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

1 Thesis Summary **1**

2 Introduction **7**

2.1 Program Analysis 7

2.2 Program Analysis by Abstract Interpretation 9

2.3 Limitations of Abstract Interpretation 13

2.4 Approximation of Values and Variables 20

2.5 Set Based Analysis 22

2.6 Overview of Thesis 30

PART I: Set-Based Approximation **33**

3 Imperative Programs **35**

3.1 Imperative Programs 36

3.2 Operational Semantics 37

3.3 Collecting Semantics 40

3.4 Environment Constraints 44

3.5 Environment Constraint Correctness 47

4 Logic Programs **57**

4.1 Logic Programs 58

4.2 Operational Semantics 59

4.3 Comparison of Operational Semantics 64

4.4 Collecting Semantics 66

4.5 Environment Constraints 72

4.6 Environment Constraint Correctness 78

4.6.1 Correctness of Top-Down Constraints 79

4.6.2 Bottom-up Semantics 96

5 Set Based Approximation **103**

5.1 Summary of Environment Constraints 104

5.2 Inter-Variable Dependencies in $\mathcal{EC}_P$ 106

5.3 Set Based Interpretation of $\mathcal{EC}_P$ 116

5.4 Alternative Definitions 122

5.5 Examples 126

5.6 Related Work 131

6 Set Constraints **143**

6.1 Set Constraints 144

6.2 Environment Constraints and Set Constraints 150

PART II: Set Based Analysis **163**

7 Solving Set Constraints **165**

7.1 Explicit Representation of $lm(\mathcal{C})$ 166

7.2 Overview of Algorithm 172

7.3 Converting Constraints to Standard Form 174

7.4 The Generic Algorithm 176

7.5 Intersection and Projection 179

7.6 Quantified Set Expressions 196

7.7 Related Work 251

8 Implementation **257**

8.1 Introduction 258

8.2 The Basic Set Constraint Algorithm 261

8.3 Outline of Implementation 263

8.4 Intersectors 270

8.5 Empirics 273

PART III: Extensions and Further Work **279**

9 Modes and Structure Sharing **281**

9.1 Mode Analysis 282

9.2 Sharing Analysis 286

10 The Unfolding Engine **291**

10.1 Motivation 292

10.2 Collecting Semantics 293
10.3 Unfolding Semantic Equations 297
10.4 The Engine 302
10.5 Variations of the Engine 312
10.6 Comparison with Abstract Interpretation 313
10.7 Discussion 317

11 Analysis of Functional Languages **319**

11.1 Higher-Order Set Constraints 320
11.2 Examples 322
11.3 Extensions and Variations 325
11.4 Implementation 328
11.5 Discussion 330

12 Conclusions **333**

Bibliography **339**

Appendix I: Existence of Least Models **347**

Appendix II: Abstract Interpretation and Monadic Programs **351**

Table of Notation **357**

Index **359**

List of Figures

2.1 **Program 1 and Its Collecting Semantics** 8

2.2 **Program 2 and Its Collecting Semantics** 10

2.3 **Abstract Interpretation of Program 2** 11

2.4 **Revised Abstract Interpretation of Program 2** 12

2.5 **Program 3 and Its Abstract Interpretation** 12

2.6 **Revised Abstract Interpretation of Program 3** 15

2.7 **Program 4 and Its Collecting Semantics** 17

2.8 **Abstract Interpretation of Program 4** 17

2.9 **Program 5** 19

2.10 **Program 2 (Revisited)** 23

2.11 **Set Constraints for Program 2** 24

2.12 **Program 6 (With and Without Program Points)** 26

2.13 **Set Constraints for Program 6** 28

3.1 **Program 2 (Revisited)** 37

3.2 **Collecting Semantics for Program 2** 43

3.3 **Program 2 and Its Environment Constraints** 46

4.1 **Program 7** 61

4.2 **Program 8** 63

4.3 **Program 9 and Its Collecting Semantics** 70

4.4 **Program 9 and Its Top-Down Left-To-Right Constraints** . 74

4.5 **Program 10 and Its Top-Down Left-To-Right Constraints** . 74

4.6 **Program 9 and Its Top-Down Interleaved Constraints** . . . 75

4.7 **Program 9 and Its Bottom-Up Constraints** 75

4.8 **Program 10 and Its Bottom-Up Constraints** 76

5.1 **Different Ways of Introducing Dependencies** 108

5.2 **Undecidability of Modified Set Based Interpretation** 123

5.3 **Inter-Argument Dependency Example** 124

5.4 **Set Based Approximation of** $X := pair(X, X)$ 126

5.5 **Set Based Approximation of Program 2** 126

5.6 **Bottom-Up Set Based Approx. of Two Logic Programs** . . 127

5.7 **Top-Down Set Based Approximation of Program 10** 128

5.8 **Program 11 and Its Set Formula** 132

5.9 **Differences between** T_{P}, $\mathcal{T}_{\mathrm{P}}$ **and** Y_{P} 135

5.10 **Differences between** T_{P}, $\mathcal{T}_{\mathrm{P}}$, Z_{P} **and** Y_{P} 136

7.1 **The Generic Algorithm** 175

7.2 **Example Constraints and Their Least Model** 178

7.3 **Constraints from Figure 7.2 Rewritten in Standard Form** . 178

7.4 **Example Algorithm Execution** 182

7.5 **The Rewrite Steps of** REDUCE() 197

8.1 **Bottom-Up Set Constraints** 257

8.2 **Top-Down Set Constraints** 258

8.3 **The Append Program and Its Top-Down Constraints** . . . 258

8.4 **Example Execution of the Reformulated Algorithm** 261

8.5 **Example Constraints and Their Representation** 265

8.6 **Dependency-Based Algorithm** 266

8.7 **The lcm and hcf Benchmarks** 273

9.1 **Mode Analysis Example** 280

9.2 **The Append Program and Its Top-Down Constraints** . . . 282

9.3 **Three Programs Illustrating Accuracy of Mode Analysis** . 284

9.4 **Sharing of Structures Between Variables** 284

9.5 **Set Constraints for Programs in Figure 9.4** 286

10.1 **Append Program** 292

10.2 **Example Clusters Containing More Than One Group** . . . 294

10.3 **Top-Down Semantics Constraints for Append** 296

10.4 **The Unfolding Engine** 304

10.5 **Example to Illustrate Execution of Unfolding Engine** . . . 304

10.6 **Example Illustrating the Order of the Unfolding Steps** . . 307

10.7 **The Standard Engine (Bottom-Up)** 311

10.8 **Example Execution of the Standard Engine** 312

10.9 **The Standard Engine (Top-Down)** 313

11.1 **Example Functional Program and Set Constraints** 319

11.2 **Example Approximation of a Functional Program** 320

11.3 **Map Program and Its Set Constraints** 321

11.4 **DNF Program and Selected Constraints** 324

11.5 **Analysis of References** 325

12.1 **Compact Representation of Constraints** 333

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

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* $[L \mapsto list, X \mapsto const]$ 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 $[L \mapsto non\text{-}empty, X \mapsto const]$. Propagating this information to point Ⓑ yields $[L \mapsto non\text{-}empty, X \mapsto const]$, and this in turn gives $[L \mapsto list, X \mapsto any]$ at point Ⓒ. Further propagation steps lead to the results summarized in Figure 2.3. Note that when $[L \mapsto non\text{-}empty, X \mapsto any]$ is added to point Ⓑ, $[L \mapsto list, X \mapsto const]$ is deleted because it is subsumed by $[L \mapsto non\text{-}empty, X \mapsto any]$.

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 termi-

nates. 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

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

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

	Point	Environments
$L := b.nil;$		
Ⓐ	Ⓐ	$\{[L \mapsto b.nil]\}$
while $len(L) \neq 3$ **do**		
Ⓑ	Ⓑ	$\left\{ \begin{array}{c} [L \mapsto b.nil] \\ [L \mapsto a.b.nil] \end{array} \right\}$
$L := a.L;$		
Ⓒ	Ⓒ	$\{[L \mapsto a.b.nil]\}$

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

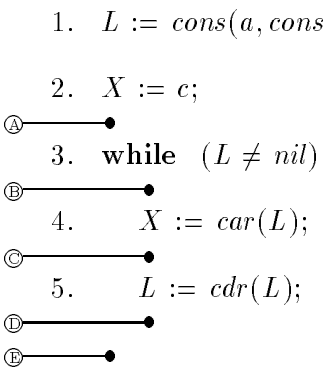

Figure 2.10: **Program 2 (Revisited)**

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

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

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

constraints for the program.

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

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

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 $\mathcal{L}^{\text{Ⓑ}} \supseteq \mathcal{L}^{\text{Ⓐ}} \cap \overline{nil}$ 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 $\mathcal{L}^{\text{Ⓑ}} \supseteq \mathcal{L}^{\text{Ⓓ}} \cap \overline{nil}$ 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

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

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

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

could be used in the place of the two constraints $\mathcal{L}^{\text{Ⓑ}} \supseteq \mathcal{L}^{\text{Ⓐ}} \cap \overline{nil}$ and $\mathcal{L}^{\text{Ⓑ}} \supseteq \mathcal{L}^{\text{Ⓓ}} \cap \overline{nil}$ 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. Sup-

pose 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

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

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

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

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

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

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

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:

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

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

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

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

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).