$([X = f(Y)] \circ [Y = U, Y = g(b), Z = c])$. (Note that the equation $X = f(g(b))$ is omitted by *select*.) The third group $[X]$ is approximate, and is obtained from $\text{ABS}(\{X\}, mgu([X = f(Y)] \circ [Y = U, Y = g(b), Z = c]))$. It contains the proposition that X is ground. The fourth group $[X = f(V)]$ is obtained from *select*$([X = f(Y)] \circ [Y = V])$. Finally, the fifth group $[true]$ is approximate and is obtained from $\text{ABS}(\{X\}, mgu([X = f(Y)] \circ [Y = V]))$.

We now define how unfolding steps are applied to a collection of semantic constraints. Let $\mathcal{C}$ be a collection of semantic constraints and consider the following transformation step.

Transformation 19 (Unfolding) *If $\mathcal{C}$ contains the constraints $\Psi^\alpha \supseteq cl_1$ and $\Psi^\beta \supseteq cl_2$ such that cl_1 contains a group $[\mathcal{E}]_\beta$ and cl_2 is in substitution form, then let cl_3 be the result of substituting cl_2 into cl_1 at the group $[\mathcal{E}]_\beta$, and construct the constraint $\Psi^\alpha \supseteq cl_3$. If this constraint does not appear in $\mathcal{C}$, then the unfolding step is said to be effective, and the constraint is added to $\mathcal{C}$.*

The complete engine is presented in figure 10.4. It is essentially an exhaustive application of the unfolding step just defined. Note that before an unfolding step is performed, it is necessary to perform some basic simplification steps. These straightforward simplifications[2] are:

- replace a group equation of the form $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ by $s_1 = t_1, \ldots, s_n = t_n$;
- replace any group of the form $[s_1 = t_1, \ldots, s_n = t_n]$, where one of the equations is of the form $f(u_1, \ldots, u_k) = W$ and W is a head-only variable, by $[s_1 = t_1\theta, \ldots, s_n = t_n\theta]$ where θ is the substitution $[W \mapsto f(W_1, \ldots, W_k)]$ and the variables $W_1, \ldots, W_k$ are new variables. (Intuitively, a group such as $[f(X) = W, Y = W]$ does define a substitution, but is not in substitution form. This simplification step remedies the problem by transforming it into $[f(X) = f(W_1), Y = f(W_1)]$, which in turn can be simplified into $[X = W_1, Y = f(W_1)]$.)

[2] We also observe that we can effectively ignore any constraint of the form $\mathcal{X} \supseteq cl$ such that cl contains either (a) a group equation of the form $f(\tilde{s}) = g(\tilde{t})$, $f \neq g$, or (b) two group equations of the form $X = f(\tilde{s})$ and $X = g(\tilde{t})$, $f \neq g$.

input an arbitrary program P;
obtain the semantic constraints associated with P;
repeat
  exhaustively apply the basic simplification steps;
  apply an effective unfolding step if possible;
until the constraints do not change;
delete all residual groups[3];
output all constraints involving substitution form clusters;

Figure 10.4: **The Unfolding Engine**

1. $\leftarrow p(f(X)),\ q(X,Y).$
2. $p(U) \leftarrow r(U).$
3. $q(g(W1,W2),W1).$
4. $r(g(c,W3)).$
5. $r(f(V)) \leftarrow r(V).$

$$\Psi^1 \supseteq \begin{pmatrix} [f(X) = U]_2 \\ [X = g(W1,W2),\ Y = W1]_3 \end{pmatrix}_1^{(a)}$$

$$\Psi^2 \supseteq \left([U = g(c,W3)]_4\right)_2^{(b)}$$

$$\Psi^2 \supseteq \left([U = f(V)]_5\right)_2^{(c)}$$

$$\Psi^3 \supseteq ()_3$$

$$\Psi^4 \supseteq ()_4$$

$$\Psi^5 \supseteq \left([V = g(c,W3)]_4\right)_5^{(d)}$$

$$\Psi^5 \supseteq \left([V = f(V)]_5\right)_5^{(e)}$$

Figure 10.5: **Example to Illustrate Execution of Unfolding Engine**

When no more effective unfolding steps can be performed, all clusters in substitution form are output after first deleting any residual groups. Such clusters represent fixed sets of substitutions because the groups they contain are independent of the values given to the Ψ^α variables. The important point here is that the output is an *explicit* representation of a set of substitutions for each variable Ψ^α. A post-processing phase may then be applied to extract the specific information sought.

Consider the program and its (simplified) semantic constraints shown in Figure 10.5. The labels (a), (b), ... are used to identify clusters in the

[3] These groups in fact do contain some structural information. We address the issue of extracting this information in the next section. For now, we shall simply ignore them.

following discussion. Two of the clusters in the constraints are in substitution form, namely (b) and (d). Using these clusters, the following three unfolding steps can be performed: cluster (b) can be substituted into (a) and cluster (d) can be substituted into (c) and (e). These three unfolding steps respectively result in the following new constraints.

$$\begin{array}{lll}
\Psi^1 & \supseteq & \begin{pmatrix} [f(X) = g(c, W3)]_4 \\ [X = g(W1, W2),\ Y = W1]_3 \end{pmatrix}_1^{(f)} \\
\Psi^2 & \supseteq & \begin{pmatrix} [U = f(V)]_5^{\circledR} \\ [true]_4 \end{pmatrix}_2^{(g)} \\
\Psi^5 & \supseteq & \begin{pmatrix} [V = f(V)]_5^{\circledR} \\ [true]_4 \end{pmatrix}_5^{(h)}
\end{array}$$

In the first of these unfolding steps, there is no residual group in (f) because $\widehat{[\mathcal{E}]} = [\,]$ when $[\mathcal{E}] = [f(X) = U]$. The new constraint involving (f) is vacuous because it contains the conflicting group equation $f(X) = g(c, W3)$.

Next, clusters (g) and (h) can be used in unfolding steps. We omit the details for (h) since these steps do not lead to any new constraints. Substituting (g) into (a) yields the constraint

$$\Psi^1 \quad \supseteq \quad \begin{pmatrix} [f(X) = f(V)]_5 \\ [true]_4 \\ [X = g(W1, W2),\ Y = W1]_3 \end{pmatrix}^{(i)}$$

which is subsequently simplified into

$$\Psi^1 \quad \supseteq \quad \begin{pmatrix} [X = V]_5 \\ [true]_4 \\ [X = g(W1, W2),\ Y = W1]_3 \end{pmatrix}^{(j)}$$

Now, there are two possible substitutions into this new cluster: (d) into (j) and (h) into (j). These two substitution steps respectively yield

$$\Psi^1 \supseteq \begin{pmatrix} [X = V]_5^{®} \\ [X = g(c, W3)]_4 \\ [true]_4 \\ [X = g(W1, W2),\ Y = W1]_3 \end{pmatrix}^{(k)}$$

$$\Psi^1 \supseteq \begin{pmatrix} [X = V]_5^{®} \\ [X = f(V)]_5 \\ [true]_4 \\ [X = g(W1, W2),\ Y = W1]_3 \end{pmatrix}^{(l)}$$

The second constraint is vacuous because it contains the conflicting equations $X = f(V)$ and $X = g(W1, W2)$. At this point the engine terminates, and outputs the constraints involving the following clusters: (b), (d), (g), (h), and (k). The only output constraint for Ψ^1 is that involving cluster (k). This cluster represents the fixed collection of substitutions given by $mgu(X = g(c, W3), X = g(W1, W2), Y = W1)$ conjoined with *true*. This in turn can be simplified and projected onto $\{X, Y\}$ to obtain $\{(X \mapsto g(c, t), Y \mapsto c)\ :\ t \text{ is any term}\}$ from which we can deduce that Y is ground.

To see the need for residual groups in the above example, consider removing the residual group in (g). Then the group $[f(X) = f(V)]_5$ would not be in (i), and this implies that the group $[X = V]_5$ would not be in (j), and this implies that the group $[X = g(c, W3)]_4$ would not be in (k). Hence it cannot be inferred that Y is ground.

One could raise the question as to why the group in (c), which gave rise to the residual group in (g), could not *itself* be used for substitution into (a). (That is, consider substituting (c) into (a).) Allowing this implies that groups that are neither terminal nor residual are allowed to be substituted into other clusters. Hence many more unfolding steps are permitted in general. Moreover, this results in substantial loss of information. To see this, consider the program and (simplified) semantics constraints in Figure 10.6. The engine dictates that cluster (o) is first substituted into cluster (n) to obtain the constraint:

$$\Psi^2 \supseteq \begin{pmatrix} [X_1 = f(Y_1), X_2 = f(Y_2)]_3^{®} \\ [X_1 \iff X_2]_4 \end{pmatrix}_2^{(p)}$$

Cluster (p) is then substituted into cluster (m) to obtain (after simplifica-

1. $\leftarrow p(U_1, f(U_2))$.
2. $p(f(X_1), X_2) \leftarrow q(X_1, X_2)$.
3. $q(f(Y_1), f(Y_2)) \leftarrow r(Y_1, Y_2)$.
4. $r(f(X), f(X))$.

$$\begin{array}{lll}
\Psi^1 & \supseteq & \left(\begin{bmatrix} U_1 = f(X_1) \\ f(U_2) = X_2 \end{bmatrix}_2 \right)_1^{(m)} \\
\Psi^2 & \supseteq & \left(\begin{bmatrix} X_1 = f(Y_1) \\ X_2 = f(Y_2) \end{bmatrix}_3 \right)_2^{(n)} \\
\Psi^3 & \supseteq & \left(\begin{bmatrix} Y_1 = f(W) \\ Y_2 = f(W) \end{bmatrix}_4 \right)_3^{(o)} \\
\Psi^4 & \supseteq & ()_4
\end{array}$$

Figure 10.6: **Example Illustrating the Order of the Unfolding Steps**

tion):

$$\Psi^1 \quad \supseteq \quad \begin{pmatrix} [U_1 = f(X_1)]_2^{®} \\ [U_2 = Y_2]_3 \\ [U_1 \Longleftrightarrow U_2]_4)_1 \end{pmatrix}_1^{(q)}$$

Finally, cluster (o) is substituted into (m) to obtain the constraint

$$\Psi^1 \quad \supseteq \quad \begin{pmatrix} [U_1 = f(X_1)]_2^{®} \\ [U_2 = f(W)]_4 \\ [U_1 \Longleftrightarrow U_2]_4 \end{pmatrix}_1^{(q)}$$

The engine then terminates and outputs the constraints involving clusters in substitution form. Now, the only constraint output for Ψ^1 is the last constraint added involving cluster (q). Hence, we may conclude that U_1 is ground iff U_2 is ground.

Now, suppose that we allow the substitution of cluster (n) into (m), obtaining $\Psi^1 \supseteq ([U_1 = f(X_1)]^{®}, [U_2 = Y_2], [true])$, and then using (o) to substitute into this, we obtain $\Psi^1 \supseteq ([U_1 = f(X_1)]^{®}, [U_2 = Y_2]^{®}, [U_2 = f(W)], [true])$. Since the cluster in this constraint is in substitution form, it is output. Clearly it forbids the inference that U_1 is ground iff U_2 is ground. Intuitively, the problem here was that the substitution of (n) into (m) was performed before the information from (o) was used; this information could not be recovered later.

In order to prove correctness of the engine, it is necessary to first define the meaning of semantic constraints used by the engine. While the meaning of initial semantic constraints obtained from a program has already been outlined (see Definition 19), we now have to define the meaning of the clusters constructed by the algorithm, and in particular, deal with approximate groups.

If θ_1 and θ_2 are substitutions, then let $\theta_1 \circ \theta_2$, the composition of θ_1 and θ_1, be the substitution that maps X into $\theta_1(\theta_2(X))$ for all program variables X. Also, write $\theta_1 \leq \theta_2$ if there exists a substitution θ such that $\theta_2 = \theta \circ \theta_1$. Now, let *mgi* be a function that maps a sequence of substitutions $\theta_1, \ldots, \theta_n$ into a most general instance of the substitutions θ_i, if one exists. That is, $\theta_i \leq mgi(\theta_1, \ldots, \theta_n)$, $i = 1..n$, and for any other substitution θ, if $\theta_i \leq \theta$, $i = 1..n$ then $mgi(\theta_1, \ldots, \theta_n) \leq \theta$. For example, $mgi([X \mapsto f(W), Y \mapsto a]$, $[X \mapsto f(b)])$ is $[X \mapsto f(b), Y \mapsto a]$ (or some renaming thereof). Next, define a function *join* that essentially maps a set S of variables and a sequence of substitutions $\theta_1, \ldots, \theta_n$ into $mgi(\theta_1, \ldots, \theta_n)|_S$ after first renaming the variables in the domains of $\theta_1, \ldots, \theta_n$ to avoid variable conflicts. More formally, define $join(S, \theta_1, \ldots, \theta_n)$ by first picking a sequence of renamings $\theta'_1, \ldots, \theta'_n$ such that the sets of variables $\bigcup_{X \in S} var(\theta'_i(\theta_i(X)))$, $i = 1..n$, are disjoint. Then define $join(S, \theta_1, \ldots, \theta_n)$ to be $mgi(\theta'_1 \circ \theta_1, \ldots, \theta'_n \circ \theta_n)|_S$. For example, $join(S, [X \mapsto f(a), Y \mapsto W], [X \mapsto W])$ is equivalent to $mgi([X \mapsto f(a), Y \mapsto W_1]$, $[X \mapsto W_2])|_S$ where W_1 and W_2 are new variables. Finally, extend the *join* function to apply over sets: $join_S(\Theta_1, \ldots, \Theta_n) \stackrel{\text{def}}{=} \{join_S(\theta_1, \ldots, \theta_n) : \theta_i \in \Theta_i, i = 1..n\}$.

An *interpretation* $\mathcal{I}$ of a collection of semantic constraints is a mapping from each variable Ψ^α into a set of substitutions, denoted $\mathcal{I}(\Psi^\alpha)$. Such a mapping is extended to exact groups as follows:

$$\mathcal{I}([\tilde{s} = \tilde{t}]_\alpha) \stackrel{\text{def}}{=} \left\{ mgu\left(\tilde{s} = \left\| \theta(\tilde{t}) \right\|_{var(\tilde{s})}\right) : \theta \in \mathcal{I}(\Psi^\alpha) \right\}.$$

Similarly, $\mathcal{I}([\mathcal{A}]_\alpha) = \textsc{conc}(\mathcal{A})$. The interpretation of clusters employs *join* as follows:

$$\mathcal{I}\left((\mathcal{G}_1, \ldots, \mathcal{G}_n)\right)_\alpha \stackrel{\text{def}}{=} join(var(\alpha), \mathcal{I}(G_1), \ldots, \mathcal{I}(G_n)).$$

As usual, an interpretation $\mathcal{I}$ is a *model* of a collection of semantic constraints if $\mathcal{I}(\Psi_i) \supseteq \mathcal{I}(cl)$ for each constraints $\Psi_i \supseteq cl$ in the collection. A collection $\mathcal{C}$ of semantic constraints is guaranteed to have a least model (modulo variable renaming), and this shall be denoted by $lm(\mathcal{C})$. Finally, the correctness of the engine can be characterized as follows:

Theorem 11 (Correctness and Termination) *Let P be a program and let $\mathcal{D}$ be a finite*[4] *abstract domain. Then the unfolding engine terminates on P using $\mathcal{D}$, and the least model of the output semantic constraints is a conservative approximation of the collecting semantics of P.*

Proof Outline: Let $\mathcal{C}_P$ be the semantic constraints for P. The first part of the proof establishes that $lm(\mathcal{C}_P)$ corresponds to P's collecting semantics. The proof is essentially similar to the proof of correspondence between environment constraints and collecting semantics (see Chapter 4), although the intermediate step of defining a ground collecting semantics is not needed.

The next part of the proof establishes that the algorithm constructs a conservative approximation of $lm(\mathcal{C}_P)$. Let $\mathcal{C}_{loop}$ denote the constraints obtained after execution of the main **repeat-until** loop of the unfolding engine. Let $\mathcal{C}_{no-res}$ denote the constraints obtained by deleting the residual groups from $\mathcal{C}_{loop}$, and let $\mathcal{C}_{final}$ denote the final constraints, that is those obtained by selecting only the constraints involving substitution form clusters from $\mathcal{C}_{no-res}$. Now, the main loop of the algorithm merely serves to add constraints to $\mathcal{C}_P$. Moreover, the original constraints in $\mathcal{C}_P$ do not contain any residual expressions. Therefore, $\mathcal{C}_P$ is a subset of both $\mathcal{C}_{loop}$ and $\mathcal{C}_{no-res}$ and so $lm(\mathcal{C}_P) \subseteq lm(\mathcal{C}_{no-res})$.

The main part of the correctness proof establishes that the exhaustive application of the unfolding step generates sufficient substitution form clusters that these clusters alone characterize the least model of $\mathcal{C}_{no-res}$. In essence we prove a form of completeness of the unfolding step with respect to the least model. The proof of this requires reasoning about the specific formulation of the unfolding step.

Termination is proved by showing that there are only a finite number of different clusters that can be produced. This is essentially because during an unfolding step, the function *select* ensures that the size of terms appearing the new cluster are not larger than those appearing in the previous clusters. The main complications are (a) the part of the simplification step that deals with head only variables, since this step can increase the size of terms and introduce new variables, and (b) the renamings of equations during the unfolding step, since this may introduce new variables. A key part of the

[4]Some algorithms use domains satisfying a weaker condition such as the "finite chain" property. Our engine can in general be adapted to terminate on such domains. We omit the details for simplicity.

termination proof is that the number of new variables can be bounded. □

10.5 Variations of the Engine

The main parameter of the engine is the domain used for approximation. We now discuss two other parts of the engine where there is scope for variation.

Consider the *select* function that is used to curtail the growth of group equations. Another reasonable definition of *select* can be obtained by using a uniform bound on the size of terms, the so-called "depth-k" approximation. The important point is that the use of *any select* function does not affect correctness. It will, however, affect accuracy and termination. In general, using a smaller or more restrictive function will make the proof of termination easier (and can also enhance efficiency). Using a bigger function, on the other hand, is more accurate, but it complicates the termination proof in general.

Next consider the output of the engine. As defined, this consists of a collection of clusters in substitution form. Such a collection is an explicit definition, for each program rule, of a set of substitutions. However, to maximize the structural information obtained from the engine, residual groups should not be deleted upon termination. This however raises the problem that if clusters contain residual groups, then they are no longer an explicit representation of a set of substitutions. In fact, the sets of values defined by (the least model of) these equations are, in general, not decidable.

Perhaps the best general technique for extracting information from these residual groups is to approximate them by interpreting them as set constraints, and then employing the intersection component of the intersection-projection algorithm described in Chapter 7. It is important to note that the engine as stated already has much of the analytical power of the set constraint approach to analysis. Indeed the third program in Section 10.1 shows that it is sometimes strictly more accurate than the set constraint approach. By augmenting the engine output with the residual groups, and applying the intersection algorithm, the engine in fact becomes uniformly more accurate than the set constraint algorithm.

0. **initialize** the array *value*;
1. **repeat**
2. $\quad$ *oldvalue* := *value*;
3. $\quad$ **for each** rule in P with label α and body $\tilde{A}$
4. $\quad\quad$ $value[\alpha] := \bigcup_{\tilde{\beta}} \mathcal{D}\text{-}unify\left(var(\alpha), \tilde{A}, \tilde{B}^{\tilde{\beta}}, value[\tilde{\beta}]\right)$
5. $\quad$ **until** (*value* = *oldvalue*) **break**;
6. **output** *value*;

Figure 10.7: **The Standard Engine (Bottom-Up)**

10.6 Comparison with Abstract Interpretation

It is difficult to provide a uniform characterization of the many abstract interpretation based algorithms for logic program analysis that have appeared in the literature. This is because each algorithm is designed with different criteria. In particular, *ad hoc* approximations are often introduced for efficiency reasons. However, these differences are rarely fundamental from a conceptual point of view. Consequently, it is possible to define an idealized engine that encompasses the essence of the abstract interpretation technique underlying these algorithms, and is at least as accurate as any of them. We call this idealized engine the *standard* engine, and we now outline its definition.

In general, each analysis algorithm starts by associating a value from the chosen abstract domain with designated parts of the program. The operation of the algorithm then consists of repeatedly recomputing each value from the values previously computed. For a bottom-up analysis, an informal and simplified outline is given in Figure 10.7. The variables *value* and *oldvalue* are arrays indexed by rule labels. The sequence $\tilde{\beta}$ in line 4 ranges over all sequence of rule labels $\beta_1, \ldots, \beta_n$ such that the head of the rule with label β_i is compatible with the i^{th} atom of $\tilde{A}$ (the body of the rule with label α). $\tilde{B}^{\tilde{\beta}}$ denotes the sequence $B_1, \ldots, B_n$ such that B_i is the head of the rule with label β_i. $value[\tilde{\beta}]$ denote the sequence of abstract values $value[\beta_1], \ldots, value[\beta_n]$. Finally, the expression $\mathcal{D}\text{-}unify(var(\alpha), \tilde{A}, \tilde{B}^{\tilde{\beta}}, value[\tilde{\beta}])$ is intended to denote the result of abstractly unifying $\tilde{A}$ with the rule heads $\tilde{B}^{\tilde{\beta}}$, in the context of the abstract values $value[\tilde{\beta}]$, and restricting the variables in the

1. $\leftarrow eq(X,Y), p(f(Y,Z))$.
2. $p(U) \rightarrow q(U)$.
3. $q(f(W,V)) \rightarrow r(W)$.
4. $r(c)$.
5. $eq(S,S)$.

	initial	1	2	3	4
$value[1]$	*false*	*false*	*false*	*false*	$X \leftrightarrow Y$
$value[2]$	*false*	*false*	*false*	*true*	*true*
$value[3]$	*false*	*false*	W	W	W
$value[4]$	*false*	*true*	*true*	*true*	*true*
$value[5]$	*false*	*true*	*true*	*true*	*true*

Figure 10.8: **Example Execution of the Standard Engine**

resulting abstract value to $var(\alpha)$. More formally, if S is a finite set of program variables, $\tilde{A}$ denotes $A_1, \ldots, A_n$, $\tilde{B}$ denotes $B_1, \ldots, B_n$, and $\tilde{\mathcal{A}}$ denotes $\mathcal{A}_1, \ldots, \mathcal{A}_n$, then $\mathcal{D}$-*unify*$(S, \tilde{A}, \tilde{B}, \tilde{\mathcal{A}})$ is

$$\text{ABS}\left(S, \left\{ mgu \begin{pmatrix} A_1 = \|\theta(B_1)\|_{var(A_1)} \\ \cdots \\ A_n = \|\theta(B_n)\|_{var(A_n)} \end{pmatrix} : \theta_i \in \text{CONC}(\mathcal{A}_i), i = 1..n \right\}\right).$$

Figure 10.8 contains an example execution of the standard engine. The abstract domain used here is $\mathcal{D}_{prop}$. Note that the i^{th} column describes *value* after i iterations of the main loop of the standard engine.

We can also define a *standard engine* for top-down analysis. An informal outline of this is given in Figure 10.9. The variables *value* and *oldvalue* are arrays indexed by labels (including rule labels and body atom labels). The sequence $\tilde{\beta}$ in line 5 ranges over all sequence of labels $\beta_0, \beta_1, \ldots, \beta_j$ such that β_0 is a body atom label and the body atom with label β_0 is compatible with A_0, and each β_i, $i \geq 1$, is a rule label such that the head of the rule with label β_i is compatible with A_i. $\tilde{B}^{\tilde{\beta}}$ denotes the sequence $B_0, \ldots, B_j$ such that B_0 is the body atom with label β_0, and each B_i, $i \geq 1$, is the head of the rule with label β_i. $value[\tilde{\beta}]$ denote the sequence of abstract values $value[\beta_0], values[\beta_1], \ldots, value[\beta_j]$. We note that the usual notions of computing the *call* and *return* substitutions for each rule, are subsumed in the standard engine by the computation of the substitutions encountered at each program point.

Both the bottom-up and top-down standard engines are parameterized by an abstract domain $\mathcal{D}$. These two engines capture the essence of most of the abstract interpretation algorithms in the literature. For example, the bottom-up engine applies in the case of [27, 43], and the top-down engine

initialize the array *value*;
do
  $value := newvalue$;
  for each rule with label α_{n+1}, head A_0 and body $A_1^{\alpha_1}, \ldots, A_n^{\alpha_n}$
    for $j = 0..n$
      let $\tilde{A}$ denote $A_0, A_1, \ldots, A_j$;
      $newvalue[\alpha_{j+1}] := \bigcup_{\tilde{\beta}} \mathcal{D}\text{-}unify\left(var(\alpha), \tilde{A}, \tilde{B}^{\tilde{\beta}}, value[\tilde{\beta}]\right)$
  until ($value = oldvalue$)
output *value*;

Figure 10.9: **The Standard Engine (Top-Down)**

applies in the case of [11, 15, 51].

In order to compare the standard engine with the unfolding engine, we will now formulate the operation of the standard engine using our framework of group equations, clusters and cluster substitution. Recall that in our framework, we can perform bottom-up or top-down analysis simply by using the appropriate semantic constraints. Now, consider modifying the unfolding engine to mimic the behavior of the standard engine as follows:

- all terminal groups $[\tilde{s} = \tilde{t}]$ in the semantic constraints for P are first replaced by $\text{ABS}(var(\tilde{s}), mgu(\tilde{s} = \tilde{t}))$;
- the unfolding step is simplified so that no residual groups are produced, and this can achieved simply by redefining $\widehat{[\mathcal{E}]}_\alpha = [\ ]_\alpha$.

The main effect of these changes is that clusters in substitution form contain only approximate groups (and no terminal or residual groups). Hence the only operation required during unfolding is the composition $[\mathcal{E}] \circ [\mathcal{A}]$ of an exact and approximate group. Call this the resulting engine the *restricted* unfolding engine. Now, if $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are abstract values, then let $\mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_n$ denote the join of these abstract value (this can be easily formalized using *mgi*, ABS and CONC). Using this notation, we can state the correspondence between the restricted unfolding engine and the standard engine as follows:

Proposition 45 *For all programs P, if the restricted unfolding engine out-*

puts semantic constraints $\mathcal{C}$ and the standard engine outputs the array value, then for each relevant label α

$$value(\alpha) \;=\; \bigcup\{\mathcal{A}_1 \wedge \cdots \wedge \mathcal{A}_n : \Psi^{\alpha} \supseteq ([\mathcal{A}_1]_{\alpha_1}, \ldots, [\mathcal{A}_n]_{\alpha_1})_{\alpha} \text{ appears in } \mathcal{C}\}.$$

□

Clearly the modified unfolding steps of the restricted unfolding engine are less accurate than the unfolding step used in the (normal) unfolding engine. Hence:

Theorem 12 *The unfolding engine is uniformly more accurate than the standard engine in the sense: given a finite abstract domain, the output of the unfolding engine, for any program, is more accurate than that of the standard engine.* □

An alternative view of the differences between the standard and unfolding engines is as follows. The standard engine implements abstract interpretation by a successive iteration starting from some initial values $\mathcal{A}_0$ and computing the sequence $\mathcal{A}_0 \xrightarrow{\mathcal{F}} \mathcal{A}_1 \xrightarrow{\mathcal{F}} \mathcal{A}_2 \xrightarrow{\mathcal{F}} \mathcal{A}_3 \cdots$ where $\mathcal{F}$ represents the abstract semantic function at hand and the $\mathcal{A}_i$ are the successive abstract values being computed. The unfolding engine, on the other hand, uses a *changing* abstract semantic function and computes $\mathcal{A}_0 \xrightarrow{\mathcal{F}_0} \mathcal{A}_1 \xrightarrow{\mathcal{F}_1} \mathcal{A}_2 \xrightarrow{\mathcal{F}_2} \mathcal{A}_3 \cdots$. Such a change takes place whenever an unfolding step is applied. The important point is that each function $\mathcal{F}_i$ is at least as accurate as the original function $\mathcal{F}$.

We conclude this section with the realization of some analysis algorithms using our engine. Let UNFOLD($\mathcal{D}$) denote the algorithm obtained by using the unfolding engine and the abstract domain $\mathcal{D}$; similarly for STANDARD($\mathcal{D}$). Let $\mathcal{D}_{sharing}$ denote the domain described in [27].

Corollary 2 *For groundness analysis,* UNFOLD($\mathcal{D}_{prop}$) *is uniformly more accurate than* [44][5]. □

Corollary 3 *For sharing analysis,* UNFOLD($\mathcal{D}_{sharing}$) *is uniformly more accurate than* [27]. □

[5] By [12], *UNFOLD*($\mathcal{D}_{prop}$) is uniformly more accurate than a number of other algorithms as well.

10.7 Discussion

We have presented a new engine for analysis of logic programs. Its starting point is a specific formulation of semantic constraints of a program. We have illustrated how these constraints can model bottom-up and top-down execution. Using a given abstract domain $\mathcal{D}$, the engine performs unfolding on these constraints. Each unfolding is augmented with a notion of approximation, defined in terms of $\mathcal{D}$, to curtail the expansion of terms. The main advantage of our specific technique of unfolding, which uses the key concept of residual groups, is that there is a uniform and accurate treatment of structural information. By formulating the standard analysis engine as a special case of the unfolding engine, we then showed that the unfolding engine is uniformly more accurate than the standard engine.

Yet to be addressed is the issue of efficiency. This has two main aspects. First is the initial number of clusters in the constraints. This number is bounded by the number of sequences of head atoms that match the sequence of body atoms, for each rule. The second part concerns the unfolding step. Though the number of new groups produced by this step is linear in the size of the input clusters, the termination of the engine is based on an exhaustive application of this step. Experience with the prototype implementation described in Chapter 8 provides some evidence that these two problems can be overcome. First, the issue of matching body and head atoms in set constraints is essentially the same as in the unfolding engine, and this has not proven to be prohibitive. Next consider the issue of unfolding efficiency. In set constraints, there is a notion of substitution that is similar to unfolding in terms of the number of new expressions generated (but whose semantics is very different). The implementation exploits the crucial fact that the substitution step gives rise to mostly *redundant* expressions. Because this implementation has shown promise, we expect that the currently planned implementation of the unfolding engine can be engineered to be practical.

We conclude by observing that the notion of combining set constraint techniques with an ability to reason about inter-variable dependencies is similar in spirit to some recent extensions of tree automata that allow a limited form of equality, such as the Bogaert and Tison's *tree automata with equality* [9] and the closely related *shallow set constraints* of Uribe [65]. Our work differs in the use of more general (although less uniform) notions of equality, and moreover, the use of abstract domains to capture information about dependencies that is tailored to the desired program analysis.

Chapter 11

Analysis of Functional Languages

We now show how set constraints may be used to analyze functional languages. In particular, we outline extensions to the basic set constraint calculus for higher-order functions as well as references and assignment. Importantly, these extensions represent a natural extension of the idea of treating variables as sets. Each extension has a simple definition and leads to an intuitive notion of program approximation. Moreover, the key advantage of the set constraint approach – accurate and uniform treatment of data structures – is preserved.

This material in this chapter is very preliminary and informal. No proofs of correctness are given. However, many of the ideas have been implemented, and we shall describe some of the results of this effort. We also discuss the close connections between the use of set constraints to analyze functional languages and subtype systems. In particular, we propose formalizing our analysis as an "optimal" system of simple subtypes.

11.1 Higher-Order Set Constraints

To analyze functional languages such as Standard ML [46], set constraints must be extended with a mechanism to analyze higher-order functions. In essence, this is achieved by the addition of three new components. First, a new set operator *apply* is introduced. Second, a new collection of function symbols are introduced, and these shall be used to denote the functions defined in a program (as oppose the program's data constructors). This new set of symbols shall be denoted by $\mathcal{F}$, and is assumed to be disjoint from Σ, the set of data constructors. Third, for each symbol $F \in \mathcal{F}$, two new variables $dom(F)$ and $ran(F)$ shall be introduced to respectively capture the domain and range of the function F.

As usual, an interpretation of a collection of set constraints is a mapping from each set variable (including $dom(F)$ and $ran(F)$, $F \in \mathcal{F}$) into sets of values. Values are defined to be either symbols from $\mathcal{F}$ or Σ, or of the form $f(v_1, \ldots, v_n)$ such that f is an n-ary symbol from Σ and each v_i is a value. An interpretation is extended to map from set expressions into sets of values as follows:

- $\mathcal{I}(f(se_1, \ldots, se_n)) \stackrel{\text{def}}{=} \{f(v_1, \ldots, v_n) : v_i \in \mathcal{I}(se_i), i = 1..n\}$, $f \in \Sigma$;
- $\mathcal{I}(F) = \{F\}$, $F \in \mathcal{F}$;
- $\mathcal{I}(f^{-1}_{(i)}(se)) \stackrel{\text{def}}{=} \{v_i : f(v_1, \ldots, v_n) \in \mathcal{I}(se_i)\}$;
- $\mathcal{I}(se_1 \cap \cdots \cap se_n) \stackrel{\text{def}}{=} \{v : v \in \mathcal{I}(se_i), i = 1..n\}$;
- $\mathcal{I}(apply(se_1, se_2)) \stackrel{\text{def}}{=} \left\{ f(v) : \begin{array}{c} f \in \mathcal{I}(se_1) \\ v \in \mathcal{I}(se_2) \end{array} \right\} \cup \bigcup_{F \in \mathcal{I}(se_1)} ran(F)$,
  provided $\mathcal{I}(dom(F)) \supseteq \mathcal{I}(se_2)$, $F \in \mathcal{I}(se_1)$;

The critical part of the interpretation is $\mathcal{I}(apply(se_1, se_2))$, which consists of two parts. In essence, the first part corresponds to application involving data structures, and the second part corresponds to the applications involving functions defined in the program. The side condition on this definition states that the set of values that each function F is applied to must be contained in the set $dom(F)$. Note that $\mathcal{I}$ is now a partial function that is

let fun *id* X = X **in** *id c* **end**	$\mathcal{X} \supseteq dom(id)$ $ran(id) \supseteq \mathcal{X}$ $\mathcal{E} \supseteq apply(id, c)$

Figure 11.1: **Example Functional Program and Set Constraints**

defined on a set expression se if $\mathcal{I}$ is defined on all (proper) subexpressions of se. An interpretation $\mathcal{I}$ is a *model* of a collection of constraints if, for each constraint $\mathcal{X} \supseteq se$ in the collection, $\mathcal{I}(se)$ is defined and $\mathcal{I}(\mathcal{X}) \supseteq \mathcal{I}(se)$ (and similarly for constraints $\mathcal{X} = se$ in the collection).

We now outline how such constraints may be used to analyze functional programs, using a fragment of ML. We shall assume that bound variables are renamed so that each bound variable is distinct. Now, consider the program P in Figure 11.1, in which c is a data constructor of arity 0. We distinguish between bound variables that represent functions (such as id) and other bound variables (such as X); we call the latter *program variables* (note that program variables can take functions as values). For each program variable we introduce a set variable whose purpose is to capture the possible values of the program variable. We shall again use letters $X, Y, \ldots$ to denote program variables and $\mathcal{X}, \mathcal{Y}, \ldots$ for the corresponding set variables.

The constraints for the program in Figure 11.1 are constructed as follows. Corresponding to the first line of the program (the definition of id), we introduce two constraints. The first is $\mathcal{X} \supseteq dom(id)$ which captures the fact that the values for X must include all of the possible values with which id may be called. The second constraint is $ran(id) \supseteq \mathcal{X}$, and this reflects the fact that the return values for id must contain all possible values for X. Finally, the third constraint $\mathcal{E} \supseteq apply(id, c)$ corresponds to the application $(id\ c)$. The set variable $\mathcal{E}$ is introduced to capture the value of the whole program. The least model of these three constraints is

$$\begin{aligned} \mathcal{X} &\mapsto \{c\} \\ dom(id) &\mapsto \{c\} \\ ran(id) &\mapsto \{c\} \\ \mathcal{E} &\mapsto \{c\} \end{aligned}$$

which exactly captures the run-time behavior of the program.

let fun *id* X = X
in
  (*id* b, *id* c)
end

$$\begin{aligned} \mathcal{X} &\supseteq dom(id) \\ ran(id) &\supseteq \mathcal{X} \\ \mathcal{E} &\supseteq (apply(id, b), apply(id, c)) \end{aligned}$$

Figure 11.2: **Example Approximation of a Functional Program**

However, the set constraints are not always exact. In general, the least model of the set constraints for a program describe a conservative approximation of the possible run-time values of variables. This is because the set constraints ignore dependencies between variables and dependencies between the domain and codomain of a function. The latter is achieved through the use of the variables $dom(F)$ and $ran(F)$, which respectively collect the values of the domain and codomain of the function F. To illustrate this, consider the program and constraints in Figure 11.2. The least model of these constraints is

$$\begin{aligned} \mathcal{X} &\mapsto \{b, c\} \\ dom(id) &\mapsto \{b, c\} \\ ran(id) &\mapsto \{b, c\} \\ \mathcal{E} &\mapsto \{(b, b), (b, c), (c, b), (c, c)\} \end{aligned}$$

and so $\mathcal{E}$, which consists of four pairs, defines an approximation of the program.

Before giving further example constraints, we note that, for convenience, we assume that each function is "named". Hence, terms containing anonymous functions such as

$((fn\ X \Rightarrow X)\ (fn\ Y \Rightarrow Y))$

must first be rewritten into **let fun** $F\ X = X$ **and** $G\ Y = Y$ **in** $(F\ G)$ **end** before they are analyzed.

11.2 Examples

Consider the program in Figure 11.3, involving the map function. We use *nil* and *cons* to denote the data constructors for lists. Before discussing the constraints for this program, first note that in ML, all data constructors

$$
\begin{array}{l}
\textbf{let fun } map(F,\ cons(X,\ L)) \ = \ cons(F\ X,\ map(F,\ L)) \\
\qquad\quad \mid\ map\ \_ \ = \ nil \\
\quad \textbf{and } id\ \ Y \ = \ Y \\
\textbf{in} \\
\quad map(id,\ cons(1,\ cons(2,\ cons(3,\ nil)))) \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{rcl}
ran(map) & \supseteq & cons((apply(\mathcal{F},\ \mathcal{X}),\ apply(map,\ (\mathcal{F},\ \mathcal{L})))) \\
ran(map) & \supseteq & nil \\
\mathcal{F} & \supseteq & \{F : (F,\ cons((X,\ L))) \in dom(map)\} \\
\mathcal{X} & \supseteq & \{X : (F,\ cons((X,\ L))) \in dom(map)\} \\
\mathcal{L} & \supseteq & \{L : (F,\ cons((X,\ L))) \in dom(map)\} \\
ran(id) & \supseteq & \mathcal{Y} \\
\mathcal{Y} & \supseteq & dom(id) \\
\mathcal{E} & \supseteq & apply\Big(map,\ (id,\ cons((1,\ cons((2,\ cons((3,\ nil)))))))\Big)
\end{array}
$$

Figure 11.3: **Map Program and Its Set Constraints**

and functions are unary. Non-unary data constructors and functions are obtained using tupling operations. For example, the program term $cons(3, nil)$ is in fact an application of *cons* to the pair $(3, nil)$. Hence, the set expression corresponding to this program term is $apply(cons, (3, nil))$. However, this is equivalent to the set expression $cons((3, nil))$, that is, the set expression $cons(se)$ where se is $(3, nil)$. The set constraints in Figure 11.3 use this simpler formulation for program terms involving applications of data constructors.

Now, consider the definition of the function *map*. This contains two match rules: $map(F,\ cons(X,\ L)) \ = \ cons(F\ X,\ map(F,\ L))$ and $map\ \_ =$ nil. Each match rule generates one constraint for $ran(map)$. Also, the first match rule generates bindings for F, X and L. The values for these variables are given by considering all environments ρ such that $\rho((F,\ cons(X,\ L))) \in$ $dom(map)$. Specifically, the values for F, X and Y are given by

$$\{\rho(F) : \rho((F,\ cons(X,\ L))) \in dom(map)\}$$
$$\{\rho(X) : \rho((F,\ cons(X,\ L))) \in dom(map)\}$$
$$\{\rho(L) : \rho((F,\ cons(X,\ L))) \in dom(map)\}$$

Recalling the definition of quantified expressions, these are exactly the condition imposed by the constraints on $\mathcal{F}$, $\mathcal{X}$ and $\mathcal{L}$ respectively.

More generally, match and case expressions may contain an arbitrary number of match rules. To model the sequential nature of the execution of match conditions, we shall use complement constants. For example, consider the following case statement, which is assumed to appear inside the scope of the variable L:

case L **of**
  $cons(U, cons(V, W)) \Rightarrow s_1$
  $\mid\ cons(X, Y) \Rightarrow s_2$
  $\mid\ nil \Rightarrow s_3$

We briefly review the execution of such a statement. First, an attempt is made to match the value of L against $cons(X, cons(Y, Z))$. If this match succeeds, then the values obtained for U, V and W are used in the evaluation of s_1. If the match fails, then an attempt to match L against $cons(X, Y)$ is made. Again, if the match succeeds, then the values obtained for X and Y are used in the evaluation of s_2. If the match fails, then the value of L is matched against nil (and assuming that L is a list, this will always succeed), and s_3 is evaluated. Now, suppose that $\mathcal{U}, \mathcal{V}, \mathcal{W}, \mathcal{X}, \mathcal{Y}$ and $\mathcal{L}$ are the set variables corresponding to U, V, W, X, Y and L respectively, and consider the following constraints for modeling the possible variable bindings resulting from execution of the case statement:

$$\mathcal{U} \supseteq \{U : cons((U, cons((V, W)))) \in \mathcal{L}\}$$
$$\mathcal{V} \supseteq \{V : cons((U, cons((V, W)))) \in \mathcal{L}\}$$
$$\mathcal{W} \supseteq \{W : cons((U, cons((V, W)))) \in \mathcal{L}\}$$
$$\mathcal{X} \supseteq \left\{X : cons((X, Y)) \in \mathcal{L} \cap \overline{cons((\top, cons((\top, \top))))}\right\}$$
$$\mathcal{Y} \supseteq \left\{Y : cons((X, Y)) \in \mathcal{L} \cap \overline{cons((\top, cons((\top, \top))))}\right\}$$

Note that in the constraints corresponding to the binding of X and Y, the expression $cons((X, Y))$ is matched against values in the set described by

$\mathcal{L} \cap \overline{cons((\top, cons((\top, \top))))}$, that is, the set of values in $\mathcal{L}$ that do not match the pattern $cons((U, cons((V, W))))$.

To illustrate the construction of constraints corresponding to case statement, consider the program in Figure 11.4, which computes the disjunctive normal form of a propositional formula. The propositional constants are $z0, \ldots z9$, and o, a and n denote and, or and not respectively. Figure 11.4 also contains the constraints corresponding to the definition of the function *dnf*.

11.3 Extensions and Variations

The basic approach of using constraints to analyze function programs can be extended in a number of ways. We first consider the imperative components of the ML language, namely references and assignment. These can be modeled as follow: for each expression $ref(t)$ that appears in the program to be analyzed, a distinct new constant c_{ref} and a corresponding new variable $val(c_{\mathrm{ref}})$ are introduced. The new variable $val(c_{\mathrm{ref}})$ is used to capture the possible values assigned to *ref* cells generated by this occurrence of *ref*. Each introduced constant c_{ref} is called a *reference constant*. Figure 11.5 gives an example program, its set constraints and the least model of the set constraints. Three new kinds of set operators are employed. The meaning of each operator is defined as follows:

- $\mathcal{I}(seq(se_1, \ldots, se_n)) \stackrel{\mathrm{def}}{=} \begin{cases} \mathcal{I}(se_n) & \text{if } \mathcal{I}(se_i) \neq \{\} \text{ for each } i < n \\ \{\} & \text{if } \mathcal{I}(se_i) = \{\} \text{ for some } i < n \end{cases}$
- $\mathcal{I}(assign(se_1, se_2)) \stackrel{\mathrm{def}}{=} \{()\}$ provided $val(c_{\mathrm{ref}}) \supseteq \mathcal{I}(se_2)$ for each reference constant $c_{\mathrm{ref}} \in \mathcal{I}(se_1)$.
- $\mathcal{I}(deref(se)) \stackrel{\mathrm{def}}{=} \bigcup_{c_{\mathrm{ref}} \in \mathcal{I}(se)} val(c_{\mathrm{ref}})$ where c_{ref} ranges over reference constants.

There is also considerable scope for varying the constraints themselves. In particular, the procedure for constructing constraints outlined in the previous section associates two variables, $dom(F)$ and $ran(F)$ for each function F in the program. This means that each function is analyzed once in the

$$
\begin{array}{l}
\textbf{datatype}\ \mathit{formulas} = z0 \mid z1 \mid z2 \mid z3 \mid z4 \mid z5 \mid z6 \mid z7 \mid z8 \mid z9 \\
\quad \mid\ o\ \textbf{of}\ \mathit{formulas} \star \mathit{formulas} \\
\quad \mid\ a\ \textbf{of}\ \mathit{formulas} \star \mathit{formulas} \\
\quad \mid\ n\ \textbf{of}\ \mathit{formulas}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{fun} & \mathit{dnf}(o(X1, Y1)) = o(\mathit{dnf}(X1), \mathit{dnf}(Y1)) \\
\quad \mid & \mathit{dnf}(a(X2, Y2)) = \mathit{norm}(a(\mathit{dnf}(X2), \mathit{dnf}(Y2))) \\
\quad \mid & \mathit{dnf}(n(X3)) = \mathit{norm}(n(\mathit{dnf}(X3))) \\
\quad \mid & \mathit{dnf}(X4) = X4 \\
\textbf{and} & \mathit{norm}(a(o(A1, B1), C1)) = o(\mathit{norm}(a(A1, C1)), \mathit{norm}(a(B1, C1))) \\
\quad \mid & \mathit{norm}(a(C2, o(A2, B2))) = o(\mathit{norm}(a(C2, A2)), \mathit{norm}(a(C2, B2))) \\
\quad \mid & \mathit{norm}(n(o(A3, B3))) = \mathit{norm}(a(\mathit{dnf}(n(A3)), \mathit{dnf}(n(B3)))) \\
\quad \mid & \mathit{norm}(n(a(A4, B4))) = o(\mathit{dnf}(n(A4)), \mathit{dnf}(n(B4))) \\
\quad \mid & \mathit{norm}(n(n(A5))) = A5 \\
\quad \mid & \mathit{norm}(n(A7)) = n(A7) \\
\quad \mid & \mathit{norm}(a(A8, A9)) = a(A8, A9)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ran}(\mathit{dnf}) & \supseteq & o((\mathit{apply}(\mathit{dnf},\ \mathcal{X}1),\ \mathit{apply}(\mathit{dnf},\ \mathcal{Y}1))) \\
\mathit{ran}(\mathit{dnf}) & \supseteq & \mathit{apply}(\mathit{norm},\ a((\mathit{apply}(\mathit{dnf},\ \mathcal{X}2),\ \mathit{apply}(\mathit{dnf},\ \mathcal{Y}2)))) \\
\mathit{ran}(\mathit{dnf}) & \supseteq & \mathit{apply}(\mathit{norm}, n(\mathit{apply}(\mathit{dnf}, \mathcal{X}3))) \\
\mathit{ran}(\mathit{dnf}) & \supseteq & \mathcal{X}4 \\
\mathcal{X}1 & \supseteq & \{X1 : o((X1,\ Y1)) \in \mathit{dom}(\mathit{dnf})\} \\
\mathcal{Y}1 & \supseteq & \{Y1 : o((X1,\ Y1)) \in \mathit{dom}(\mathit{dnf})\} \\
\mathcal{X}2 & \supseteq & \left\{X2 : a((X2,\ Y2)) \in \mathit{dom}(\mathit{dnf}) \cap \overline{o((\top,\ \top))}\right\} \\
\mathcal{Y}2 & \supseteq & \left\{Y2 : a((X2,\ Y2)) \in \mathit{dom}(\mathit{dnf}) \cap \overline{o((\top,\ \top))}\right\} \\
\mathcal{X}3 & \supseteq & \left\{X3 : n(X3) \in \mathit{dom}(\mathit{dnf}) \cap \overline{o((\top,\ \top))} \cap \overline{a((\top,\ \top))}\right\} \\
\mathcal{X}4 & \supseteq & \left\{X4 : X4 \in \mathit{dom}(\mathit{dnf}) \cap \overline{o((\top,\ \top))} \cap \overline{a((\top,\ \top))} \cap \overline{n(\top)}\right\}
\end{array}
$$

Figure 11.4: **DNF Program and Selected Constraints**

let fun $id\ X = (1; X)$
  and $f\ Y = 1;$
  val $Z = ref(id)$
in
  $id\ 2;\ Z := f;\ (!Z)\ 3$
end

$$\begin{aligned} ran(id) &\supseteq seq(1, \mathcal{X}) \\ \mathcal{X} &\supseteq dom(id) \\ ran(f) &\supseteq 1 \\ \mathcal{Z} &\supseteq c_{\mathrm{ref}} \\ val(c_{\mathrm{ref}}) &\supseteq id \\ \mathcal{E} &\supseteq seq(id\ 2, assign(\mathcal{Z}, f), deref(\mathcal{Z})) \end{aligned} \qquad \begin{aligned} ran(id) &\mapsto \{1,2,3\} \\ \mathcal{X} &\mapsto \{1,2,3\} \\ ran(f) &\mapsto 1 \\ \mathcal{Z} &\mapsto \{c_{\mathrm{ref}}\} \\ val(c_{\mathrm{ref}}) &\mapsto \{id, f\} \\ \mathcal{E} &\mapsto \{1,2,3\} \end{aligned}$$

Figure 11.5: **Analysis of References**

sense that there is no attempt to separately analyze the function on different inputs. For example, in the analysis of the program

let fun $id\ X = X$
in
  $(id\ 1,\ id\ nil)$
end

the two uses of id are combined, and so in the least model $dom(id)$ is $\{1, nil\}$. Hence the result of the program is approximated by $\{(1,1),\ (1, nil),\ (nil, 1),\ (nil, nil)\}$. In essence, the analysis is monomorphic.

It is natural to consider modifying the constraints generated from a program to facilitate some degree of *poly-variant* analysis in which functions are analyzed for a number of different calls. The main issue is how to introduce and control the notion of different calls. One method considers each occurrence of each function in a program, and separately analyses each function occurrence. For example, in the above program, the two occurrences of id in $(id\ 1,\ id\ nil)$ would be analyzed independently; the domain of the first would be $\{1\}$ and the domain of the second would be $\{nil\}$. Using this scheme, the result of the program is approximated by $\{(1, nil)\}$.

We observe that there is a close analogy between the use of function occurrences and the notion of polymorphism embodied in the polymorphic let typing rule for ML. Taking this analogy a step further, it is in principle possible to extract set constraints from a **let** bound expression, simplify these constraint and then appropriately generalize them, just as the type inference rule for **let** in ML allows (certain) type variables in the type expression obtained for a **let** bound expression to be generalized. Such an approach may provide a basis for implementing separate analyze of distinct function occurrences. We also note that the general problem of introducing and controlling the notion of "different calls" arises in a number of contexts in type theory. For example, in the intersection type discipline [64], type inference procedures must limit the number of different types that a function is given. See, for example, [17, 53].

11.4 Implementation

The main modifications to the set constraint algorithm for solving the new kinds of constraints introduced in this chapter are new transformations to simplify the operations *apply*, *seq*, *assign* and *deref*. In essence, these new transformations can be stated as follows:

- If $\mathcal{C}$ contains $\mathcal{X} \supseteq \mathit{apply}(F, a)$ where $F \in \mathcal{F}$, then output $\mathit{dom}(F) \supseteq a$ and $\mathcal{X} \supseteq \mathit{ran}(F)$.
- If $\mathcal{C}$ contains $\mathcal{X} \supseteq \mathit{seq}(a_1, \ldots, a_n)$ and $\mathit{lm}(\mathit{explicit}(\mathcal{C}))(a_i) \neq \{\}$ for $i < n$, then output $\mathcal{X} \supseteq a_n$.
- If $\mathcal{C}$ contains $\mathcal{X} \supseteq \mathit{assign}(c_{\mathrm{ref}}, a)$ then output $\mathit{val}(c_{\mathrm{ref}}) \supseteq a$ and $\mathcal{X} \supseteq ()$.
- If $\mathcal{C}$ contains $\mathcal{X} \supseteq \mathit{deref}(c_{\mathrm{ref}})$ then output $\mathcal{X} \supseteq \mathit{val}(c_{\mathrm{ref}})$.

where $a, a_1, \ldots, a_n$ are non-variable atomic expressions.

The prototype set constraint implementation described in Chapter 8 has been extended to include these operators. Very preliminary results suggest that the cost of solving set constraints from functional programs is not substantial and is comparable to type inference in the Standard ML of New Jersey compiler. Table 11.4 presents some benchmark results, again all

	time	equations
dnf	0.70	25+95
binary_add	0.19	26+49
poly_binary_add	1.09	246+287

Table 11.1: **Preliminary Results for Three Functional Programs**

times are in seconds on a Sun Sparc 1+ (24MB) running Mach and using version 0.75 of Standard ML of New Jersey. Entries in the second column of the table are of the form $x + y$, where x is the number of equations initially generated from the program, and y is the number of equations added during the execution of the algorithm. The dnf benchmark is based on the program that appears in Figure 11.4. The binary_add program is a simple 12 line program for adding numbers in binary representation. The poly_binary_add is a program that simulates the poly-variant analysis in which different occurrences of a function are analyzed separately. This is done by making a separate copy of the body of each function for each occurrence of that function. The program consists of about 130 lines. We remark that the results in Table 11.4 are very preliminary, and no effort has been put into optimization of the functional analysis part of the implementation. It is expected that the results can be substantially improved.

The analysis of functional programs appears to be substantially faster than for comparable logic programs. The main reason for this is that the intersection operation is not heavily used in set constraints arising from functional programs. In fact, the main place where intersection is used is in the analysis of case statements, and these intersection are usually of a very simple form. For many programs (including the dnf, binary_add and poly_binary_add benchmarks), the uses of intersection can be solved without introducing new variables. The set based analysis of such programs can be performed in polynomial time.

We conclude with an example of the results of the analysis of a program. Recall the dnf program, which appears in Figure 11.4. Suppose we wish to approximate the result of applying the function *dnf* to the program variable Q, where Q is bound to some unknown formula (that is, Q is some value constructed from o, a, n and the propositional constants $z0, \ldots, z9$). The output of the set constraint algorithm for this analysis is

$$\begin{aligned}
\mathcal{E} &= z0 \cup z1 \cup z2 \cup z3 \cup z4 \cup z5 \cup z6 \cup z7 \cup z8 \cup z9 \cup o((\mathcal{E},\mathcal{E})) \cup a((\mathcal{X},\mathcal{X})) \cup n(\mathcal{Y})\\
\mathcal{X} &= z0 \cup z1 \cup z2 \cup z3 \cup z4 \cup z5 \cup z6 \cup z7 \cup z8 \cup z9 \cup a((\mathcal{X},\mathcal{X})) \cup n(\mathcal{Y})\\
\mathcal{Y} &= z0 \cup z1 \cup z2 \cup z3 \cup z4 \cup z5 \cup z6 \cup z7 \cup z8 \cup z9
\end{aligned}$$

where $\mathcal{E}$ is the set variable that captures the values of $(dnf\ Q)$. Note that this defines $\mathcal{E}$ to be exactly the set of formulas in disjunctive normal form.

11.5 Discussion

The main aspect of the set based analysis of functional programs that has not been addressed is correctness. We are currently investigating a direct proof of this using an operational semantics, as well as an alternative method using connections with simple subtype systems [47]. In particular, there appears to be an intuitive characterization of the results of the set based analysis of a program in terms of an *optimal* system of simple subtypes. To see why this is so, observe that in a system of simple subtypes, the "accuracy" of the types obtained for a program depends intimately on the structure of the underlying base types. As more base types are added, the ability of the type system to distinguish between different program terms is enhanced. This situation is analogous to that of abstract interpretation (see the discussion in Section 5.6, page 133). Moreover, just as set based analysis can be characterized as an optimal abstract interpretation that ignores inter-variable dependencies, so it seems natural to seek a characterization of the set based analysis of functional programs as an *optimal* subtype system. We also note that there are intriguing connections between the kinds of transformations used to solve the set operator *apply* (see Section 11.4) and the algorithm described by Mitchell [47] for type inference in simple subtype systems.

We now briefly outline the related literature. One of the early uses of constraints in the context of functional languages is by Mishra and Reddy, who described the use of (an extended notion of) regular trees for generating types for untyped functional programs in [49]. The type system described is discriminative in the sense that it is based on sets of terms that are cartesian closed (specifically, the sets of terms are closed under the $\star$ operator described in Section 5.6, page 134).

Subsequently, Aiken and Murphy described a type inference algorithm for FL in [3]. This system is also based on the use of regular trees for the representation of types, although the restriction to discriminative types is omitted. In essence, the focus of both works is on obtaining *some* conservative approximation of a program. However, it is usually difficult to understand exactly what approximation will be computed for a given program. In particular, the algorithms employed often introduce *ad hoc* approximations as they execute. For example [3] uses a heuristic for dealing with recursion, and although this ensures that the algorithm terminates, it results in an unpredictable loss of information.

In contrast, our analysis for functional programs seeks to extend the set based analysis philosophy of constructing simple constraints that have a straightforward connection with the program at hand, and then solving these constraints exactly. An important motivation for this is the desire to obtain a natural and intuitive notion of program approximation that can form the basis of an expressive subtype system.

More closely related to our work is the algorithm for analysis of higher order functional programs by Jones [31]. Our work differs from [31] in two main respects. First, we analyze a strict language, whereas [31] considers a lazy language. Second, and perhaps more importantly, we have used constraints instead of grammars. The advantage of constraints is that they provide a simple and intuitive characterization of the analysis that is independent of algorithmic considerations. In contrast, the analysis in [31] is characterized only by a somewhat intricate grammar rewriting algorithm. Moreover, by using constraints, connections with other type systems (particularly subtype systems) are more apparent.

Very recent work by Aiken and Wimmers [6] is also closely related to ours. However, their formulation of constraints over ideals appears to differ substantially from our approach, although the precise connections are unclear at this stage.

We finally observe that implicit in set based analysis is a form of minimal function graph computation or control flow analysis [35, 57]. In particular, the least model of the set constraints provides a conservative approximation of the possible function calls at each call site in the program. The notion of control flow analysis embodied in essentially the same as the Shiver's 0^{th} order control flow analysis of [57], although the accuracy of the information we obtain is somewhat more accurate because of the improved treatment of structures in set based analysis (in particular, a function can be stored in a data structure and then recovered at some later stage).

Chapter 12

Conclusions

We briefly summarize the central ideas of the thesis and discuss some of the strengths and weakness of the set based analysis approach. We also outline the current status of this work and highlight some important areas of future work.

The underlying objective of this work was to develop an approach to program analysis in which the definitional and algorithmic aspects of the analysis are separated. In particular, we wanted a definition of program approximation that had a simple underlying intuition and could be stated in a declarative non-algorithmic manner. Moreover, we required a definition that yielded accurate information on program structure (an area where traditional methods have been weak) and could be computed with acceptable efficiency over the kinds of programs that are typically appear in practice.

Toward this end we considered an approach to analysis based on ignoring inter-variable dependencies. This was formalized by viewing program variables as sets of values and then extending this view to provide a *set based* approximation of a program's collecting semantics. The core chapter of the thesis shows that this approximation (denoted sba_P) is decidable. The use of constraints is featured heavily throughout, both in the definition of sba_P, and in its computation. In particular, we develop a calculus of set constraints that is used to represent sba_P and also forms the main data structure of the algorithm for sba_P. Importantly, one central calculus is used to represent the set based approximation of a number of different languages and operational semantics. The specific set operators required for different languages vary according to the language's semantic operations, but the essence of the constraints is the same in each case.

Not only is the set constraint calculus important for computing the specific approximation sba_P, but it appears to be useful in its own right. In particular, it provides a very flexible and declarative intermediate language for defining, reasoning about and computing program approximations in the set based style. Numerous variations are possible in the way set constraints are written to analyze a program. For example, the basic set constraints from a program can be modified to provide forms of analysis that distinguish between different instances of a function or predicate or calls from different call sites. On the other hand, the constraints can be simplified to trade off some of the accuracy of the constraints and against the cost of the analysis. Although these modifications mean that the resulting set constraints will not be strictly faithful to the approximation sba_P, they retain many of the important intuitions and accuracy advantages inherent in set based analysis. Set constraints can also adapted to compute information other than values of program variables. For example, in Chapter 9, we outlined adaptations for mode and structure sharing information.

logic program	result of set based analysis	result of abstract interpretation analysis
1. $\leftarrow p(U)$. 2. $p(f(X)) \leftrightarrow q(X)$. 3. $q(g(Y)) \leftrightarrow r(Y)$. 4. $r(h(a))$.	$\mathcal{U}^1 \supseteq f(\mathcal{X}^2)$ $\mathcal{X}^2 \supseteq g(\mathcal{Y}^3)$ $\mathcal{Y}^3 \supseteq h(a)$	$\mathcal{U}^1 \mapsto \{f(g(h(a)))\}$ $\mathcal{X}^2 \mapsto \{g(h(a))\}$ $\mathcal{Y}^3 \mapsto \{h(a)\}$

Figure 12.1: **Compact Representation of Constraints**

Although the use of the inter-variable dependency approximation leads to simple notions of program approximation and gives a uniform treatment of program structure, the loss of inter-variable dependencies is a substantial drawback for certain kinds of analysis. Motivated by the desire to capture some information about inter-variable dependencies and also maintain an accurate treatment of structure, we developed an extended notion of set constraint that incorporates inter-variable dependencies through an abstract interpretation style domain. This *unfolding* engine is strictly more accurate than either abstract interpretation or set constraint approaches to analysis. One of the tradeoffs for this gain in accuracy is that there is no longer a declarative connection between a program and its approximation.

A potential disadvantage of set based analysis is its computational cost. In general, the set based analysis algorithms have worst case EXPTIME behavior. However, in practice, it appears that programs rarely exhibit this. Significant progress has been made towards practical set based analysis by reformulating the set constraint algorithm and employing appropriate representation techniques. Although much work remains, practical set based analysis is within reach, particularly for functional programs.

We observe that the use of constraints in program analysis has one potential computational advantage over other methods. In particular, constraints provide a very compact representation of a program approximation because the set of values at one point can be defined recursively in terms of the values at some other program point. In contrast, abstract interpretation approaches construct a complete representation of each set at each program point. For example, Figure 12.1 shows a program, the constraints obtained from set based analysis of the program and the results from an abstract interpretation of the program (using a domain the represents values exactly up to a depth of 4). In essence, when constraints are used to analyze a program, the result is a single structure (which is a regular tree grammar), and the specific sets for each program variable and program point are identified by the different non-terminals of the grammar. In contrast, abstract interpretation approaches essentially construct a separate structure for each program variable and program point.

Although the algorithms of set based analysis are simple and direct, a number of the proofs are very complex. The main reason for this is that we have separated the definition of program approximation from the algorithms to compute it. Hence, the correctness proof of the sba_P algorithm (and this includes the correctness proofs for the translation of environment constraints to set constraints and the correctness proofs for the set constraint algorithm) must establish and an exact correspondence between sba_P and the output of the algorithm. In other words, our correctness proofs establish an "iff" conditions. If we were just interested in showing that our analysis computed a conservative approximation of the program's collecting semantics (and this is the format of the correctness proofs for most other analysis algorithms), then it would suffice to consider only one direction of these proofs. Moreover, the design of the set based approximation was primarily motivated by the desire for a simple definition of approximation. In a number of cases (notably in the approximation of imperative programs), this resulted in increased algorithmic complexity. In fact a substantial part of the complexity of the quantified expression algorithm is due to the presence of projections in program terms t in quantified conditions $t \in se$ and $t \dagger se$. The main difficulty here relates to the definedness of environments on such terms t, and this is the sole reason for the introduction of the safeness invariant. Another factor contributing to the complexity of the quantified expression algorithm is the presence of complement symbols, which are used to model the inequalities in imperative programs. In other words, a number of the complexities in the algorithm are a direct consequence of the complexities of reasoning about imperative programs. We note that if the only quantified expression constraints of interest are those that arise from logic programs, then the quantified expression algorithm can be very greatly simplified.

The main concern of this thesis has been the foundational issues of analysis that ignores inter-variable dependencies, and in particular, to show that for a variety of languages and operational semantics, such analysis is decidable. An additional concern has been to show that set based analysis is practical, and this is still work in progress. A number of extensions to the basic ideas have been sketched to demonstrate the flexibility of the set based approach. Two particularly promising extensions are the unfolding engine and the analysis of functional programs. The unfolding engine provides a general purpose engine for analyzing programs that combines accurate reason about structures with an ability to employ abstract interpretation domains that have been developed for reasoning about various aspects of inter-variable dependencies. We are currently investigating the implementation of the unfolding engine as well adaptations of the engine to the analysis of other languages such as constraint logic programming languages. The set based analysis of functional programs appears to have close connections to simple subtype systems, and this may potentially provide a very simple and intuitive characterization of this analysis. Moreover, the analysis of functional programs appears to be practical using current set constraint implementation techniques, and further improvements are expected.

Bibliography

[1] S. Abramsky and C. Hankin (Eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[2] W. Ackerman, *Solvable Cases of the Decision Problem*, North-Holland Pub. Co., Amsterdam, 1954.

[3] A. Aiken and B. Murphy, "Static Type Inference in a Dynamically Typed Language", *Proc. 18th ACM Symp. on Principles of Programming Languages*, Orlando, pp. 279–290, January 1991.

[4] A. Aiken and B. Murphy, "Implementing Regular Trees", *Proc. 5th ACM Conf. on Functional Programming and Computer Architecture*, Cambridge, LNCS 523, pp. 427–447, August 1991.

[5] A. Aiken and E. Wimmers, "Solving Systems of Set Constraints", *Proc. 7th IEEE Symp. on Logic in Computer Science*, Santa Cruz, pp. 329–340, June 1992.

[6] A. Aiken and E. Wimmers, private communication, June, 1992.

[7] K. Apt and M. van Emden, "Contributions to the Theory of Logic Programming", *Journal of the ACM*, Vol. 29, No. 3, pp. 841–862, 1982.

[8] L. Bachmair, H. Gandzinger and U. Waldmann, "Set Constraints are the Monadic Class", Technical Report MPI–I–92–240, Max-Planck-Institute for Computer Science, December 1992.

[9] B. Bogaert and S. Tison, "Equality and Disequality Constraints on Direct Subterms in Tree Automata", *Proc. 9nd Symp. on Theoretical*

Aspects of Computer Science, Cachan, France, LNCS 577, pp. 161–171, February 1992.

[10] M. Bruynooghe, "A Framework for the Abstract Interpretation of Logic Programs", Technical Report CW 62, Department of Computer Science, K. U. Leuven, October 1987.

[11] M. Bruynooghe and G. Janssens, "An Instance of Abstract Interpretation integrating Type and Mode Inference", *Proc.* 5^{th} *International Conf. and Symp. on Logic Programming*, Seattle, MIT Press, pp. 669–683, August 1988.

[12] A. Cortesi, G. Filé and W. Winsborough, "Equivalence of Abstract Domains for Groundness Analysis", Technical Report, Dip. di Matematica Pure é Applicata, Universitá di Padova, 1991.

[13] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", *Proc.* 4^{th} *ACM Symp. on Principles of Programming Languages*, Los Angeles, pp. 238–252, January 1977.

[14] P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks", *Proc.* 6^{th} *ACM Symp. on Principles of Programming Languages*, San Antonio, pp. 269–282, January 1979.

[15] S. Debray, "Global Optimization of Logic Programs", Ph.D. Thesis, Department of Computer Science, SUNY at Stony Brook, 233 pp., December 1986.

[16] G. Filé, "Tree Automata and Logic Programs", *Proc.* 2^{nd} *Symp. on Theoretical Aspects of Computer Science*, Saarbrucken, LNCS 182, pp. 119–130, January 1985.

[17] T. Freeman and F. Pfenning, "Refinement Types for ML", *Proc. 1991 ACM Symp. on Programming Language Design and Implementation*, Toronto, pp. 268–277, June 1991.

[18] T. Fruhwirth, E. Shapiro, M. Vardi, E. Yardeni, "Logic Programs as Types for Logic Programs", *Proc.* 6^{th} *IEEE Symp. on Logic in Computer Science*, Amsterdam, pp. 300–309, June 1991.

[19] F. Gècseg and M. Steinby, *Tree Automata*, Akadèmiai Kiadò, Budapest, 1984.

[20] N. Heintze, "Practical Aspects of Set Based Analysis", *Proc. Joint International Conf. and Symp. on Logic Programming*, Washington D.C., MIT Press, pp. 765–779, November 1992.

[21] N. Heintze and J. Jaffar, "A Finite Presentation Theorem for Approximating Logic Programs", *Proc.* 17^{th} *ACM Symp. on Principles of Programming Languages*, San Francisco, pp. 197–209, January 1990. (A full version of this paper appears as IBM Technical Report RC 16089 (# 71415), 66 pp., August 1990)

[22] N. Heintze and J. Jaffar, "A Decision Procedure for a Class of Herbrand Set Constraints", *Proc.* 5^{th} *IEEE Symp. on Logic in Computer Science*, Philadelphia, pp. 42–51, June 1990. (A full version of this paper appears as Carnegie Mellon University Technical Report CMU–CS–91–110, 42 pp., February 1991)

[23] N. Heintze and J. Jaffar, "A Decidable Analysis for Imperative Programs", draft manuscript, July 1990.

[24] N. Heintze and J. Jaffar, "Set Based Program Analysis", draft manuscript, January 1991.

[25] N. Heintze and J. Jaffar, "Semantic Types for Logic Programs" in *Types in Logic Programming*, F. Pfenning (Ed.), MIT Press Series in Logic Programming, pp. 141–155, 1992. (The work described in this paper was originally presented at the workshop on Types and Logic Programming held at the North American Conf. on Logic Programming, Cleveland, October 1989.)

[26] N. Heintze and J. Jaffar, "An Engine for Logic Program Analysis", *Proc.* 7^{th} *IEEE Symp. on Logic in Computer Science*, Santa Cruz, pp. 318–328, June 1992.

[27] D. Jacobs and A. Langen, "Accurate and Efficient Approximation of Variable Aliasing in Logic Programs", *Proc. North American Conf. on Logic Programming*, Cleveland, MIT Press, pp. 154–165, October 1989.

[28] J. Jaffar and J-L. Lassez, "Constraint Logic Programming", *Proc.* 14^{th} *ACM Symp. on Principles of Programming Languages*, Munich, pp. 111–119, January 1987. (An extended version appears as Technical

Report 86/73, Department of Computer Science, Monash University, June 1986.)

[29] J. Jensen, "Generation of Machine Code in Algol Compilers", *BIT* 5, pp. 235–245, 1965.

[30] T. Jensen and T. Mogensen, "A Backwards Analysis for Compile-Time Garbage Collection", *Proc.* 3^{rd} *European Symp. on Programming*, Copenhagen, LNCS 432, pp. 227–239, May 1990.

[31] N. Jones, "Flow Analysis of Lazy Higher-Order Functional Programs", in *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin (Eds.), Ellis Horwood, 1987.

[32] N. Jones and S. Muchnick, "Flow Analysis and Optimization of LISP-like Structures", *Proc.* 6^{th} *ACM Symp. on Principles of Programming Languages*, San Antonio, pp. 244–256, January 1979.

[33] N. Jones and S. Muchnick, "Complexity of Flow Analysis, Inductive Assertion Synthesis, and a Language due to Dijkstra", *Proc.* 21^{st} *IEEE-FOCS*, Syracuse, pp. 185–190, October 1980. (Also in, *Program Flow Analysis: Theory and Applications*, N. Jones and S. Muchnick (Eds.), Prentice-Hall, 1981.)

[34] N. Jones and S. Muchnick, "A Flexible Approach to Interprocedural Dataflow Analysis and Programs with Recursive Data Structures", *Proc.* 9^{th} *ACM Symp. on Principles of Programming Languages*, pp. 244–256, January 1982.

[35] N. Jones and A. Mycroft, "Data Flow Analysis of Applicative Programs Using Minimal Function Graphs" *Proc.* 13^{th} *ACM Symp. on Principles of Programming Languages*, St. Petersburg Beach, Florida, pp. 296–306 January 1986.

[36] N. Jones and H. Søndergaard, "A Semantics-based Framework for the Abstract Interpretation of PROLOG", in *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin (Eds.), Ellis Horwood, pp. 123–142, 1987.

[37] G. Kahn, "Natural Semantics", *Proc. Symp. on Theoretical Aspects of Computer Science*, Springer-Verlag LNCS 247, pp. 22–39, 1987.

[38] T. Kanamori and K. Horiuchi, "Type Inference in PROLOG and its Applications", *Proc. 9th International Joint Conf. on Artificial Intelligence*, Vol. 2, pp. 704–707, 1985.

[39] G. Kildall, "A Unified Approach to Global Program Optimization", *Proc. 1st ACM Symp. on Principles of Programming Languages*, Boston, pp. 194–206, January 1973.

[40] F. Kluzniak, "Type Synthesis for Ground PROLOG", *Proc. 4th International Conf. on Logic Programming*, Melbourne, MIT Press, pp. 788–816, July 1987.

[41] J. Lloyd, *Foundations of Logic Programming*, Second Edition, Springer-Verlag Series on Symbolic Computation, 1987.

[42] L. Löwenheim, "Über Möglichkeiten im Relativkalkül", *Math. Annalen*, Vol. 76, pp. 228-251.

[43] K. Marriott and H. Søndergaard, "Bottom-Up Abstract Interpretation of Logic Programs" *Proc. 5th International Conf. and Symp. on Logic Programming*, Seattle, MIT Press, pp. 733–748, August 1988.

[44] K. Marriott, H. Søndergaard and N. Jones, "Abstract Interpretation of Logic Programs: The Denotational Approach", *ACM Transactions on Programming Languages and Systems*, to appear. (Also, *Proc. 5th Italian Conf. Logic Programming*, pp. 399–425, 1990.)

[45] C. Mellish, "Some Global Optimizations fro a Prolog Compiler" *Journal of Logic Programming*, Vol. 2, No. 1, pp. 43–66, 1985.

[46] R. Milner, M. Tofte and Robert Harper, *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts, 1990.

[47] J. Mitchell, "Type Inference with Simple Subtypes", *Journal of Functional Programming*, pp. 245–285, July, 1991. (An early version of this paper appears as "Coercion and Type Inference (Summary)", *Proc. 11th ACM Symp. on Principles of Programming Languages*, Utah, pp. 175–185, January 1984.)

[48] P. Mishra, "Toward a Theory of Types in PROLOG", *Proc. 1st IEEE Symp. on Logic Programming*, Atlantic City, pp. 289–298, 1984.

[49] P. Mishra and U. Reddy, "Declaration-free Type Checking", *Proc. 12^{th} ACM Symp. on Principles of Programming Languages*, New Orleans, pp. 7–21, January 1985.

[50] T. Mogensen, "Separating Binding Times in Language Specifications", *Proc. Functional Programming and Computer Architecture*, London, ACM, pp. 12–25, September 1989.

[51] K. Muthukumar and M. Hermenegildo, "Determination of Variable Dependence Information Through Abstract Interpretation", *Proc. North American Conf. on Logic Programming*, Cleveland, MIT Press, pp. 166–186, October 1989.

[52] P. Naur, "The Design of the Gier Algol Compiler, Part II", *BIT* 3, pp. 145–166, 1963.

[53] B. Pierce, "Programming with Intersection Types and Bounded Polymorphism", Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Technical Report CMU–CS–91–205, December 1991.

[54] G. Plotkin, "Structural Approach to Operational Semantics", Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, 1981.

[55] C. Pyo and U. Reddy, "Inference of Polymorphic Types for Logic Programs" *Proc. North American Conf. on Logic Programming*, Cleveland, MIT Press, pp. 1115–1134, October 1989.

[56] P. Sestoft, "Replacing Function Parameters by Global Variables", *Proc. Functional Programming and Computer Architecture*, London, ACM, pp. 39–53, September 1989.

[57] O. Shivers, "Control-Flow Analysis of Higher-Order Languages", Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Technical Report CMU–CS–91–145, May 1991.

[58] M. Sintzoff, "Calculating Properties of Programs by Valuations on Specific Models", *Proc. ACM Conf. on Proving Assertions about Programs*, New Mexico, *SIGPLAN Notices*, Vol. 7, No. 1, pp. 203–207, 1972.

[59] H. Søndergaard, “An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction”, *Proc.* 1^{st} *European Symp. on Programming*, Saarbrucken, pp. 327–338, LNCS 213, March 1986.

[60] H. Søndergaard, “Semantics-Based Analysis and Transformation of Logic Programs”, Ph.D. thesis, University of Copenhagen, 1989. (Also available as Melbourne University Technical Report 89/21.)

[61] F. Pfenning (Ed.), *Types in Logic Programming*, MIT Press Series in Logic Programming, 1992.

[62] M.O. Rabin, “Decidability of Second-order Theories and Automata on Infinite Trees”, *Transactions of the American Math. Society* 141, pp 1 - 35, 1969.

[63] J. Reynolds, “Automatic Computation of Data Set Definitions”, *Information Processing 68*, North-Holland, pp. 456–461, 1969.

[64] S. Ronchi Della Rocca, “Principal type scheme and unification for intersection type discipline”, *Theoretical Computer Science*, Vol 59, pp. 181–209, 1988.

[65] T. Uribe, “Sorted Unfication and the Solution of Semi-Linear Membership Constraints”, M.S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Technical Report UIUCDCS-R-91-1720, December 1991.

[66] B. Wegbreit, “Property Extraction in Well-Founded Property Sets”, *IEEE Transactions on Software Engineering*, Vol. SE–1, No. 3, pp. 270–285, 1975.

[67] J. Xu and D.S. Warren, “A Type Inference System for PROLOG”, *Proc.* 5^{th} *International Conf. and Symp. on Logic Programming*, Seattle, MIT Press, pp. 604–619. August 1988.

[68] E. Yardeni and E. Shapiro, “A Type System for Logic Programs”, *Journal of Logic Programming*, Vol. 10, pp. 125–153, 1991. (An early version of this paper appears in *Concurrent PROLOG: Collected Papers*, Vol. 2, MIT Press, pp. 211–244, 1987.)

Appendix I: Existence of Least Models

Two main families of set calculi are used in the body of the thesis – environment constraints and set constraints. To avoid repeating results on basic properties of these calculi, this appendix proves these properties in a generic set calculi which subsumes both environment constraints and set constraints. The results themselves are not surprising and the proofs are rather straightforward. They are included for completeness rather than significance. The main property that is proved is that if all the operations of a set calculus are monotonic, then constraints of the form $var \supseteq exp$ have a model intersection property. An important corollary of this is the existence of least models.

A *generic set calculus* is a tuple of the form $(\mathcal{V}, \mathcal{OP}, \mathcal{D}, [\![\ ]\!])$ where $\mathcal{V}$ is a set of variables, $\mathcal{OP}$ is a set of operators, each with a unique arity, $\mathcal{D}$ is the set of underlying values of the calculus, and $[\![\ ]\!]$ defines the meaning of the operators by associating to each $op \in \mathcal{OP}$ of arity n, a function $[\![op]\!]$ which maps each n-tuple of subsets of $\mathcal{D}$ into a subset of $\mathcal{D}$. In the context of such a calculus, *expressions* are defined to be either variables or of the form $op(exp_1, \ldots, exp_n)$ where op is an operator whose arity is n, and the se_i are expressions. A *constraint* is of the form $exp \supseteq exp'$ where exp and exp' are expressions. An $\mathcal{I}$ *interpretation* is a mapping from variables into subsets of $\mathcal{D}$, and is extended to map from expressions into subsets of $\mathcal{D}$ in the obvious manner:

$$\mathcal{I}\left(op(exp_1, \ldots, exp_n)\right) = [\![op]\!]\left(\mathcal{I}(exp_1), \ldots, \mathcal{I}(exp_n)\right)$$

An interpretation is a *model* of a collection of constraint if, for each constraint

$exp \supseteq exp'$ contained in the collection, it is the case that $\mathcal{I}(exp) \supseteq \mathcal{I}(exp')$. Interpretations are ordered componentwise: $\mathcal{I} \supseteq \mathcal{I}'$ if $\mathcal{I}(var) \supseteq \mathcal{I}'(var)$ for each variable var in $\mathcal{V}$.

A *generic set calculus* is *monotonic* if each set operator $op \in \mathcal{OP}$ is such that $[\![op]\!]$ is monotonic in each argument. Given such a set calculus, an easy structural induction proves that expressions are monotonic in the following sense:

Proposition 46 *If $\mathcal{I} \supseteq \mathcal{I}'$ then $\mathcal{I}(exp) \supseteq \mathcal{I}'(exp)$.*

Proof: The proof is by a straightforward structural induction on exp. In the base case, where exp is a variable, the proof follows immediately from the fact that $\mathcal{I}(var) \supseteq \mathcal{I}'(var)$ for all variables var. For the induction case, suppose that exp is of the form $op(exp_1, \ldots, exp_n)$, and suppose that the proposition holds for each exp_i. This means that $\mathcal{I}(exp_i) \supseteq \mathcal{I}'(exp_i)$, $i = 1..n$, and combining this with the monotonicity of $[\![op]\!]$ proves that

$$[\![op]\!] \Big(\mathcal{I}(exp_1), \ldots, \mathcal{I}(exp_n)\Big) \supseteq [\![op]\!] \Big(\mathcal{I}'(exp_1), \ldots, \mathcal{I}'(exp_n)\Big).$$

It is immediate that $\mathcal{I}(op(exp_1, \ldots, exp_n)) \supseteq \mathcal{I}'(op(exp_1, \ldots, exp_n))$. $\Box$

Where $\mathcal{S}$ is a collection of interpretations, define that $\bigcap \mathcal{S}$ is the intersection of these interpretations defined by

$$(\bigcap \mathcal{S})(var) \stackrel{\text{def}}{=} \bigcap_{\mathcal{I} \in \mathcal{S}} \mathcal{I}(var), \text{ for each } var \in \mathcal{V}$$

Now, consider a collection of constraints $\mathcal{C}$ such that each constraint is of the form $var \supseteq exp$ where var is a variable from $\mathcal{V}$ and exp is an expression. Such constraints are said to be in *variable-expression* form, and they satisfy the following model intersection property:

Proposition 47 (Model Intersection Property) *Let $\mathcal{C}$ be a collection of variable-expression form constraints in a monotonic set calculus. If $\mathcal{S}$ is a collection of models of $\mathcal{C}$, then $\bigcap \mathcal{S}$ is also a model of $\mathcal{C}$.*

Proof: Let $var \supseteq exp$ be any constraint in $\mathcal{C}$, and consider the following chain of inequalities:

$$(\bigcap \mathcal{S})(var) \quad = \quad \bigcap_{\mathcal{I} \in \mathcal{S}} \mathcal{I}(var) \quad \supseteq \quad \bigcap_{\mathcal{I} \in \mathcal{S}} \mathcal{I}(exp) \quad \supseteq \quad (\bigcap \mathcal{S})(exp)$$

The first equality follows immediately from the definition of $\bigcap \mathcal{S}$. For the second inequality, recall that each $\mathcal{I} \in \mathcal{S}$ is in fact a model of $var \supseteq exp$, and so $\mathcal{I}(var) \supseteq \mathcal{I}(exp)$. The inequality then follows from a simple property of intersection. For the third inequality, note that $\mathcal{I} \supseteq \bigcap \mathcal{S}$ for each $\mathcal{I} \in \mathcal{S}$. The third inequality then follows form the fact that the intersection of a collection of sets, where each set is a superset of $(\bigcap \mathcal{S})(exp)$, must also be a superset of $(\bigcap \mathcal{S})(exp)$. Hence $\bigcap \mathcal{S}$ is a model of $var \supseteq exp$ and it follows that $\bigcap \mathcal{S}$ is a model of $\mathcal{C}$. $\square$

Corollary 4 (Least Models) *If $\mathcal{C}$ is a collection of variable-expression form constraints in a monotonic set calculus then $\mathcal{C}$ has a least model.*

Proof: Let $\mathcal{M}$ be the set of all models of $\mathcal{C}$. By proposition 47, $\bigcap \mathcal{S}$ must also be a model of $\mathcal{C}$. But $\bigcap \mathcal{S}$ is smaller than each $\mathcal{I} \in \mathcal{S}$, and so it must be the least model of $\mathcal{C}$. $\square$

We conclude this appendix with three properties relating to least models of variable-expression form constraints. First, define that a constraint of the form $var \supseteq exp$, where var is a variable and exp is an expression, is called a *lower bound* for the variable *var*.

Proposition 48 *Let $\mathcal{C}$ be a collection of variable-expression form constraints in a monotonic set calculus. If $v \in lm(\mathcal{C})(var)$ then $\mathcal{C}$ contains a lower bound for var of the form $var \supseteq exp$ such that $v \in lm(\mathcal{C})(exp)$.*

Proof: Let *var* be a variable and let v be a value. Suppose that all constraints of the form $var \supseteq exp$ in $\mathcal{C}$ are such that $v \notin lm(\mathcal{C})(exp)$. Then define an interpretation $\mathcal{I}$ by

$$\mathcal{I}(var') \quad \stackrel{\text{def}}{=} \quad \begin{cases} lm(\mathcal{C})(var) - \{v\} & \text{if } var' \text{ is } var \\ lm(\mathcal{C})(var') & \text{otherwise} \end{cases}$$

Consider a constraint of the form $var \supseteq exp$ in $\mathcal{C}$. Clearly $\mathcal{I} \supseteq lm(\mathcal{C})$, and so $\mathcal{I}(exp) \supseteq lm(\mathcal{C})(exp)$. Combining this with the definition of $\mathcal{I}$ and the fact that $lm(\mathcal{C})$ is a model of $var \supseteq exp$ proves that

$$\mathcal{I}(var) \; = \; lm(\mathcal{C})(var) - \{v\} \; \supseteq \; lm(\mathcal{C})(exp) - \{v\} \; = \; lm(\mathcal{C})(exp) \; \supseteq \; \mathcal{I}(exp)$$

and so $\mathcal{I}$ is a model of $var \supseteq exp$. Now consider a constraint of the form $var' \supseteq exp$ in $\mathcal{C}$ where var' is different from var. Since $\mathcal{I}(var') = \mathcal{I}(var') \supseteq \mathcal{I}(exp) \supseteq lm(\mathcal{C})(exp)$, it follows that $\mathcal{I}$ is also a model of this constraint. Hence $\mathcal{I}$ is a model of $\mathcal{C}$ that is less than $\mathcal{I}$, and this contradicts that assumption that $\mathcal{I} = lm(\mathcal{C})$. $\square$

An important corollary of this proposition is:

Proposition 49 *Let $\mathcal{C}$ be a collection of variable-expression form constraints in a monotonic set calculus. If $var \supseteq exp$ is the only lower bound for var then $lm(\mathcal{C})(var) = lm(\mathcal{C})(exp)$.*

The final proposition states a simple property of least models.

Proposition 50 *If $\mathcal{C}_1$ and $\mathcal{C}_2$ are collections of constraints such that $lm(\mathcal{C}_1)$ and $lm(\mathcal{C}_2)$ both exist, then*

$\mathcal{C}_1 \subseteq \mathcal{C}_2$ *implies* $lm(\mathcal{C}_1) \subseteq lm(\mathcal{C}_2)$.

Proof: Clearly any model of $\mathcal{C}_2$ is a model of $\mathcal{C}_1$. Hence $lm(\mathcal{C}_2)$ is a model of $\mathcal{C}_1$. Moreover, any model of $\mathcal{C}_1$ must be larger than $lm(\mathcal{C}_1)$, and so $lm(\mathcal{C}_1) \subseteq lm(\mathcal{C}_2)$. $\square$

Appendix II: Abstract Interpretation and Monadic Programs

To try to quantify the limitations of abstract interpretation, we shall outline why, unlike set constraint approaches, abstract interpretation cannot be exact over the class of monadic programs. Unfortunately, a completely formal and general account of this argument is very difficult to formulate, because it is always possible to give *ad hoc* extensions to an abstract interpretation algorithm, so that it can compute exact information on a specific monadic program. However, this cannot be done in a uniform manner to cover all monadic programs.

We therefore consider a somewhat restricted class of abstract interpretations based on the bottom-up semantics of logic programs. The underlying concrete domain shall be environments, and we shall consider arbitrary abstract domains for representing environments. After outlining the based arguments, we shall show how they can be extended to abstract interpretation involving the widening and narrowing operators.

We begin by considering the following program, were c is a constant and f is a monadic function symbol.

1. $p(c)$.
2. $p(f(X)) \leftarrow p(X)$.

Denote this program by P_1. We formalize the collecting semantics of P_1 by collecting sets of environments with each program rule. Specifically, define that an *association* d is a mapping from program rules into sets of environments, and let $\perp$ denote the association that maps each program rule into the empty set. Now, the exact semantic function for P_1 maps from and into such associations such that d is mapped into d' where

$$d(1) = \{\rho : \mathit{true}\}$$
$$d(2) = \left\{\rho : \begin{array}{l} \rho(p(X)) = \rho'(p(c)) \text{ for some } \rho' \in d(1), \quad \text{or} \\ \rho(p(X)) = \rho'(p(f(X))) \text{ for some } \rho' \in d(2) \end{array}\right\}$$

This definition can be more simply stated as:

$$d(1) = \{\text{all environments}\}$$

$$d(2) = \left\{\rho : \begin{array}{l} \rho(X) = c \text{ and } d'(1) \neq \{\} \quad \text{or} \\ \rho(X) = \rho'(f(X)) \text{ for some } \rho' \in d(2) \end{array}\right\}$$

Let $\mathcal{F}_1$ denote this semantic function. Now, consider the Kleene sequence $\perp, \mathcal{F}_1(\perp), \mathcal{F}_1(\mathcal{F}_1(\perp)), \ldots$ generated by this function, and we note that it does not converge finitely. The j^{th} element of this sequence, for $j \geq 2$, has the form

$$d_j(\alpha) \;=\; \left\{ \begin{array}{ll} \{\text{all environments}\} & \text{if } \alpha = 1 \\ \Theta_{j-2} & \text{if } \alpha = 2 \end{array}\right.$$

where Θ_j is the set of environments $\{[X \mapsto c], \ldots, [X \mapsto f^j(c)]\}$.

In abstract interpretation, sets of environments are approximated using an abstract domain and this induces an approximate semantic function. There many different formulations of abstract interpretation, differing in the specification of abstraction and concretization functions, and algebraic properties relating the concrete domain, the abstract domain and these functions. However, it is generally the case that the abstract domain can be treated as a collection of subsets of environments, and the abstraction function maps any collection Θ of environments into the smallest superset of Θ that is contained in the abstract domain. Since the abstraction function and the abstract domain are so closely related, we shall denote both by the same symbol, $\mathcal{A}$. We shall assume that the abstract domain contains the empty set and the set of all environments. Now, given a semantic function $\mathcal{F}$ and an abstract domain $\mathcal{A}$, the approximate semantic function can be defined by[1]:

[1] Strictly speaking, many abstract interpretations use a conservative approximation of this ap-

$\mathcal{F}^{\mathcal{A}}(d) = \mathcal{A}(\mathcal{F}(d))$.

Returning to the program P_1, the approximate semantic function corresponding to $\mathcal{F}_1$ maps an association d into

$$d(1) = \{\text{all environments}\}$$
$$d(2) = \mathcal{A}\left(\left\{\rho : \begin{array}{l} \rho(X) = c \text{ and } d(1) \neq \{\} \quad \text{or} \\ \rho(X) = \rho'(f(X)) \text{ for some } \rho' \in d(2) \end{array}\right\}\right)$$

Denote this semantic function by $\mathcal{F}_1^{\mathcal{A}}$. Now, if the abstract interpretation of P_1 is to terminate, then the sequence $\perp, \mathcal{F}_1^{\mathcal{A}}(\perp), \mathcal{F}_1^{\mathcal{A}}(\mathcal{F}_1^{\mathcal{A}}(\perp)), \ldots$ must converge finitely to some association d. Clearly it must be the case that $d(2) \supseteq \{[X \mapsto f^n(c)] : n \geq 0\}$. This means that, at some step i, the i^{th} step of the Kleene sequence for $\mathcal{F}^{\mathcal{A}}$ must differ from the i^{th} step of the Kleene sequence for $\mathcal{F}$. Let i be the step such that the Kleene sequences first differ. If d_i is the i^{th} step of the sequence for $\mathcal{F}^{\mathcal{A}}$, then it must be the case that d_i is a proper subset of Θ_{i-2}, because $\mathcal{A}$ is monotonic.

Now, using the value i, construct the logic program P_2 as follows:

$$\begin{array}{rl} 1. & p(c). \\ 2. & p(f(X)) \leftarrow q(X). \\ 3. & q(f(c)) \leftarrow p(c). \\ 4. & q(f(f(c))) \leftarrow q(f(c)). \\ \vdots & \qquad\quad \vdots \\ i+2. & q(f^i(c)) \leftarrow q(f^{i-1}(c)). \end{array}$$

The semantic function corresponding to P_2 can be stated as

---

proximate semantic function for algorithmic reasons. The formulation given here is an idealization of abstract interpretation, and represents an upper bound on its accuracy.

$$
\begin{array}{rcl}
d(1) & = & \{\text{all environments}\} \\
d(2) & = & \left\{ \rho : \begin{array}{c} \rho(X) = c \text{ and } d(3) \neq \{\} \text{ or} \\ \rho(X) = f(c) \text{ and } d(4) \neq \{\} \text{ or} \\ \cdots \quad \cdots \\ \rho(X) = f^i(c) \text{ and } d(i+1) \neq \{\} \end{array} \right\} \\
d(3) & = & \{\text{all environments}\} \\
d(4) & = & \{\rho : d(3) \neq \{\}\} \\
d(5) & = & \{\rho : d(4) \neq \{\}\} \\
\vdots & \vdots & \vdots \\
d(i+2) & = & \{\rho : d(i+1) \neq \{\}\}
\end{array}
$$

Let $\mathcal{F}_2$ denote this semantic function, and consider the Kleene sequence $\perp, \mathcal{F}_2(\perp), \mathcal{F}_2(\mathcal{F}_2(\perp)), \ldots$ that it generates. The j^{th} element of this sequence, for $j \geq 2$, has the form

$$
d_j(\alpha) = \left\{ \begin{array}{ll} \{\text{all environments}\} & \text{if } \alpha = 1 \\ \Theta_{min(i,j-2)} & \text{if } \alpha = 2 \\ \{\text{all environments}\} & \text{if } 3 \leq \alpha \leq j \\ \{\} & \text{if } j < \alpha \leq i+2 \end{array} \right\}
$$

Note that $\mathcal{F}_1$ and $\mathcal{F}_2$ are quite closely related, and that the first i elements of the respective Kleene sequences for these functions are virtually identical. It follows that the approximate semantic functions $\mathcal{F}_1^{\mathcal{A}}$ and $\mathcal{F}_2^{\mathcal{A}}$ are also closely related, and in fact it is easy to verify that the respective Kleene sequences for $\mathcal{F}_1^{\mathcal{A}}$ and $\mathcal{F}_2^{\mathcal{A}}$ are related in the following sense: if $j \leq i$ and $d_{1,j}$ and $d_{2,j}$ are respectively the j^{th} elements of the the Kleene sequences for $\mathcal{F}_1^{\mathcal{A}}$ and $\mathcal{F}_2^{\mathcal{A}}$, then

$$
\begin{array}{rcl}
d_{1,j}(1) & = & d_{2,j}(1) \\
d_{1,j}(2) & = & d_{2,j}(2)
\end{array}
$$

This means that the analysis of P_2 using the abstract domain $\mathcal{A}$ cannot be exact, since the least fixed point of $\mathcal{F}_2^{\mathcal{A}}$ must yield an association d such that $d(2)$ is a proper subset of Θ_i.

Finally, consider the operation of widening. Widening allows the use of abstract domains that would otherwise give non-terminating analysis. In effect, it allows that abstraction function to take, as an additional argument, the values obtained from the previous iteration. This is used to detect and truncate ascending chains of abstract values (see the discussion on page 15). The above argument showing that abstract interpretation cannot be exact on P_2 can be easily applied to abstract interpretation that employs the widening operator. The main observation is that the construction of the j^{th} step of the Kleene sequences for P_1 and P_2 using widening and the abstract domain $\mathcal{A}$ are respectively is given by

$$\begin{aligned} d_{1,j}(\alpha) &= \Big(d_{1,j-1}(\alpha)\Big) \nabla \Big(\mathcal{F}_1^{\mathcal{A}}(d_{1,j-1})\Big), \quad \alpha = 1..2 \\ d_{2,j}(\alpha) &= \Big(d_{2,j-1}(\alpha)\Big) \nabla \Big(\mathcal{F}_2^{\mathcal{A}}(d_{2,j-1})\Big) \quad 1 \leq \alpha \leq i+2 \end{aligned}$$

where ∇ denotes the widening operator employed. It is again easy to show that $d_{1,j}(1) = d_{2,j}(1)$, where, as before, i is the smallest number such that the i^{th} step of the Kleene sequence for $\mathcal{F}$ differs from $d_{1,j}$. Hence, the analysis of P_2 using widening and the abstract domain $\mathcal{A}$ results in an association d that does not correspond to the exact collecting semantics of P_2. We remark that narrowing can be subsequently applied to improve the accuracy of d, but in general it cannot recover the exact collecting semantics of a monadic program.

Table of Notation

Notation	**Explanation**
VAR	the set of program variables
X, Y, Z	a program variable
Σ	the set of function symbols
f, g, h	a data constructor or function symbol
b, c	a constant (0-ary function symbol)
$f_{(i)}^{-1}$	the i^{th} projection of f
s, t	a term (logic program or imperative program)
cond	an imperative program condition
Stat	an imperative program statement
Seq	a sequence of imperative program statements
A, B, C	a logic program atom
R	a logic program rule
P	a program (imperative or logic)
α, β, γ	a program label
μ, λ	a program point
v	a program value
S	a set (usually of program values)
ρ	an environment (a mapping from VAR into values)
Θ	a set of environments
$\rho[X \mapsto v]$	the environment ρ except that X is mapped into v
θ	a substitution (a mapping from VAR into terms)
E	a conjunction of term equations
$\rho \models cond$	ρ satisfies *cond*
$\rho \models E$	ρ satisfies each equation in E
Ψ	an environment variable
Ψ^{μ}	the environment variable for program point μ

Notation	**Explanation** (contd.)
ϱ	a set environment
$\varrho[X \mapsto S]$	the set environment ϱ except that X is mapped into S
$\rho \models_\varrho cond$	ρ satisfies *cond* in the context of ϱ
$var(exp)$	the program variables in *exp* if *exp* is a sequence of program terms, rules or equation conjunctions
$var(exp)$	the set variables in *exp* if *exp* is a collection of set expressions or set constraints
$var(\alpha)$	the program variables in the rule R if α is the label of R
$var(\alpha)$	the program variables in the rule R if α is the label of a body atom in R
$\mathcal{X}, \mathcal{Y}, \mathcal{Z}$	a set variable
$\mathcal{V}_N$	an intersection variable
se	a set expression
a	an atomic set expression
$conj$	a conjunction of quantified conditions
$\{X : conj\}$	a quantified set expression
$\mathcal{C}$	a collection of set constraints
$\mathcal{I}$	a mapping from variables into sets
$\mathcal{I} \models \mathcal{C}$	$\mathcal{I}$ is a model of $\mathcal{C}$
$lm(\mathrm{C})$	least model of the constraints $\mathcal{C}$
$\mathcal{EC}_P$	the environment constraints for P
$\mathcal{CS}_P$	the collecting semantics of P
sba_P	the least set based model of $\mathcal{EC}_P$
$\mathcal{SC}_P$	the set constraints for P
δ	a transformation

Notes:

1 Function symbols and projection symbols are overloaded in the sense that these symbols are not only used to denote mappings from values into values, but they are also used in set expressions as mappings from sets into sets. The intended usage is usually clear from context.

2 A value is a data structure in the case of an imperative program and a ground term in the case of a logic program.

3 For logic programs, program labels and program points coincide.

Index