A FRAMEWORK FOR SPECIFICATION AND IMPLEMENTATION OF PROGRAM ANALYSIS ALGORITHMS

by

G A Venkatesh

Computer Sciences Technical Report #875

August 1989

A FRAMEWORK FOR SPECIFICATION AND IMPLEMENTATION OF PROGRAM ANALYSIS ALGORITHMS

by

G. A. Venkatesh

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN—MADISON
1989

A FRAMEWORK FOR SPECIFICATION AND IMPLEMENTATION OF PROGRAM ANALYSIS ALGORITHMS

G. A. Venkatesh

Under the supervision of Professor Charles N. Fischer

Program analysis algorithms take a program as input and output information about static and dynamic properties of the program. Flow analysis algorithms form an important class of analysis algorithms and have been used extensively in implementation of programming languages. Recent developments in programming environments have resulted in language-based editors that carry out many tasks that were once traditionally considered as compiler functions. In addition to various flow analysis algorithms, other kinds of program analysis algorithms such as type inference or complexity analysis can be used to provide powerful interactive programming environments. The growing sophistication of these analysis algorithms necessitates a structured approach to their design to ease their development as well as to ensure their correctness.

Currently, analysis algorithms are either developed in operational frameworks that make formal verification difficult or specified in theoretical frameworks that do not translate easily into implementations. This thesis bridges the gap between theory and practice by developing a framework to specify program analysis algorithms. The feasibility of the framework is demonstrated by the construction of a tool based on the framework.

This thesis

- Demonstrates the feasibility of developing a wide variety of analysis algorithms through high-level specifications.
- Identifies certain characteristics of program analysis algorithms and documents their impact on the design of a denotational specification language.
- Proposes a specification language with features that allow analysis algorithms to be expressed in a clear and concise fashion.
- Provides a formal semantics for the specification language.
- Develops guidelines for deriving correctness proofs for analysis algorithms.
- Provides a tool that can be used for rapid prototyping of analysis algorithms.

Acknowledgments

Although the letters in this thesis are attributed to a single name on the title page, every empty space is filled with the silent contributions of several. To these people, I wish to express my gratitude.

My first thanks go to my advisor, Charles Fischer, who guided me impeccably through the intricacies of academic research. As a mentor and a friend, he managed to instill in me an appreciation for ideas and events, both in and out of the realm of Computer Science.

I thank Marvin Solomon and Thomas Reps for their insightful comments on both the form and content of this thesis. Their comments have resulted in great improvements over earlier versions of the thesis.

My fellow researchers, from past and present, have contributed towards many of the ideas in this thesis. In particular, I thank Felix Wu and Will Winsborough for their willingness to listen to my ideas and their solutions to several nagging problems.

I thank the Computer Sciences department and the National Science Foundation for providing financial support. I also wish to thank the administrative and the computing resources staff for their continued assistance during my entire stay in school.

Many special friends were responsible for making my stay in Madison, a truly memorable experience. In particular, I would like to thank Norm Carlisle for being an equally insane tennis fan and a regular sparring partner, Dana van Kooy for having been part of several of my sanity checks, and Viranjit Madan for all the help, advice and friendship provided during my stay in Madison.

And very special thanks go to my parents whose love and support were always assumed but never sufficiently acknowledged. To them, I dedicate this thesis.

Table of Contents

Chapter 1
Introduction

Program analysis algorithms take a program as input and output information about static and/or dynamic properties of the program. An important area of application for such information is program translation. A class of program analysis algorithms usually known as *data flow analysis algorithms* [1, 17] have been traditionally used in compilers to aid in various stages of program translation.[1] Some examples of flow analysis algorithms and their use in compilers are mentioned below:

(1) *Reaching definitions*: The definition site of a variable is a statement in a program (input or assignment statement) that assigns a value to the variable. A definition *reaches* a use of the variable if there is a path from the definition site to the use without any intervening definition sites. For example, consider:

```
(1) A := 1;
(2) B := A;
(3) while B > 0 do
(4)       B := B - 1;
(5)       A := A + B;
(6) od;
```

The definition in line (2) reaches the use in line (4) but not in line (5). The definitions in both lines (1) and (5) reach the use in (5). The information from such an analysis can be used for a variety of purposes including efficient register allocation. For example, all definitions of a variable that reach a particular use may be made to assign their values to the same register to minimize register copying.

(2) *Available expressions*: An expression is *available* at a program point if the expression has already been computed and recomputation would be redundant. For example, consider:

```
(1) A := B + C;
(2) C := A + B + C;
```

The expression `B + C` is available in line (2). Such an analysis can be used to perform optimizations involving elimination of common subexpressions.

[1]In fact, the compiler itself can be considered as an implementation of a specific program analysis algorithm. This notion will become more obvious later in the thesis.

More detailed descriptions of flow analysis techniques and their uses can be found in literature such as [1,17,31].

Recent developments in programming environments have blurred the distinction between language editors and compilers/interpreters. Modern language based editors carry out many tasks that were once traditionally considered as compiler functions. In addition to various flow analysis algorithms, other kinds of program analysis algorithms such as type inference or complexity analysis can be used to provide powerful programming environments. Information obtained from the analysis algorithm can be used to provide feedback to the programmer to aid in construction, understanding and debugging of programs. Type checking is an example of a program analysis commonly used to provide feedback to the programmer.

Another example of an application of program analysis is the use of *dependence analysis* to construct *program slices. Dependence analysis* provides information about data dependences between program points. This information can be used to construct program slices [65] which are abstractions of a program that capture a subset of the program's behavior. Facilities to view program slices rather than the entire program can be of immense help in understanding or debugging of programs. For illustration, consider the following program to compute the factors of a number.

```
(1)  read(n);
(2)  i := 2; c := 0;
(3)  while i < (n div 2) do
(4)       if (n mod i) = 0 then
(5)            write(i);
(6)            c:=c+1;
(7)       fi;
(8)       i := i+1;
(9)  od;
(10) write(c);
```

The above program has a bug and does not compute all the factors for certain numbers. Rather than look at the entire program one may wish to see only the parts of the program that affect the computation of the factors. A program slice with respect to the statement in line (5) is given below:

```
read(n);
i := 2;
while i < (n div 2) do
    if (n mod i) = 0 then
        write(i);
    fi;
    i := i+1;
od;
```

Note that the portion of the program that counts the number of factors is not present in the slice. The bug in the program is also present in the program slice. Program slices can thus be used to reduce the complexity of the program for debugging purposes or just for comprehension of programs. More recent studies have used program slicing to integrate multiple versions of programs [21].

Other useful applications for analysis algorithms in programming environments include

- Program browsing.
- Detection of unreachable code.
- Automatic variable declaration.
- Automatic type and type implementation selection.
- Automatic documentation.

Given the large number of potential applications and the growing sophistication of program analysis algorithms, it is desirable to establish a structured approach to the design of analysis algorithms to ease their development as well as to insure their correctness.

1.1. Correctness of Analysis Algorithms

To gain confidence about an analysis, it is necessary to establish certain desirable properties of the analysis. Three essential properties are:

(1) Consistency with the semantics of the language:

Program analysis algorithms are designed to make assertions about properties (static and/or dynamic) of the program. Since such properties are determined by the semantics of the language, it is important to ensure that the analysis algorithms are consistent with the language semantics. The correctness issue becomes crucial as the analyses become more sophisticated.

(2) Termination:

Static analysis algorithms must terminate for any program. It is useful to provide formal reasoning that guarantees termination.

(3) Safety:

A compile-time analysis usually provides an approximation to the actual execution of the program either due to inherent limits such as unknown input values or due to implementation considerations that trade-off precision for efficiency and/or termination. The analysis is designed to err on the conservative side. The actual runtime property of the program (that is being approximated) must imply the information gathered from the analysis. Note that safety implies consistency with the semantics.

1.2. Approaches to Program Analysis Design

A solution to a program analysis problem consists of three components:

(1) The program representation on which the analysis is to be carried out:

Algorithms use various representations for programs. *Control flow graph* representations are commonly used in the specification of flow analysis algorithms. Other representations include *abstract syntax trees*, *static single assignment graphs* [3,54,64], and variants of the *program dependence graph* [16,22,33].

(2) The domain of values that describe the required property of programs:

The value domains are commonly modeled as *lattices*.[2]

(3) A mapping between programs and the value domain:

The mapping is usually specified through equations or functions.

There are three requirements for any approach to the design of program analysis algorithms:

(1) Methods for natural expression of solutions to program analysis problems.

(2) Implementation of program analysis algorithms for testing and rapid prototyping.

(3) Formal verification to ensure the correctness of program analysis algorithms.

A successful approach to the design of analysis algorithms must include suitable facilities to satisfy all the three issues. There are two distinct approaches currently used for specification of solutions to program analysis problems: the *operational* approach and the *semantics-driven* approach.

[2] Actually *semilattices* or just *complete partial orders* are sufficient for most analysis problems.

1.2.1. Operational approach

The operational approach is the most commonly used approach for specification of program analysis algorithms. Representations of the programs to be analyzed and the structure (lattice, semilattice, cpo, etc.) of the value domain are described informally. The mapping between the program and the value domain is defined operationally in an algorithmic language. A typical example is the specification of conditional constant propagation algorithms by Wegman and Zadeck in [64].

This approach is less than ideal for several reasons. First, the informal language used for the specifications makes it difficult to systematically derive formal proofs of correctness, in particular, the proof of consistency with the formal semantics of the language in which the programs to be analyzed are written. Second, an algorithm specification must be manually implemented in some programming language for testing. In addition to the programming effort, the consistency of the algorithm specification with the implementation must be established. This problem can be alleviated by using a programming language to specify the algorithm. However, the use of a programming language results in the third objection to the operational approach — the presence of implementation details in the specification.

Although the operational approach is convenient for expressing the mapping between programs and value domains, the lack of support for high-level specification of structures (lattices, abstract syntax trees or control flow graphs) on which these mappings are defined results in the presence of implementation details in the algorithm specification. This has several consequences:

- The design process is complicated by the need to handle low-level implementation details. The design of analysis algorithms is heavily dependent on the application. Generally, analysis algorithms provide approximate information about the dynamic behavior of programs. There is always a trade-off between the effort involved in the analysis and the precision with which information is obtained from the analysis. The application sets the requirements for precision and efficiency. Low-level development of an analysis algorithm makes it difficult to prevent duplication of effort in developing techniques that are closely related.
- The presence of implementation details makes it harder to understand the algorithms in order to maintain or improve existing ones.
- Correctness proofs are difficult to derive for algorithms with too much implementation detail.

These problems highlight the need for a formally defined high-level specification language for designing analysis algorithms.

1.2.2. Semantics-driven approach

Readers familiar with the formal specification of language semantics will have noticed the similarity between the methods of formal semantics and the components of a program analysis specification listed at the beginning of Section 1.2. Methods for specification of formal semantics such as Denotational Semantics [55,57] or Natural Semantics [29] map programs to values in semantic domains. This similarity was first explored in the seminal paper on *abstract interpretation* [11].

Abstract interpretation introduced the notion that a variety of program analysis algorithms could be formally expressed as interpretations over abstract domains. An abstract domain has a structure which is the minimum required to encode the properties that are to be derived from an analysis. Many flow analysis algorithms such as ***sign propagation, constant propagation*** or ***range propagation*** can be defined as a sequence of successive abstractions with the limit being the standard interpretation (corresponding to the normal execution) of the program. The abstraction relationship between the analysis and the standard semantics could be formalized, enabling formal proofs of consistency between the two.

Although the original work involved operational specifications for flow-chart languages, most of the later studies based on abstract interpretation [14,23,25,27,41] have used denotational specifications. A denotational specification of a program analysis can be considered as an alternative semantics for the language. Most of the techniques developed for providing standard semantics of a language can be used to provide an alternative semantics. The structural and compositional nature of denotational specifications ease the development of correctness proofs through the use of well understood methods such as structural induction or fixed-point induction. Consistency with standard semantics can be established easily with respect to the denotational semantics of the language. Studies [4,44,46,56] on implementation techniques for denotational specifications demonstrate the possibility of providing efficient evaluators for denotational specifications.

Unfortunately, denotational frameworks for specification of program analysis algorithms have remained theoretical. With a few notable exceptions [14,41], most studies have been in the domain of functional programs and for specification of a rather limited number of analysis algorithms such as strictness analysis. Despite the considerable amount of theoretical foundation set up by Nielson [41-43] for expression of solutions to data flow problems, denotational frameworks have remained impractical for program analysis design.

Two of the problems associated with denotational frameworks are mentioned below. They will be discussed in more detail in the next chapter.

(1) Many simple data flow analysis algorithms, such as dependence analyses, use domains that are not straightforward abstractions of domains used in standard interpretation.

(2) The requirement of most analysis algorithms to associate information with program points cannot be expressed naturally in denotational frameworks, leading to messy specifications that are difficult to understand (the specifications provided in [41] illustrate this point). Moreover, this thesis will show that the use of partial evaluations of denotational functions to capture information at intermediate points raises some theoretical concerns regarding the correctness of such information.

1.3. A hybrid approach

This thesis makes a case for a hybrid approach that combines the positive aspects of both the operational and the semantics-driven approach. An extended denotational framework is used to provide specifications in a modular fashion. This extended framework eases the design of analysis algorithms as well as the derivation of correctness proofs. Facilities for natural expression of solutions to program analysis problems are explored. Guidelines for establishing formal verification proofs are provided.

This approach is useful for design of program analysis algorithms that operate at the source level on programs written in structured languages. The thesis does not explore the facilities required for analysis on programs with unstructured constructs (gotos or exceptions).

The feasibility of this approach is demonstrated through a tool called SPARE (Structured Program Analysis Refinement Environment) that enables rapid prototyping of analysis algorithms. The high-level nature of the specification language results in clear and concise specifications in which implementation details can be ignored.

Chapter 2 describes the hybrid approach. The characteristics of analysis algorithms and their theoretical and practical implications are discussed. Chapters 3, 4 and 5 describe the SPARE system. The specification language is briefly described in Chapter 3. Example specifications are provided for some analysis algorithms in Chapter 4. Chapter 5 provides some details about the implementation. Chapter 6 discusses the theoretical aspects of correctness proofs for specifications written in SPARE. Chapter 7 describes related work, suggests issues for further research and summarizes the thesis.

Chapter 2
A Structured Approach to Analysis Specifications

Structured specifications are a common ground between an operational approach and a semantics-driven approach. Denotational frameworks provide for structured specifications and have several advantages as mentioned in the previous chapter. In this chapter, we will discuss the pragmatic and theoretical issues involved in the expression of program analyses in denotational frameworks.

Most of the flow analyses designed using the operational approach use a low-level representation of programs. However, the advantages of imposing some structure on these low-level representations have been realized for a long time. The initial studies based on a structured approach [2,7] used the structure of *nested strongly connected regions* (e.g. nested loops). The studies based on *graph grammars* [15] are examples of earlier approaches to structured specification. A large number of control flow graphs that arise in practice were shown to be reducible using a graph grammar. The flow analysis is specified through equations for each production in the graph grammar. The flow analysis for the entire graph is then obtained by structural composition.

With the evolution of structured Algol-like languages, the syntactic structure of a program could be exploited for the specification of program analyses. This evolution led to the notion of *high-level data flow analysis* [53] where the parse tree representation of the program is used. The flow analyses are specified through equations for each production in the grammar for the language. The system of equations for the entire program can be constructed by structural composition. The equations specified for each construct in the language encode the control flow information implicit in the parse tree.

For the class of flow analysis problems considered in these studies, the flow equations could be specified using simple set operations (union and intersection). One can conceive of enriching the specification language in order to accommodate other kinds of analysis and this is where we find a meeting point with the semantics-driven approach.

The meaning of programs is specified in denotational semantics through *semantic equations* which associate with each production in the grammar a function representing the meaning of a construct derived from the production. The meaning of the entire program is then derived through structural induction. An extended version of the lambda calculus notation is usually used for denoting the functions. As this notation is powerful enough to express set operations, the high-level data flow analysis specifications can be expressed in denotational frameworks. The studies mentioned in the previous chapter demonstrate that the richness of denotational notation allows the expression of many other analyses as well. However, expressional power does not necessarily imply ease of expression.[3]

Before we consider the pragmatic and theoretical deficiencies of denotational frameworks for expression of program analyses, we need to look closely at the way control flow information is encoded using functional forms.

2.1. Functional forms and control flow

In high-level data flow analysis, the flow equations derived for each production take into account the control flow semantics of the corresponding language construct. In denotational frameworks, functional forms implicitly capture the control flow semantics. Language constructs with a single entry/single exit property can be easily represented by functional forms built using a small set of functional constructors such as composition, conditional, or fixed-point constructors. Fig. 2.1 illustrates a mapping between some basic control structures and functional forms.

(1) We will assume that a basic unit is mapped to a function expressed in lambda calculus notation.

(2) A sequence of constructs is mapped to a functional composition of corresponding functional forms.

(3) The conditional branch construct is mapped to the conditional functional constructor. A merge node D has been introduced into the construct to ensure a single exit. The function f_D is the identity function in standard semantics specification. However, in program analysis specifications one may wish to merge information flowing into D and specify f_D accordingly.

(4) The loop construct is mapped to a recursive equation that can be denoted by a functional form using the fixed-point constructor. The same comments for f_D in the conditional construct apply to f_C here. I is the identity function.

It should be noted that the mapping between control structures and functional forms is not unique. The particular functional form used for a construct usually depends on the analysis being defined. For example, in analyses where all paths are assumed to be possible in execution, the conditional constructor is not required for denoting conditional branch constructs. In contrast, Nielson [42,43] assumes a unique mapping. An analysis is then specified through the functions for basic blocks and the selection of one of several alternative interpretations available for the metalanguage. Although this approach results in an elegant theoretical framework, it is rather impractical for development of real-life program analysis algorithms, especially those (e.g. dependence analysis) that are not simple abstractions of the standard interpretation.

[3]Those familiar with Turing Machines will readily attest to this fact.

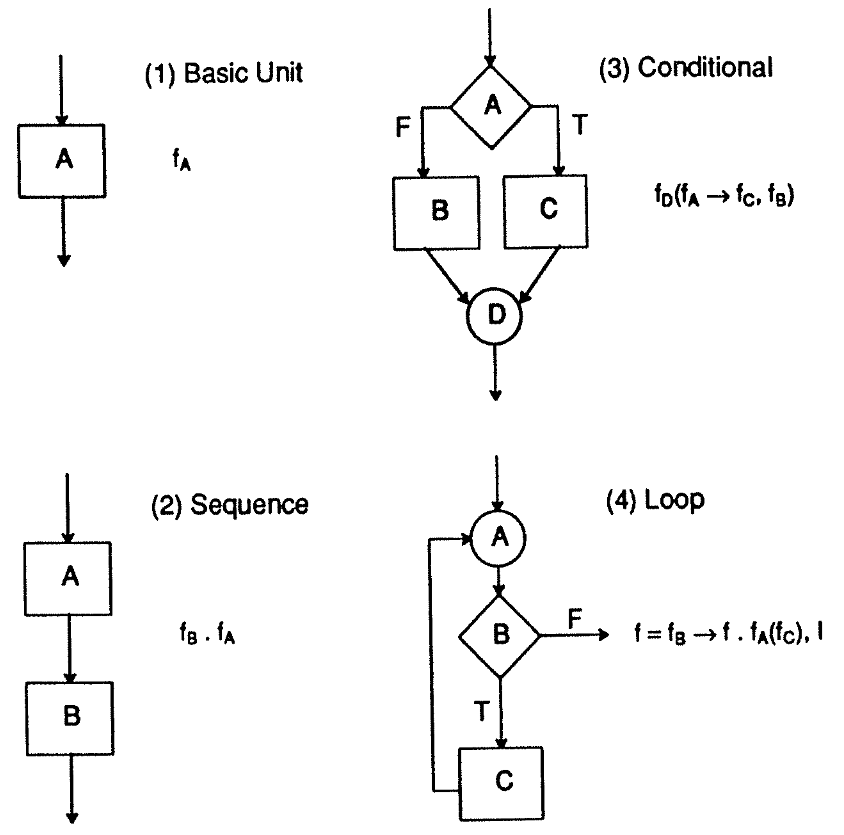

Figure 1. Encoding control flow in functional forms

What, one may ask, is a "valid" (or useful) mapping? A mapping is "valid" (and useful) if answers to questions about certain properties of the language constructs can be obtained, in a formally verifiable way, from answers to questions about the corresponding functional forms. In specification of standard semantics, the only question that needs to be answered concerns the relationship between the program states before and after the execution of the construct. The answer is derived from the input/output relationship for the corresponding functional form. The formal verification consists of proving that the following diagram commutes for all constructs.

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\text{Input program state} \arrow[r, "\text{Semantic mapping}"] \arrow[d, "\text{Execution}"'] & \text{Input to the function} \arrow[d, "\text{Function evaluation}"] \\
\text{Output program state} \arrow[r, "\text{Semantic mapping}"'] & \text{Output of the function}
\end{tikzcd}$$

Figure 2.2. Validity of standard semantics mapping

However, when we consider program analyses, many different kinds of questions need to be answered. Typical questions posed [53] in data flow analyses include

- Can execution of statement α modify the value of variable X?
- Can execution of statement α preserve the value of variable X?
- Can execution of statement α use the value of variable X?

Questions that differentiate program analysis from standard interpretation are of the form

- What is X at the program point Y
  (where X is some predicate that provides the desired information)?

In the flow graph model, answering this question requires associating desired information with arcs (or nodes) in the graph[4]. How is the answer derived in denotational frameworks? For example, consider the mapping of the sequence construct to functional composition in Fig. 2.1. What denotes the program state when the control reaches the program point between the two constructs A and B? By making natural assumptions about the evaluation of functional compositions, one may indicate the input to the function f_B (or alternatively, the output of the function f_A) during evaluation as the denotation of the intermediate state.

In the next section, we will describe desirable extensions to denotational frameworks that ease the specification of program analyses from a pragmatic point of view, given some natural assumptions about

[4]Originally called *static semantics* in [11]. Later studies have used the term *collecting interpretation*.

evaluation of functional forms. We will consider the theoretical implications of such assumptions in Section 2.3.

2.2. Pragmatic issues in denotational frameworks

In order to make tools based on denotational frameworks practical, the framework should allow for natural expression of analysis algorithms resulting in clear and concise specifications. It should be possible to derive implementations automatically from the specification to enable testing and rapid prototyping. This requirement places certain demands on the metalanguage used for the specifications. The metalanguage must provide facilities to abstract out routine operations and require minimal implementation details. Given these concerns, the denotational frameworks used in previous studies have two major inadequacies.

2.2.1. Domain specifications

Although analyses may use domains that are abstractions (in the sense of information content) of domains used in standard interpretations, the structures imposed on abstract domains are not necessarily simpler than the structure of standard domains. For example, the lattice used for sign propagation is an abstraction of the integer domain used in standard interpretation. The integer domain is usually a flat domain. On the other hand, the lattices pictured in Fig. 2.3 have been suggested [11,43] for sign propagation.

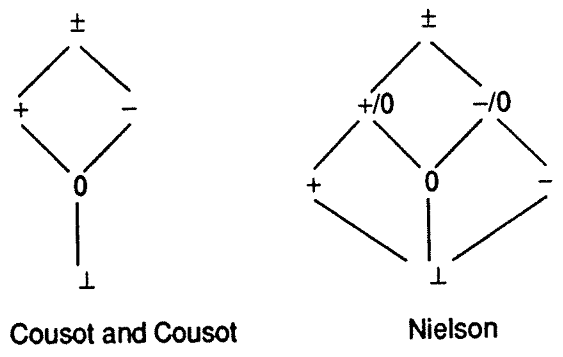

Figure 2.3. Lattices for sign propagation

The domain constructors generally used in denotational semantics do not directly allow specification of such structures. It is essential that suitable domain constructors be available in the metalanguage to allow automatic derivation of implementations for analyses that require such structures.

2.2.2. Collection of partial evaluations

We mentioned earlier that the inputs and/or outputs to semantic functions denote information to be derived from the analysis under certain assumptions about the evaluation model of functional forms. The most obvious way to collect results of intermediate evaluations[5] is through the introduction of an extra component into the argument and result domains of all functions. For example, let

$$f_{\mathrm{Label}} : A \rightarrow B$$

be a semantic function corresponding to an instance of a syntactic object uniquely identified as Label[6]. Then a "collecting" function $\bar{f}_{\mathrm{Label}}$ can be defined as

$$\bar{f}_{\mathrm{Label}} : A \times C \rightarrow B \times C = \lambda a.\ \lambda c.\ (f(a),\ \mathit{update}(c, \mathrm{Label}, g(a)))$$

An element of domain C, often called a *cache* [24], is used to implement the "side-effect" of associating results of intermediate evaluations with syntactic structures. g is a function that derives the information to be collected. If g is the identity function, then the input to the semantic function is collected. If $g = f$, the output of the semantic function is collected. In general, g can be any function. The function *update* is used to update the cache with the value to be collected from the current evaluation. Functions such as $\bar{f}$ are said to define the *collecting semantics* of the language. Analysis algorithms are expressed as approximations to *collecting semantics*.

Although this technique allows the expression of analyses that require intermediate information, it has a serious impact on the ease and clarity of specifications.

First, collecting specifications tend to get complex. Each function definition contains operations that constitute the analysis as well as the mechanism to collect intermediate information [14,41]. The collecting mechanism often obscures the analysis portion of the definition. This problem is somewhat alleviated by the framework developed by Hudak and Young in [24][7] in which the collecting specification is separated from the analysis specification. However, even this framework does not address the second disadvantage — the need to provide an imperative definition for the collecting mechanism.

[5] Assuming a purely functional model with no global variables and side-effects.

[6] Corresponds to *occurrences* in [14,41] and *places* in [40]

[7] Although, their motivation was different.

Collection mechanisms for various analysis specifications tend be very similar. Usually either the input or the output of a semantic function is associated with the syntactic object for which the semantic function is defined.[8]

Our solution to these problems consists of extending denotational frameworks to allow declarative rather than procedural definition of the collecting mechanism. The implementation required for collection is derived automatically from the declarative specification. The features of the specification language that provide this facility is described in the next chapter.

2.3. Theoretical issues in denotational frameworks

In the previous section, we discussed the need to collect results of partial evaluations. In this section, we will demonstrate the need to provide an evaluation model for the metalanguage in which the specifications are expressed, so that we can reason about the results of partial evaluations. Given such a requirement, we will discuss restrictions on specifications under which the derivation of correctness proofs can be simplified.

The denotational specification of a language specifies precisely what the value of any language construct is, but not how it is computed. The meaning of a construct is simply the function denoted by the lambda expression used in the definition. Denotational definitions are related to language implementations by providing simplification rules for the denotations (or in other words, a semantics for the metalanguage). For example, the semantics of λ-calculus is used for the metalanguage of λ-notation.

One can view denotational equations as a translation of a program to its denotation and program execution as an evaluation (i.e., a simplification) of the denotation according to the semantics of the metalanguage. If the simplifications resemble the computational steps that occur in a conventional implementation, then partially simplified denotations that occur during the simplification can be used to represent states that occur during execution. Although the Church-Rosser property makes the simplification order irrelevant for the final form in λ-calculus, the denotations that occur after partial simplification depend on the order in which the simplifications are performed. Thus any assertion about possible intermediate states must be made relative to a semantics of the metalanguage.

A practical use of an analysis specification is in the automatic derivation of an implementation for that analysis. Programs can then be analyzed to predict some aspect of their execution behavior. For the predictions to be meaningful, the following must be established:

[8]corresponds to the association of information on incoming or outgoing arcs with the node in flow graph models.

(1) The consistency of the analysis implementation with the specification.

(2) The consistency of the analysis specification with the standard semantics of the language. This verification is immensely easier if the standard semantics can be expressed in the same metalanguage as the analysis. Denotational frameworks have the advantage that a denotational semantics is available for many languages.

(3) The consistency of the language implementation with the operational definition of the standard semantics expressed in the metalanguage. The studies dealing with the *full abstraction problem* [39,47] are motivated by this goal. This thesis will not explore this problem.

If implementations for analyses are obtained through an automatic translation of the metalanguage, then the correctness of the translation process guarantees (1) for all analysis specifications. The major goal of the theoretical portion of this thesis is to facilitate the derivation of proofs to ensure (2).

The classical framework developed by Cousot and Cousot [11] uses a pair of adjoined functions α and γ called abstraction and concretization functions respectively. To show consistency with standard semantics, a collecting semantics is induced from the standard semantics. The set of states that occur whenever control reaches a program point is associated with that program point. Correctness is proved by defining adjoined functions relating the semantic domain of the collection semantics (called the collecting domain) to the semantic domain of the analysis specification (called the approximation domain) to satisfy the following schema:

$$\begin{tikzcd}
\text{Collecting Domain} & \text{Approximation Domain} \\
\text{Input} \arrow[r, "\alpha"] \arrow[d, "\text{Collecting Interpretation}"'] & \text{Input} \arrow[d, "\text{Abstract Interpretation}"] \\
\text{Output} \leq & \text{Output} \arrow[l, "\gamma"]
\end{tikzcd}$$

Figure 2.4. Correctness of an abstract interpretation

For more details on this approach the reader is referred to [11].

The derivation of abstraction and concretization functions is fairly simple for analyses which use approximation domains that are simple abstractions of the collecting domain. For example, if the collecting domain D_c, is a store that maps identifiers to sets of integer values and the approximation domain D_a is a store that maps identifiers to values in the lattice C in Figure 2.5, the abstraction function $\alpha : D_c \rightarrow D_a$ is defined pointwise using the function $\bar{\alpha} : P(I) \rightarrow C$ where

$$\bar{\alpha}(x) = \begin{cases} \perp & \text{if } x = \{\} \\ n & \text{if } x = \{n\} \\ \top & \text{otherwise} \end{cases}$$

What if an analysis uses an approximation domain that is not a simple abstraction of the collecting domain? For example, algorithms for reaching definitions use a store that maps identifiers to statement labels. The program state denotations used in standard semantics usually have no information about statement labels. One solution is to extend the standard domain to include information that is approximated in the analysis.[9] For example, the standard semantics for an assignment statement can be extended to map the identifier to the value being assigned to it as well as the statement label of the assignment statement. The collecting semantics is then induced over this extended "standard" semantics and the abstraction function is established as before to relate the collecting domain and the approximation domain. Unfortunately, there is significant effort involved in writing these various semantics formulations and establishing relationships among them.

However, if analysis algorithms are designed as interpretations of the control flow graph over abstract domains (in the spirit of abstract interpretation), we can devise an alternate proof scheme in which only the standard semantics is required. Moreover, given certain restrictions on the use of semantic

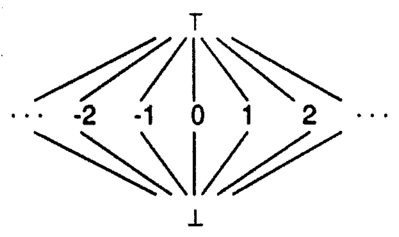

Figure 2.5. Lattice for constant propagation

[9] The semantics obtained by such an extension has been called *instrumented semantics* in [20].

functions in the specifications, portions of the verification can be automated. This approach is explained in more detail in Chapter 6. We will provide an outline of the scheme here.

Standard semantics defines the behavior of single executions of a program. Collecting semantics defines the behavior over all possible executions.[10] Analysis specifications generally approximate the collecting semantics because of efficiency considerations (including termination). The proof scheme described earlier ensures that the information derived from an analysis safely approximates the information that can be provided by the collecting semantics. Because the collecting semantics considers all possible execution paths and executions paths are determined by the standard semantics, an analysis specification provides safe approximations that are consistent with the standard semantics of the language.

Given that the main purpose behind the use of a collecting semantics is to ensure that all possible paths are considered, is there an alternative approach that guarantees that all paths are considered but avoids the use of a collecting semantics? The answer is yes, provided the analysis algorithms are expressed as abstract interpretations. The alternate approach is based on relating the analysis specification to a denotation of "all possible paths" rather than the denotation of "information over all possible paths." The most obvious denotation for "all possible paths" is the control flow graph.

In the beginning of this chapter, we mentioned the mapping from flow graphs to functional forms encoding control flow information. The operational semantics for the metalanguage uniquely defines control flow dependences between functions that represent program structures. The control flow graph for any program (assuming only structured constructs) can be obtained using these dependences and the standard semantics specification for the language in which the programs are written. Any execution path possible in the control flow graph can be denoted by an *execution sequence* which is a sequence of semantic function instances that are evaluated when execution occurs along that path.

The consistency and safety properties of an analysis can be established easily if it can be shown that for all programs:

(1) Dependences in the control flow graph induced by the standard semantics are preserved in the control flow graph induced by the analysis specification.[11] This verification can be automated if the semantic functions are not used in a Curried fashion or used as first class objects.

[10]Although this information is not necessarily computable.

[11]This is trivially satisfied by the approach in [43] where the same denotation is used for both standard and analysis specifications.

(2) Corresponding to every evaluation of a standard semantic function instance F in any execution sequence, there is an evaluation of an instance of the corresponding (abstract) semantic function $\bar{F}$ in the analysis specification satisfying an invariant $P(x, \bar{x})$ where x and $\bar{x}$ are inputs to the semantic function instances F and $\bar{F}$ respectively.

The predicate P depends on the analysis. If the approximation domain is an abstraction of the standard domain (in constant propagation, range propagation, etc.), P is of the form

$$\bar{x} \geq \alpha(x)$$

where α is an abstraction function.

For algorithms that include the traditional flow analysis algorithms, P is usually of the form

$$(\bar{x} = x = \perp) \vee C(\bar{x}).$$

For example, in reaching definitions specifications in which $\bar{x}$ is a store that maps identifiers to a set of statement labels

$$C(\bar{x}) \equiv \forall id \in dom(\bar{x}).\ \bar{x}[id] = \{L \mid \text{definition of id at L is live}\}$$

Proofs using this approach will be provided in Chapter 6 for some example specifications in Chapter 4.

2.4. Summary

Although denotational frameworks can be used to express a wide variety of analysis algorithms, pragmatic and theoretical concerns place several demands on the design of the metalanguage and its semantics. The guidelines established in this chapter can be used to design practical tools/systems based on denotational frameworks, that enable development of analysis algorithms. The next chapter describes a tool that has been built to demonstrate the feasibility of a denotational approach to program analysis development.

Chapter 3
SPARE

The Structured Program Analysis Refinement Environment (SPARE) is a tool for rapid prototyping of program analysis algorithms through structured high-level specifications. It has been designed to overcome the inadequacies of denotational frameworks mentioned in the previous chapter. The specification language is based on the notation of lambda-calculus and the conventions used for writing denotational semantics of programming languages. Language features have been specially designed to express analysis algorithms in a clear and concise fashion.

SPARE is designed to be used in conjunction with the Synthesizer Generator [52]. Analysis specifications are translated into specifications in the Synthesizer Specification Language (SSL). The SSL translation, combined with the SSL specification for an editor for the language on which the analysis is defined, can be used to generate an editor for the language. The generated editor performs the specified analysis on programs input to the editor and the results can be displayed to verify the analysis. The reference manual for SPARE is in [60].

3.1. SPARE language

The SPARE specification language is an extended version of the notation used in denotational semantics. The language description is included in the reference manual [60]. In this chapter, we will describe briefly salient features of the language and the motivations behind it. We will assume that the reader is familiar with the method of denotation semantics. Reference material on denotational semantics includes [55,57,58].

A specification consists of *domain declarations* and functions defined over these domains. There are two kinds of functions. *Semantic functions* define the analysis corresponding to various syntactic objects in the language. *Auxiliary functions* provide a way to abstract operations used in the specifications.

3.1.1. Domain constructors

Domains required for the analysis are constructed using the primitive domains — integer, boolean, syntactic, label — and several domain constructors. Constructors are available for the domains traditionally used in domain theory such as product domains, disjoint union domains and powerset domains. Domains can be lifted[12] through the explicit use of the **lifted** constructor. A dual constructor **topped** is available to introduce the top element $\top$.

An enumeration domain constructor is provided to specify finite collections of distinct objects. An enumeration domain is specified as a list of string elements. For example, the domains containing the proper elements[13] of the lattices in Fig. 2.3 can be declared as

```
Cousot_Lattice = { "0", "+", "-", "+-" }
Nielson_Lattice = { "0", "+", "-", "0+", "0-", "+-" }
```

The **function** constructor is used to build function domains. Stores are traditionally modeled in denotational semantics using function domains. Stores usually map a *finite* number of elements to proper (non-$\perp$) elements. A **store** constructor is provided to explicitly denote such finite maps. Store domains can be used in lookup and update operations whose syntactic representations are distinct from operations on functions (e.g. application). This facility not only enhances readability but also allows for efficient implementation of finite maps. However, as we shall describe below, a major motivation for a separate store domain constructor comes from the ordering imposed on the various domains.

Structuring domains

Partial orders within domain elements are either pre-defined, induced by domain constructors or explicitly specified by the user. The primitive domains are all pre-defined as flat domains[14]. The domain constructors induce partial orders traditionally defined in domain theory. For example, the elements of a product domain are ordered component-wise. Let A = A1 * A2 * $\cdots$ * An. The induced ordering $\leq_A$ is defined as follows:

$$(x_1, x_2, \cdots, x_n) \leq_A (y_1, y_2, \cdots, y_n) \text{ iff } x_i \leq_{Ai} y_i,\ 1 \leq i \leq n$$

In domain theory, function domains are ordered point-wise. For example, let $D = A \rightarrow B$. The ordering in D is defined as

$$\forall d_1, d_2 \in D.\ d_1 \leq_D d_2 \text{ iff } \forall a \in A.\ d_1(a) \leq_B d_2(a)$$

Although this ordering is mathematically well defined, in general it is not computable. Since we are only interested in the specification of effective semantics, the function constructor induces a flat ordering in the

[12]Lifting a domain D creates the domain $D_\perp$, where $D_\perp = D \cup \{\perp\}$ and $\perp \leq d, \forall d \in D$.

[13]Non-$\perp$ elements of a domain are called *proper* elements. The $\perp$ is called an *improper* element.

[14]A domain D is called a *flat domain* iff $\forall d_1, d_2 \in D.\ d_1 \leq d_2$ iff $d_1 = d_2$ or $d_1 = \perp$

SPARE language[15]. As mentioned earlier, function domains are commonly used in denotational semantics to model stores. Store lattices (or cpos) are essential for most specifications. Fortunately, stores being finite maps, the pointwise ordering in store domains is computable. Hence, an explicit *store* constructor is provided to induce pointwise ordering.

Enumeration domains are ordered flat unless an explicit ordering is provided. An explicit ordering is specified as a list of tuples <a , b> where a and b are string literals specified in the enumeration domain declaration. The tuple establishes the order $a \leq b$. It is sufficient to provide all the "immediately less than" relations in the required ordering. The ordering through transitivity is implicitly established. There are no checks made to ensure that the specified ordering is a pre-ordering. The lattice structures in Fig. 2.3 can be specified as follows

```
Cousot_Lattice = { "0", "+", "-", "+-" }
 ordered by (<"0","-">,<"0","+">,<"-","+-">,<"+","+-">)

Nielson_Lattice = { "0", "+", "-", "0+", "0-", "+-" }
 ordered by (<"0","0+">,<"0","0-">,<"+","0+">,<"+","+-">,
       <"-","0-">,<"-","+-">,<"0+","+-">,<"0-","+-">)
```

The bottom element is introduced through explicit lifting of the above domains. Note that the element "+-" is the $\top$ in each of these lattices. Alternatively, it could have been left out of the enumeration list and introduced through the explicit use of the **topped** operator.

Finally, an ordering can also be specified externally. This is an escape clause to specify complex orderings outside the specification language or to provide a library of standard orderings (e.g. power-domain orderings). The ordering is defined by the four functions - <*join, equal, bottom, top*>[16]. We assume that an analysis algorithm that uses the meet operation can be expressed dually using the join operation.

Pragmatic issues

Certain decisions made in the language design are based on pragmatics rather than theory or elegance.

[15]Moreover, the equality operator is undefined for elements of function domains.

[16]In domain theory, the ordering is usually represented by the equivalent tuple $<\sqcup, \leq, \bot, \top>$. In practice, the equality function is more useful than the partial order relation.

(1) Recursive domains can make type verification difficult as well as result in inefficient implementations. Our decision was to allow recursive (direct or indirect) specifications in function domains only. The other kind of recursive domain that is most often used in denotational semantics - list domains - can be constructed using the explicit list constructor.

(2) A strongly typed language allows many bugs to be detected through static checks. SPARE requires specifications to be sufficiently typed to allow complete static type checking of operations. Each domain declaration creates a type. Type equivalence is defined through name equivalence for all non-functional domains. One-level structural equivalence is used for function domains (i.e., two function types are equivalent if and only if each of the corresponding argument and result types of the two function types are name equivalent). To make type checking easier, only a single domain constructor can be used in any domain declaration. This restrictions avoids the creation of anonymous types without affecting the expressive power.

3.1.2. Function declarations

The functions in the SPARE language are of two kinds - *auxiliary functions* that abstract operations in the specifications and *semantic functions* that specify the analysis for the various syntactic objects in the language. The function type declaration is similar to the function domain declaration except for the facility to specify *strictness*. Each argument domain name in the type declaration can have the prefix **strict**. The *strict* option forces the function to be strict[17] in the arguments for which the option is used. For example, consider the declaration for an auxiliary function[18]

```
Select : strict A --> B --> C is λx. λy. y
```

The function defined by $\lambda x.\ \lambda y.\ y$ is not strict in its first argument. However, the use of the strict option forces the function `Select` to be equivalent to the function

$$\lambda x.\ \lambda y.\ (x = \perp) \rightarrow \perp, y$$

Although the strict option appears in the type declaration, it is not considered as part of the type description (functionality) of the function. It specifies an implicit program transformation on the function definition.

Semantic functions are declared using the same notation as used in denotational semantics. The first argument domain of a semantic function is always a syntactic domain. We will use a "Curried" notation for

[17]A function is said to be *strict* relative to one of its arguments if the function returns $\perp$ whenever that argument is $\perp$.

[18]λ is replaced by the keyword **lambda** in the actual syntax due to the current limitations of I/O devices.

semantic function type declaration. For example,

```
    E [[exp]] = Store --> Sign_Lattice
```

defines a semantic function E associated with the syntactic domain exp. Throughout this thesis we will use the notion of semantic functions being *associated* with syntactic objects rather than the traditional notion of semantic functions being *applied* on a semantic object. The semantic functions can only be used in this "Curried" form[19]. This decision allows each semantic equation to be treated as an independent function definition rather than as a clause in an implicit case expression[20].

The declaration of semantic functions is similar to the type declaration of auxiliary functions except for two optional suffixes - the collect option and the external option. The external option specifies that the semantic equations for the function will be provided in a separate specification. This feature can be used to make specifications share common semantic functions or to specify analyses that use information provided by a separate analysis. The collect option is a declarative method to specify collection of intermediate evaluations.

Collect option

In Chapter 2, we discussed the need to collect results from intermediate evaluations. Of all the possible partial evaluations, evaluations of semantic functions are of the most interest. Evaluation of semantic functions models the execution of syntactic constructs. The input to and output from these semantic functions represent information flowing in and out of the constructs during execution.

Operationally, execution of a program (or an analysis) can be viewed as a traversal of the abstract syntax tree in which a visit to a node results in the execution of the denotation corresponding to that node[21]. One may view collection as storing some information at a node during every visit to it. The information is derived either from the information flowing into or out of the node.

In practice, the collection of the required information is done uniformly across all instances of a semantic object (e.g. the inputs to all statements or the outputs from all statements). This uniformity is exploited in specifying the collect option for a semantic function as part of the type declaration.

[19]The function may be curried further, of course.

[20]In [37], the semantic equations are specified explicitly as clauses in a case statement.

[21]In fact, studies such as [44] use this model in actual implementations.

The collect option specifies two things:

- The argument or the result from which the collect information must be obtained.
- The procedure to extract the required information from the argument or result.

For example, in the declaration

```
P [[ prog ]] : D1 --> S --> D2 collect D1
```

the value of the first argument will be collected whenever an instance of the semantic function is evaluated. In general, one may imagine applying an arbitrary function on the argument value and collecting the result. In practice, we do not find sufficient motivation to allow an arbitrary function to be specified. Hence, there are two options available to specify a collection. If S in the above declaration is a product domain and the information of interest is a component of a value in that domain, one can use

```
P [[ prog ]] : D1 --> S --> D2 collect S^3
```

This specification specifies the value of the third component of the second argument to the function as the information to be collected.

If the same instance of the semantic function P is evaluated more than once, the information from successive evaluations is "joined" together using the *least upper bound* operation. In the above example, the least upper bound in the component domain would be performed. However, one may wish to obtain a collection of values from all intermediate evaluations at a program point rather than the least upper bound of these values. This can be achieved by using the powerset option. For example,

```
P [[ prog ]] : D1 --> S --> D2 collect powerset of D2
```

In this case the value to be collected is a singleton set containing the result of the function. Note that the values from successive evaluations will be "joined" in the powerset domain where the least upper bound is given by set union. Hence, the collected information will be a set of all outputs from successive evaluations. This method is used in the collecting framework of [24] to effectively separate the collecting domain from the standard domain. Both the powerset and the selection options can be used together with the selection taking precedence.

If a domain name appears more than once in the type declaration, occurrences must be differentiated by using suffixes of the form "$n" where n is an integer. For example, in the declaration

```
P [[ prog ]] : D$1 --> S --> D$2 collect D$1;
```

the value of the first argument rather than the result is collected. The suffix does not create new domains. Both `D$1` and `D$2` are the same as the domain `D` (as opposed to `D1` and `D2` which are distinct domain names).

The pre-defined operation

cache : syntactic → cache

can be used to access information associated with syntactic objects. cache is a store domain

```
cache = store label --> collect-domain
```

where `collect-domain` is determined from collect option declaration. All objects in syntactic domains are implicitly associated with a label which can be accessed using the *label* operator.

3.1.3. Semantic equations

Semantic functions are defined through a list of *semantic equations*. A semantic equation defines the function for a single production in the abstract syntax. The function is denoted by an expression that is an extended version of the lambda expressions used in denotational semantics. For example,

```
E [[(Add exp$2 exp$3 )]] =
     lambda x. ( E [[exp$2]] + E [[exp$3]] ) x
```

Occurrences of the same syntactic domain are differentiated by using suffixes of the form "$n". Only the syntactic objects listed in the production may be used in the expression.

3.1.4. Expressions

Several pre-defined operations are available for use in expressions. Besides the usual operations on integers, booleans, sets, lists, stores, etc., there are three overloaded operations - *join, unary join* and *equal*.

JOIN

The operation join : $A \times A \rightarrow A$ for any domain A is denoted by +. If a_1 and a_2 denote values in the same domain A, then $a_1 + a_2$ denotes the least upper bound of the values denoted by a_1 and a_2. If the least upper bound for the two values does not exist, then the join is undefined.

UNARY JOIN

The unary join operator, also denoted by +, is used to summarize information in elements of power-set and store domains. The unary join on a set provides the least upper bound of all the elements in the set. The unary join on a store provides the least upper bound of all the elements in the codomain that have a preimage in the domain.

EQUAL

Two values $a_1, a_2 \in A$ are equal iff $a_1 \leq_A a_2$ and $a_2 \leq_A a_1$.

Function application, composition, conditional and let expressions have the traditional syntax and semantics. The following expressions deserve comment:

EXPRESSION UNION

An expression union is denoted by `e1 + e2` where `e1` and `e2` are expressions. If `e1` and `e2` evaluate to functions f_1 and f_2 respectively, `e1 + e2` evaluates to the functional form

$$f_1 \oplus f_2 = \lambda x.\ f_1 x + f_2 x$$

Note that the + operator here is not the join operation on function domains since all function domains are flat domains.

TYPECAST EXPRESSION

Typecast expressions are denoted by `dname expr` where `dname` is a domain name and `expr` is an expression. There are two applications of typecast expressions:

(1) To inject an element into a union domain or a lifted or a topped domain.

(2) To provide local type information when the type of an expression cannot be inferred from context.

FIXED-POINT EXPRESSION

A fixed-point combinator is used in denotational semantics to represent iteration and recursion. The denotation of iterative and recursive constructs requires an equation of the form

$$f = F(f)$$

In domain theory, the meaning of such an expression is taken to be fix F, the least fixed point of the functional denoted by F. The fixed-point theorem tells us that if $F : D \rightarrow D$ is a continuous function and D is a pointed cpo[22] then the least fixed point fix F exists and is defined as

$$\text{fix } F = \bigcup_{n \geq 0} F^n(\bot).$$

The recursive equation $f = F(f)$ is usually represented by the notation fix $\lambda f.\ F(f)$ in denotational semantics.

[22]A partially ordered set D is a complete partial ordering (cpo) if and only if every chain in D has a least upper bound in D. A pointed cpo is a cpo with a least element.

Although the above definition is a constructive one, it is seldom computed if D is a function domain. Note that $\mathrm{fix}\ F$ is used to denote the meaning of a recursive function f. We usually require the values of $f(x)$ where $x \in X$ and X is a finite set rather than the value of the function f itself. Hence, operationally the expression $\mathrm{fix}\ f.\ F(f)$ is usually evaluated as a closure whose body is not evaluated until the function is applied. The application of this closure on some input x is computed by the recursive unfolding of the function f. It is important to note that the termination of this computation is not guaranteed by the existence of a fixed point. Computation by recursive unfolding terminates if and only if the least fixed point, $\mathrm{fix}\ F$, applied on the same input, is finitely computable.

In the specification of program analyses, algorithms are almost always designed to terminate for all programs even if the program itself may not terminate during execution. Several mechanisms are used to ensure termination. The use of the widening operator (∇) [11] is an example of such a method. The widening operator is used in at least one arc of every loop in the flow graph. This ensures that the information on that arc forms a strictly non-decreasing chain. Given a finite lattice, a lattice of finite height or a lattice that satisfies the ascending-chain condition[23], termination of the computation can usually be guaranteed. We provide the options described below to declaratively specify some termination mechanisms normally used in analysis algorithms.

Consider the problem of obtaining the least fixed-point solution to an equation of the form:

$$x = f(x)$$

If f is a continuous function, the least fixed-point solution can be computed as a limit of the following sequence (called the Kleene sequence):

$$\perp,\ f(\perp),\ f^2(\perp),\ \cdots$$

It can be easily verified that, if the above sequence converges in finite steps, the limit can be computed by evaluating $F(\perp)$ where F is a recursive function defined as:

$$F(x) \overset{df}{=} \quad (x = f(x)) \rightarrow x,\ F(f(x))$$

Observe that the test for convergence is essentially an equality test between arguments to two successive recursive calls. We can define a function F_1 that maintains the arguments to two successive recursive calls to f (i.e., two successive elements of the Kleene sequence) explicitly using two arguments. Consider,

$$F_1(p, x) \overset{df}{=} \quad (x = p) \rightarrow x,\ F_1(x, f(x))$$

Suppose f has the functionality $D \rightarrow D$. Then F_1 has the functionality $D_\perp \rightarrow D \rightarrow D$. It can be easily verified that

[23]Every strictly increasing chain is finite although the height of all chains may not be bounded by a constant.

$$F = F_1(\bot_{D_\bot})$$

Our goal is to provide a language construct to specify a recursive function such that its semantics includes a test for convergence. The function definition can then be written succinctly without explicitly providing the test for convergence. For example, a solution is to provide a new language construct so that we can write:

$$\overline{\textbf{fix}}\ F.\ \lambda x.\ F(f(x))$$

where the semantics of the $\overline{\textbf{fix}}$ specifies a source-to-source transformation of the above expression into the following expression:

$$\textbf{fix}\ F_1.\ \lambda p.\ \lambda x.\ (x = p) \rightarrow x,\ F_1(x, f(x))$$

For reasons that we will discuss shortly, we find that a transformation using a test for convergence of the form $(x = p) \rightarrow \bot, \cdots$ is more useful than the form $(x = p) \rightarrow x, \cdots$ used above. The **fix argument** expression is provided to perform such a transformation. We will describe the semantics of the **fix argument** expression below and show how a recursive function to compute the fixed-point solution to the equation $x = f(x)$ can be expressed using the **fix argument** expression. We will also provide the motivation for the use of a test for convergence of the form $(x = p) \rightarrow \bot, \cdots$.

The expression

$$\boxed{\textbf{fix argument}\ F.\ \lambda x.\ \cdots\ F(\cdots)\ \cdots}$$

defines a recursive function F such that for any x_0, $F(x_0)$ is evaluated by normal recursive unfolding except for one variation. Whenever, a recursive call to F is encountered in the body of the expression, the argument to the call is compared with the argument to the previous call to F. If they are the same (tested using the *equal* operation), then the recursive call is substituted with $\bot$. Otherwise, normal recursive unfolding occurs.

A **fix argument** expression is well-defined only when the recursive call occurs just once in the body of the expression. In other words, for some functions G and g, a **fix argument** expression must be of the form:

$$\boxed{\textbf{fix argument}\ F.\ \lambda x. G(F(g(x)), x)}$$

Suppose the function defined above has the functionality $A \rightarrow B$. The semantics of such an expression can then be expressed using the usual fixed-point expression as

$$\boxed{[\textbf{fix}\ F'.\ \lambda p.\ \lambda x.\ (x = p) \rightarrow \bot_B,\ G(F'(x, g(x)), x)](\bot_{A_\bot})}$$

where F' has the functionality $A_\perp \rightarrow A \rightarrow B$.

To illustrate, consider:

$$\boxed{\textbf{fix argument } F.\ \lambda x.\ F(f(x))+x}$$

where f is a function whose least fixed-point is to be computed. Let f have the functionality $D \rightarrow D$. The above expression defines the function $F'(\perp_{D_\perp})$ where

$$F' \overset{df}{=} \lambda p.\ \lambda x.\ (x = p) \rightarrow \perp_D,\ F'(x,\ f(x)) + x$$

Consider the evaluation of $F'(\perp_{D_\perp})(x_0)$ for some x_0. By successive recursive unfolding, we get

$$\begin{aligned} F'(\perp_{D_\perp})(x_0) &\overset{df}{=} (x_0 = \perp_{D_\perp}) \rightarrow \perp_D,\ F'(x_0,\ f(x_0)) + x_0 \\ &\overset{df}{=} (x_0 = \perp_{D_\perp}) \rightarrow \perp_D,\ (x_0 = f(x_0)) \rightarrow x_0,\ F'(f(x_0),\ f^2(x_0)) + f(x_0) + x_0 \\ &\quad \cdots \end{aligned}$$

It can be easily verified that the least fixed-point to the equation $x = f(x)$ can be obtained by evaluating $F'(\perp_{D_\perp},\ \perp_D)$. If the Kleene sequence for f is finite, then this evaluation terminates with the required least fixed-point.

In the next chapter, we will provide some examples to demonstrate the use of the argument option to specify analyses of loop constructs. In Chapter 6, we will provide some conditions under which a **fix argument** expression is finitely computable and is equivalent to the expression without the **argument** option. However to understand the motivation for the semantics provided for a **fix argument** expression, consider the flow graph for a while loop in Fig. 3.1.

The semantic function for an *any path* analysis on the while loop is usually expressed in the following form:

$$\textbf{fix argument } F.\ \lambda x.\ F(g(x))+h(x)$$

where g and h are some continuous functions. Intuitively, the expression to the left of the join operator provides the information derived by traversing the loop one or more times. The expression to the right of the join operator provides the information obtained assuming that the body of the loop is not evaluated. The value of $g(x)$ in successive recursive calls to F corresponds to the state at point A (Fig. 3.1) in successive iterations. Usually, termination is guaranteed by ensuring that $g(x)$ is a non-decreasing function. If all ascending chains in the range of g are finite, then for any x, $g(x)$ converges after a finite number of iterations implying that no more information can be derived from further iterations (or equivalently, from further recursive unfolding). The semantics of the **fix argument** expression ensures that the evaluation of the expression terminates at that point.

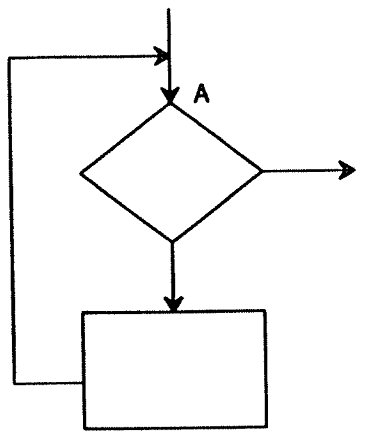

Figure 3.1. Flow chart for a while loop

The least element $\perp$ is usually used to model non-termination of target language constructs. The test for convergence of the form $x = p \rightarrow \perp, \cdots$ is consistent with the assertion that further recursive unfolding (and hence, further iterations of the loop) after convergence leads to non-termination. In any path problems, the information over all paths is "joined" together and consequently, the information from any infinite execution path is just ignored.

The fix **argument** semantics can be generalized to functions with more than one argument. The general form of a fix **argument** expression is as follows:

```
fix argument F. lambda x1. ... lambda xn. E
```

where `E` is an expression containing `F`. From a generalization of the semantics described above, this expression denotes the function

$$\lambda x_1 . \lambda x_2 . \cdots \lambda x_n . \mathrm{E}\,[x_i / \mathtt{xi}]\,[G(\perp_1, \perp_2, \cdots, \perp_n) / \mathrm{F}]$$

where

$$G = \lambda p_1 . \lambda p_2 . \cdots \lambda p_n . \lambda y_1 . \lambda y_2 . \cdots \lambda y_n .$$
$$(\bigwedge_{i=1}^{n} (p_i = y_i)) \rightarrow \perp, \mathrm{E}\,[y_i / \mathtt{xi}]\,[G(y_1, y_2, \cdots, y_n) / \mathrm{F}]$$

Often, some of the arguments to a fixed-point expression may be constant for all recursive calls. For example, a semantic function for a statement in an imperative language usually has an *environment* argument that binds identifiers to locations or procedure definitions.

```
S [[ stmt ]] : Env --> Store --> Store
```

If declarations are not allowed inside a loop statement, this argument remains constant for all recursive evaluations of the fixed-point expression corresponding to the loop. Although a **fix argument** expression can still be used in such a situation, our implementation will wastefully test for equality of the environment argument to successive recursive calls.

To allow for cases in which only one of the arguments to a fixed-point expression needs to be used in the test for convergence, we provide the **fix cache** construction.

A **fix cache** expression is meant to be used for definition of semantic functions (as opposed to auxiliary functions). The semantics of a **fix cache** expression is similar to that of a **fix argument** expression except that rather than test for the equality of arguments, the value currently collected in the cache is compared with the value in the cache at the time of the previous recursive call. Hence, the expression

$$\textbf{fix cache } F.\ \lambda x. G(F(g(x)), x)$$

is equivalent to the expression

$$[\textbf{fix } F'.\ \lambda p.\ \lambda x.\ (\$'cache[\$] = p) \rightarrow \perp, G(F'(\$'cache[\$], g(x)), x)](\perp)$$

Note that the expression $\$'cache[\$]$ is defined only when the above expression occurs in a semantic equation and the collect option has been used in the corresponding semantic function declaration. In the above example, to test for the convergence of just the store argument one can declare

```
S [[ stmt ]] : Env --> Store$1 --> Store$2 collect Store$1
```

and use a **fix cache** expression in the semantic equations for loop constructs.

An orthogonal option is to specify a bound on the length of any evaluation chain. Even if a least fixed point is finitely computable, its evaluation sequence may be arbitrarily large. For efficiency reasons, a decision is often made to settle for any fixed point (or at least an approximation that is guaranteed to be safe). The most common decision is to settle for $\top$. To enable bounds on the length of any chain that is computed, the bound option can be used. If the number of recursive unfoldings exceeds the bound, the next recursive call is replaced by the constant function $\top$. For example, consider

```
(fix argument bounded (50) F.lambda x.(F(f x)) + x) bottom
```

The least fixed point is obtained if the length of the Kleene sequence is less than 50. Otherwise, the expression evaluates to $\top$.

3.2. Formal semantics of the SPARE language

In the previous chapter we discussed the need for an operational definition of the SPARE specification language in order to reason about the information obtained from analysis specifications. One can provide the operational semantics by writing an interpreter for the specification language. However, an interpreter contains implementation details many of which are not necessarily required for the semantics. It is difficult to obtain the "minimum definition" from an interpreter to reason about the analysis specifications.

Moreover, because correctness is verified with respect to the standard semantics of the target language (the language in which the programs to be analyzed are written), which is expressed in the specification language, the question of whether the analysis information is valid for implementations of the target language that do not follow the same model as the interpreter is also difficult to answer.

This thesis does not address the problem of establishing correctness of analysis algorithms with respect to specific target language implementations. However, it is clear that implementation commitments in a semantics for the specification language must be minimized to allow analysis specifications to be meaningful for a wide variety of target language implementations.

On the other hand, using denotational semantics to provide the definition would just shift the problem onto the notation used in denotational semantics. Moreover, we require a static semantics for the specification language in order to define the type rules. Denotational frameworks are not convenient for expression of static semantics since type errors must be explicitly represented and propagated through all semantic equations.

We find a good compromise between operational and abstract notations in Natural Semantics [29] which has its origin in Plotkin's structural semantics [48]. The level of abstraction provided by a Natural Semantics specification is determined by the combinators available in the metalanguage. Natural Semantics has been used to provide formal semantics for ML [10,18]. Natural Semantics specifications can be used to prove the correctness of translations [12] or used as executable specifications [13]. However, the notation that we use is designed for clarity and conciseness rather than for execution. The goal is to express the semantics in a formalism that would facilitate the derivation of correctness proofs for translations for the SPARE language as well as correctness proofs for the analysis algorithms expressed in the SPARE language.

The semantic definition consists of a set of rules of the form

$$\frac{\text{hypothesis}}{\text{conclusion}}$$

The conclusion is a sentence (called a *judgment*) of the form

$$C \vdash \text{phrase} \Rightarrow C'$$

where C (called a *context*) and C′ are semantic objects. phrase is usually a syntactic object. The operator $\Rightarrow$ is overloaded. For example, in static semantics, a judgment for a syntactic phrase denoting an expression can be read as - "The phrase phrase has the type C′ in the *context* of C". In dynamic semantics, a judgment for a syntactic phrase denoting an expression should be read as - "The phrase phrase evaluates to C′ in the *context* of C".

The hypothesis is either empty, in which case the conclusion is an axiom, or it consists of a list of conditions — $c_1, c_2, \cdots, c_n$. The conclusion is a theorem if and only if all the conditions c_i can be derived as theorems. A condition c_i is either a *judgment* or a predicate in first-order logic.

Natural semantics is based on natural deduction [49] and the conclusion of a rule is established by deriving a proof tree such that the conditions in the hypothesis are satisfied. The rules that we provide for static semantics are intended to be interpreted in this fashion. However, in the proof scheme that we will develop in Chapter 6 for the verification of certain analysis specifications, we will assume a more operational interpretation of the rules for dynamic semantics. The conditions in the hypothesis are considered as a set of steps to be followed to evaluate the phrase in the conclusion. The steps can be performed in any order (unless an order is dictated by explicit dependences in the free variables used in the conditions). However, each step is assumed to be atomic. In other words, the evaluation corresponding to a condition must be completed before an evaluation corresponding to another condition can be started.

The formal semantics for the specification language is provided in [61].

Chapter 4
Example specifications

We will start with algorithms for conditional constant propagation [63, 64] to illustrate the features of the specification language as well as to indicate the nature of specifications for different constructs in the target language (the language for which the analyses are written). Other example specifications will be provided later in the chapter.

4.1. The target language

We will use a small imperative language whose abstract syntax is provided in Fig. 4.1. The language includes as base values integers and booleans. The identifiers can only be of integer types. The standard semantics for the language is expressed in the SPARE language in Fig. 4.2.

```
prog    ::=  (Prog stmt)

stmt    ::=  (Assign id exp)           Assignment statement
          |  (If b_exp stmt stmt)      Conditional Statement
          |  (While b_exp stmt)        Loop statement
          |  (Comp stmt stmt)          Statement Composition

exp     ::=  (ExpId id)
          |  (Int_Lit int)
          |  (Add exp exp)

b_exp   ::=  (Bool_Lit bool)
          |  (And b_exp b_exp)
          |  (Equal exp exp)
```

Figure 4.1. Abstract Syntax

```
Domains Declarations:

   bool, int, id, exp, b_exp, stmt, prog = syntactic;
   P_Store = store id --> integer;
   Prog_Store = lifted P_Store;

Semantic Function Declarations:

   E[[exp]] : P_Store --> integer;
   B[[b_exp]] : P_Store --> boolean;
   S[[stmt]] : strict Prog_Store --> Prog_Store;
   P[[prog]] : --> Prog_Store;

Semantic Function Definitions:

   P[[(Prog stmt)]] = S[[stmt]](⊥[]);

   S[[(Assign id exp)]] = λx. x[(E[[exp]]x)/[[id]] ];
   S[[(If b_exp stmt$2 stmt$3)]] =
      λx. B[[b_exp]]x --> S[[stmt$2]]x, S[[stmt$3]]x;
   S[[(While b_exp stmt$2)]] =
      fix F. λx. B[[b_exp]]x --> (F.S[[stmt$2]]x), x;
   S[[(Comp stmt$2 stmt$3)]] =
      λx. (S[[stmt$3]].S[[stmt$2]])x;

   E[[(ExpId id)]] = λx. x[ [[id]] ];
   E[[(Int_Lit int)]] = λx. [[int]];
   E[[(Add exp$2 exp$3)]] =
      λx. add (E[[exp$2]]x) (E[[exp$3]]x);

   B[[(Bool_Lit bool)]] = λx. [[bool]];
   B[[(And b_exp$2 b_exp$3)]] =
      λx. and (B[[b_exp$2]]x) (B[[b_exp$3]]x);
```

```
B[[(Equal exp$1 exp$2)]] =
    λx. (E[[exp$1]]x) = (E[[exp$2]]x);
```

Figure 4.2. Standard Semantics

For the SPARE specifications in this thesis, we will use the following notation for clarity. Limitations of current I/O devices require direct translations in actual use.

- Reserved words will be in bold.
- λ is used instead of the reserved word **lambda.**
- $\perp$ and $\top$ will be used instead of the pre-defined constants `bottom` and `top` respectively.

4.2. Conditional Constant Propagation

Conditional constant propagation algorithms can discover more constants than the simple constant propagation algorithm developed by Kildall [32] by evaluating all conditional branches with all constant operands. Parts of a program that are never executed are ignored. Hence, assignments in those parts cannot kill potential constants. For example, consider the code segment:

```
i := 1;
if i = 1
    then j := 1
    else j := 2
```

Evaluation of the conditional can show that `j` is never assigned the value `2` and in the absence of any further assignments to `j`, `j` is a constant.

We will first provide a specification (Fig. 4.3) for a naive version of a conditional constant propagation. The specification corresponds to an analysis that essentially simulates the execution of the program in a simple domain. The collect option is not used in this specification.

```
Domain Declarations:
    bool, int, id, exp, b_exp, stmt, prog = syntactic;
    con = lifted integer;
    Con = topped con;
    cstore = store id --> Con;
    Cstore = lifted cstore;
```

```
Semantic Function Declaration:
    P_ccp[[prog]] :  --> Cstore;
    S_ccp[[stmt]] :  strict Cstore --> Cstore;                -- (1)
    E_ccp[[exp]] :  strict Cstore --> Con;
    Bt_ccp[[b_exp]] :  strict Cstore --> Cstore;
    Bf_ccp[[b_exp]] :  strict Cstore --> Cstore;

Semantic Function Definitions:
    P_ccp[[(Prog stmt)]] = S_ccp[[stmt]](⊥[]);               -- (2)

    S_ccp[[(Assign id exp)]] =                                      -- (3)
      λx. let n=E_ccp[[exp]]x in x[(x[ [[id]] ]+ n)/[[id]] ];

    S_ccp[[(If b_exp stmt$2 stmt$3)]] =
      λx. let bt=Bt_ccp[[b_exp]]x; bf=Bf_ccp[[b_exp]]x in
                    (S_ccp[[stmt$2]]bt + S_ccp[[stmt$3]]bf);

    S_ccp[[(While b_exp stmt$2)]] =                           -- (4)
      fix argument F. λx.
            let bt=Bt_ccp[[b_exp]]x; bf=Bf_ccp[[b_exp]]x in
                (F(S_ccp[[stmt$2]](bt))) + bf;

    S_ccp[[(Comp stmt$2 stmt$3)]] =
      λx. (S_ccp[[stmt$3]] . S_ccp[[stmt$2]])x;

    E_ccp[[(ExpId id )]] = λ x. x[ [[id]] ];

    E_ccp[[(Int_Lit int )]] = λ x. Con [[int]];

    E_ccp[[(Add exp$2 exp$3 )]] =
      λx. let n1=E_ccp[[exp$2]]x; n2=E_ccp[[exp$3]]x in
            (n1 = ⊤) or (n2 = ⊤) --> ⊤, add n1 n2;
                -- add is strict in its arguments

    Bt_ccp[[(Bool_Lit bool )]] = λx. [[bool]] --> x, ⊥;
```

```
Bt_ccp[[(And b_exp$2 b_exp$3 )]] =
  λx. let bt1=Bt_ccp[[b_exp$2]]x; bt2=Bt_ccp[[b_exp$3]]x in
        (bt1 = ⊥) or (bt2 = ⊥) --> ⊥, x;

Bt_ccp[[(Equal exp$1 exp$2 )]] =
  λx. let n1=E_ccp[[exp$1]]x; n2=E_ccp[[exp$2]]x in
        (n1 = ⊤) or (n2 = ⊤) or (n1 = n2) --> x, ⊥;

Bf_ccp[[(Bool_Lit bool )]] = λx. [[bool]] --> ⊥, x;

Bf_ccp[[(And b_exp$2 b_exp$3 )]] =
  λx. let bf1=Bf_ccp[[b_exp$2]]x; bf2=Bf_ccp[[b_exp$3]]x in
        (bf1 = ⊥) and (bf2 = ⊥) --> ⊥, x;

Bf_ccp[[(Equal exp$1 exp$2 )]] =
  λx. let n1=E_ccp[[exp$1]]x; n2=E_ccp[[exp$2]]x in
      (not ((n1 = ⊤) or (n2 = ⊤)) and (n1 = n2)) --> ⊥, x;
```

Figure 4.3. Naive Conditional Constant Propagation

Con is the constant propagation lattice pictured in Fig. 2.5. $\perp$ in this lattice indicates that the identifier may or may not be a constant, while $\top$ indicates that the identifier is not a constant. Cstore maps identifiers to elements in Con. Identifiers mapped to $\top$ in the output of the analysis are not constants. An identifier mapped to an integer is a constant with that value. Identifiers are not mapped to $\perp$ unless they were used before their definition in the program.

The least element ($\perp$) of Cstore represents an unreachable path[24]. As all the semantic functions are declared to be strict in their arguments, information from program parts that are unreachable are never propagated. Unreachable states arise from the evaluation of the semantic functions for boolean expressions. The semantic function Bt_ccp outputs $\perp$ if the boolean expression contains all constant operands and evaluates to *false*. For all other expressions, it is an identity function. The semantic function Bf_ccp is

[24]For a lifted store, such as Cstore, $\perp[]$ denotes the empty store, while $\perp$ denotes the least element obtained from lifting.

similar but outputs $\perp$ when the expression evaluates to the constant *true*.

The argument option in the fixed point expression for the loop is used to ensure termination even if the loop never terminates in actual execution. It can be shown that the arguments to successive recursive calls form a strictly non-decreasing chain. When the arguments to successive recursive calls become the same no more information can be obtained from the loop (all recursive calls after that will have the same arguments). When the argument option is used, the call to `F` is replaced by the $\perp$ function which terminates the recursive call sequence.

An equivalent but less efficient alternative specification is to specify the collect option on the semantic function `S_ccp` to collect the input and use the cache termination option. As noted earlier, cache termination is more useful when there are many arguments to a function and only some of its components form a strictly non-decreasing chain.

4.2.1. Increasing precision

As explained earlier, although this naive algorithm can potentially detect more constants than Kildall's, it does not use information from conditions that contain identifiers that are constants locally but not globally. For example consider the code segment below:

```
i := 1; i := 2;
if i=2
   then j := 1
   else j := 2;
```

In the semantic equation for the assignment statement, the current assignment to the identifier is joined with any previous assignment to the same identifier. After the assignment `i := 1`, `Cstore` maps `i` to `1`. Since the semantic equation for the composition ensures that the evaluation corresponding to `i := 2` is carried out with this `Cstore`, the identifier `i` is mapped to $\top$ (join of `1` and `2`). The conditional is then evaluated using this value. The analysis does not use the fact that `i` is locally a constant in the conditional and assumes that both the branches are executed.

The reason for the join in the semantic equation for the assignment statement is to ensure that all semantic functions corresponding to statements are strictly increasing. This ensures that the fixed-point expression terminates, since the arguments to successive recursive calls form a strictly non-decreasing chain and converge for any finite program.

To improve this analysis to handle local constants, only the four numbered lines in Fig. 4.3. need to be modified. The modifications are shown in Fig.4.4. We will use `Cstore` to carry only local information and use the collect option to obtain the global information.

```
(1)   S_ccp[[stmt]] :  strict Cstore$1 --> Cstore$2
                                  collect Cstore$2;


(2)   P_ccp[[(Prog stmt)]] =
          let dummy=S_ccp[[stmt]](⊥[]) in
             +([[stmt]]'cache);


(3)   S_ccp[[(Assign id exp)]] =
          λx. let n=E_ccp[[exp]]x in x[ n/[[id]] ];


(4)   S_ccp[[(While b_exp stmt$2)]] =
        fix argument F. λx.
          let bt=Bt_ccp[[b_exp]]x;bf=Bf_ccp[[b_exp]]x in
                (F (S_ccp[[stmt$2]]bt + x) + bf);
```

Figure 4.4. Modifications to Fig. 4.3.

The semantic function S_ccp is modified to output the current assignments to the identifiers rather than a join of all previous assignments. The collect option is used to form a join of all previous states at each program structure.

(1) Modified declaration for S_ccp. The output of the semantic function is specified to be collected.

(2) Modified definition for P_ccp. As the output of evaluation of S_ccp on the body of the program is included in the cache, the value of dummy is not required. However, it is used in the let clause to force evaluation of the semantic function for the statement that forms the body of the program. The cache associated with the body of the program contains a map from labels for instances of statements to the local Cstores. The unary join operator forms a join of Cstores over all labels and provides the required mapping.

(3) Modified definition for the semantic function for the assignment statement. The join with the previous value associated with the identifier has been removed. Hence, Cstore at any point maps every identifier to a join of all the values that may actually reach that point.

(4) Modified definition for the semantic function for the loop statement. The strictly non-decreasing property mentioned earlier for the original semantic function (Fig. 4.3) for the loop no longer holds (due to the change in assignment). To restore this property the information coming into the loop from successive evaluations is joined together.

4.2.2. Handling non-recursive procedures

Analysis specifications expressed as abstract interpretations differ from standard semantics specifications in three aspects:

(1) Use of abstract domains. e.g. use of `Con` to abstract the integer domain.

(2) Merging of information from alternative paths. e.g. semantic function for the conditional statement in Fig. 4.3.

(3) Mechanisms to ensure termination. e.g. semantic function for the loop statement in Fig. 4.3.

Introduction of non-recursive procedures into a language does not result in any new situations where (2) and (3) are required. The semantic functions used in a standard semantics for procedure declarations, call statements and argument processing are independent of the domain of denotable values and hence carry over from the standard semantics specification to analysis specifications. Only the domains need to be modified to enable the collection of required information.

First, let us examine the domains used in a specification of a standard semantics specification. Because procedures can create aliases through the parameter mechanism, identifiers are usually mapped to locations which in turn are mapped to values. Aliased identifiers map to the same location. The semantic function for statement is declared as:

```
    S[[ stmt ]] : Env --> Strict Store --> Store;
```

The standard domains are defined below:

```
Locations = integer;
Proc = function Env --> Store --> Store;
Denotable_value = Locations + Proc;

Storable_value = integer;
Env = store id --> Denotable_value;
Value_Store = store Locations --> Storable-value;
Store = lifted Value_Store;
```

Suppose we wish to determine whether the identifiers used within the procedure as well as identifiers passed as parameters to a procedure are constants for any execution of the program. The most obvious required change is to abstract the storable values from integers to the domain Con.

```
Storable_value = Con;
```

Note that it is not sufficient to collect just the store at every statement to gather constant identifier names. We need both the environment and the store. Consider

```
Context = Env * Store;
S_ccp[[ stmt ]] : Env --> strict Store --> Context
                          collect Context;
```

Unfortunately, the above declaration does not work if an identifier is aliased to different identifiers in different instances of a procedure. The join in the domain of **Context** fails if an identifier is mapped to more than one location in successive evaluations since the join over locations (integers) is not defined. To correct this problem, a solution is to use a set of locations instead.

```
Locations = powerset of integer;
```

For each statement in the program (including statements in procedures), the cache now provides the set of locations that identifiers may be mapped to at that statement and whether these locations are constants. For statements within a procedure, the information is summarized over all possible calls to that procedure.

At the other extreme, one may wish to determine whether an identifier used in a procedure is a constant at each instantiation of the procedure. The solution is to generate unique locations to identifiers used in the procedure at every instantiation of the procedure. In standard semantics specifications, the denotation for a procedure contains functions to update the environment at entry and restore the environment at exit. The update function is changed to map all identifiers to the new locations and copy the values coming in into the new locations. The restore function is modified to copy the values available outside the procedure into the old locations.

Some applications may require the collection of information about the procedure summarized over all calls from each call-site. The solution is to use labeled locations.

```
      Locations = label * integer;
```

The semantic functions for statements now take an extra argument - a list of labels.

```
Label_list = list label;
S_ccp[[ stmt ]] :
     Env --> strict Store --> Label_list --> Context
                  collect Context;
```

The semantic function for a procedure call concatenates the label of the syntactic object corresponding to the call site to the head of the current list.

```
S_ccp[[(Call id args)]] =
  λe.λs.λl.e[ [[id]] ](A_ccp[[args]]e s)e s ($'label::l)
```

The update environment function assigns the same location number to each identifier used in the procedure but uses the label at the head of the label list to tag the locations.

4.2.3. Recursive procedures

The mechanisms described for handling non-recursive procedures also work for recursive procedures. The only additional problem is that of ensuring termination. As a procedure is entered, in general, from different call-sites one cannot ensure the ascending chain condition on the inputs for successive calls to the procedure. Hence, the procedure body cannot be denoted by a fixed point expression with either the argument or the cache termination option.

A solution is to use the bounded chain option. Constants are detected only if the length of the longest recursive chain is less than the specified bound. We will illustrate this solution for procedures with no parameters to avoid aliasing problems and the use of locations. Our solution can be extended to include the parameter mechanisms described earlier.

```
Proc = function Cstore --> Cstore;
Value = Con + Proc;
Cstore = store id --> Value;
D_ccp[[ procdecl ]] : Cstore --> Cstore;
D_ccp[[(Proc id stmt)]] =
  λs.let p=Value(Proc(fix bounded (50) λs.S_ccp[[stmt]]s)) in
           s[ p/[[id]] ];
S_ccp[[(Call id)]] = λs. s[ [[id]] ]s
```

4.3. Conditional Range Propagation

Constant propagation is an abstraction of range propagation. The constant propagation lattice can be expressed as an abstraction of the range propagation domain pictured below:

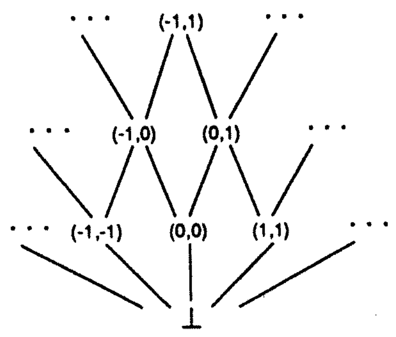

Figure 4.5. Lattice for Range Propagation

An abstraction function α maps elements in a range propagation domain to a constant propagation domain as follows:

$$\alpha(x) = \begin{cases} \bot & \text{if } x=\bot \\ n & x=(n,n) \\ \top & \text{otherwise} \end{cases}$$

The specification of conditional range propagation for the language in Fig. 3.1. is in Fig. 4.6.

```
Domain Declarations:
    bool, int, id, exp, b_exp, stmt, prog = syntactic;
    ordinal = integer ordered by arithmetic;
    range = ordinal * ordinal ordered by range_order;
    Range = lifted range;
    store =  store id --> Range;
    Store =  lifted store;

Auxiliary Function Declaration:
    Min : ordinal --> ordinal --> ordinal is
            λx.λy. ((x+y)=y) --> x, y;
            -- + is the join operator on ordinals
    Max : ordinal --> ordinal --> ordinal is
            λx.λy. ((x+y)=y) --> y, x;
            -- + is the join operator on ordinals
    Intersect : strict Range --> strict Range --> Range is
            λx.λy. ((Max x^1 y^1), (Min x^2 y^2));

Semantic Function Declaration:
    P_crp[[prog]] :  --> Store;
    S_crp[[stmt]] :  strict Store$1 --> Store$2 collect Store$1;
    E_crp[[exp]] :  strict Store --> Range;
    Bt_crp[[b_exp]] :  strict Store --> Store;
    Bf_crp[[b_exp]] :  strict Store --> Store;

Semantic Function Definitions:
    P_crp[[(Prog stmt)]] = let r=S_crp[[stmt]](⊥[]) in
                            (+[[stmt]]'cache)+r;

    S_crp[[(Assign id exp)]] =
      λx. let n=E_crp[[exp]]x in x[ n/[[id]] ];

    S_crp[[(If b_exp stmt$2 stmt$3)]] =
λx. let bt=Bt_crp[[b_exp]]x; bf=Bf_crp[[b_exp]]x in
        (S_crp[[stmt$2]]bt + S_crp[[stmt$3]]bf);


 S_crp[[(While b_exp stmt$2)]] =
   fix argument F. λx.
               let bt=Bt_crp[[b_exp]]x; bf=Bf_crp[[b_exp]]x in
                     (F (S_crp[[stmt$2]]bt + x) + bf);


 S_crp[[(Comp stmt$2 stmt$3)]] =
   λx. (S_crp[[stmt$3]] . S_crp[[stmt$2]])x;


 E_crp[[(ExpId id )]] = λ x. x[ [[id]] ];


 E_crp[[(Int_Lit int )]] = λ x. ([[int]], [[int]]);


 E_crp[[(Add exp$2 exp$3 )]] =
   λx. let n1=E_crp[[exp$2]]x; n2=E_crp[[exp$3]]x in
          ((add n1^1 n2^1), (add n1^2 n2^2))


 Bt_crp[[(Bool_Lit bool )]] = λx. [[bool]] --> x, ⊥;


 Bt_crp[[(And b_exp$2 b_exp$3 )]] =
   λx. let bt1=Bt_crp[[b_exp$2]]x; bt2=Bt_crp[[b_exp$3]]x in
          (bt1 = ⊥) or (bt2 = ⊥) --> ⊥, x;


 Bt_crp[[(Equal exp$1 exp$2 )]] =
   λx. let n1=E_crp[[exp$1]]x; n2=E_crp[[exp$2]]x in
          Intersect n1 n2;


 Bf_crp[[(Bool_Lit bool )]] = λx. [[bool]] --> ⊥, x;


 Bf_crp[[(And b_exp$2 b_exp$3 )]] =
   λx. let bf1=Bf_crp[[b_exp$2]]x; bf2=Bf_crp[[b_exp$3]]x in
          (bf1 = ⊥) and (bf2 = ⊥) --> ⊥, x;
```

```
Bf_crp[[(Equal exp$1 exp$2 )]] =
  λx. let n1=E_crp[[exp$1]]x; n2=E_crp[[exp$2]]x in
        (n1 = n2) --> ⊥, x;
```

Figure 4.6. Conditional Range Propagation

The domain `ordinal` is defined by imposing the arithmetic ordering on the domain of integers. We will assume that the arithmetic ordering definition is available externally. A range is represented by two ordinals. The domain of ranges is ordered using an external ordering definition. The range order is defined as

join = λx. λy. ((Min x^1 y^1), (Max x^2 y^2))
equal = λx. λy. ((x^1=y^1) **and** (x^2=y^2))
bottom = ?
top = ?

bottom and *top* are left undefined (defined to output an error value in practice [60]. The least element of `Range` represents unknown range information at reachable points. As in the conditional constant propagation specification, a lifted domain is used for the program store to avoid collecting information from unreachable paths.

The definitions of `P_crp` and `S_crp` are identical to the definition of `P_ccp` and `S_ccp` except for the collect option which specifies the just the inputs to be collected rather than the output. The change in the collect option requires a small change in the definition for `P_crp` to include the information at the end of the program. This collection is equivalent to the collection used for constant propagation. The alternative formulation is motivated by the modifications that will be provided in the next section to increase the precision of inference.

The semantic function `E_crp` approximates the standard semantic function `E` by defining the arithmetic operations over integer ranges. The semantic function `Bt_crp` and `Bf_crp` together approximate the standard semantic function `B`. However, they do not return boolean values. Since we would like to make the analysis flow-sensitive and use the information in boolean expressions, they act as filters. `Bt_crp` provides an approximation to the input states in which `E` on the same expression would evaluate to *true* while `Bf_crp` provides an approximation to the input states in which `E` would evaluate to *false*.

The domain for range propagation in Fig. 4.5 does not make use of the greatest element ⊤. In actual implementations, integer domains usually have bounds and the range containing all the possible integer values forms the ⊤. Although this ensures that all chains in the domain are finite, thus guaranteeing the termination of the fixed-point expression, in practice, the maximum length of any chain is too large to provide for an efficient analysis. We recommend the use of the bounded option in the semantic equation for the loop construct. The bound essentially determines the maximum number of times a loop may be unwound before the most general approximation is made.

4.4. Flow Analysis Algorithms

Both range propagation and constant propagation are examples of analyses that use simple abstractions of the standard domain. Next we will consider some examples of another class of analyses that use the notion of dependence between program points. Traditional flow analysis algorithms belong to this class.

Readers will have noticed in the previous specifications that information was propagated in the same direction as the control flow. Moreover, the information at any program point was a join over all information reaching that program point (i.e., the derived property was a function of information that was available along some path to the program point). In data flow analysis terminology, problems with these two characteristics are described by the terms *forward flow* and *any path* respectively. The characteristics are orthogonal and have inverses. In *backward flow* problems, the data flows in a direction opposite to that of control flow. In an *all paths* problem, the information at a program point must be a function of information that holds along all paths to the program point.

4.4.1. Use-Def analysis

We will start with Use-Def analysis which is a forward flow, any path problem and can be expressed naturally in the denotational framework. Use-Def associates with each program structure the set of possible definition sites for the identifiers used in the structure. The specification for obtaining Use-Def information is in Fig. 4.7.

```
Domain Declarations:
   id, exp, b_exp, stat, prog = syntactic;
   Label_Set = powerset of label;
   Def_Store = store id --> Label_Set;
   I_Store = store label --> Label_Set;
   Result_Domain = Def_Store * Label_Set;

Semantic Function Declarations:
   P_ud[[prog]] : --> I_Store;
   S_ud[[stmt]] : Def_Store --> Result_Domain
                  collect Result_Domain^2;
   B_ud[[b_exp]] : Def_Store --> Label_Set;
   E_ud[[exp]] : Def_Store --> Label_Set;

Semantic Function Definitions:
   P_ud[[(Prog stmt)]] =
      let x=S_ud[[stmt]](⊥[]) in [[stmt]]'cache;

   S_ud[[(Assign id exp)]] =
      λx. let L=E_ud[[exp]]x in (x[ {$'label}/[[id]] ],L);

   S_ud[[(If b_exp stmt$2 stmt$3)]] =
      λx. let Lb=B_ud[[b_exp]]x; Ls1=S_ud[[stmt$2]]x;
                  Ls2=S_ud[[stmt$3]]x in
                        ((Ls1^1 + Ls2^1),(Lb + Ls1^2 + Ls2^2));

   S_ud[[(While b_exp stmt$2)]] =
      fix argument F. λx.
            let Lb=B_ud[[b_exp]]x; Ls=S_ud[[stmt$2]]x in
                  (F(Ls^1)) + (x,((Lb + Ls^2)));

   S_ud[[(Comp stmt$2 stmt$3)]] =
      λx. let Ls1=S_ud[[stmt$2]]x in
                     S_ud[[stmt$3]](Ls1^1) + (⊥,Ls1^2);


   E_ud[[(ExpId id)]] = λx. x[ [[id]] ];

   E_ud[[(Int_Lit int)]] = λx. ⊥;

   E_ud[[(Add exp$2 exp$3)]] =
      λx. (E_ud[[exp$2]]+E_ud[[exp$3]])x;

   B_ud[[(Bool_Lit bool)]] = λ x. ⊥;
```

```
B_ud[[(And b_exp$2 b_exp$3)]] =
    λ x. (B_ud[[b_exp$2]]+B_ud[[b_exp$3]])x;

B_ud[[(Equal exp$1 exp$2)]] =
    λ x. (E_ud[[exp$1]]+E_ud[[exp$2]])x;
```

Figure 4.7. Use-Def Analysis

The result of the semantic function `S_ud` consists of two components. The first component is a map from the identifiers *defined* within the statement to the labels at which they are defined and downwards exposed. The second component is the set of labels of current definition sites for all identifiers *used* in the statement. The implicit cache collects the latter component and provides a map between each statement and the definition sites for all identifiers used within the statement. The semantic functions `E_ud` and `B_ud` compute the set of labels of all definition sites that reach the identifiers used in any expression.

4.4.2. Def-Use analysis

Def-Use analysis associates with each definition site, the set of use sites that may use the definition. This analysis is a backward flow, any path problem. Backward flow problems can be solved by finding an inverse to the analysis and then inverting the information obtained from it. The Use-Def analysis is an inverse of Def-Use analysis. The specification to create the Def-Use information from the Use-Def information is provided in Fig. 4.8. For each statement, `S_du` inverts the Use-Def information and computes Def-Use information. No semantic functions are required for expressions.

```
Domain Declarations:
   id, exp, b_exp, stat, prog = syntactic;
   Label_Set = powerset of label;
   I_Store = store label --> Label_Set;

Auxiliary Function Declarations:
   NewStore : label --> Label_Set --> I_Store is
      λL.λS. ⊥[ foreach l in S do L/l];

Semantic Function Declarations:
```

```
Semantic Function Definitions:
   P_rd[[(Prog stmt)]] =
      let t=P_iu[[$]]; dummy=S_rd[[stmt]]t^1 t^2 in
                [[stmt]]'cache;


   S_rd[[(Assign id exp)]] =
      λu.λs. (E_rd[[exp]]u)^1 + s[ [[id]] ];
      -- Expressions killed by id and expressions not computed
      -- in exp are unavailable after the assignment statement


   S_rd[[(If b_exp stmt$2 stmt$3)]] =
      λu.λs. let t=B_rd[[b_exp]]u in
                (S_rd[[stmt$2]] + S_rd[[stmt$3]])t^1 s


   S_rd[[(While b_exp stmt$2)]] =
      fix argument F. λu.λs.
            let t=B_rd[[b_exp]]u; s1=S_rd[[stmt$2]]t^1 s in
                F(s1 + u) + u;


   S_rd[[(Comp stmt$2 stmt$3)]] =
      λu.λs. S_rd[[stmt$3]](S_rd[[stmt$2]]u s)s;



   E_rd[[(ExpId id)]] =
      λu. ([[$]] in u) --> (u-{[[$]]},A_Status "unavail"),
                           -- id is unavailable
                           (u,A_Status "avail");
                           -- id is available


   E_rd[[(Int_Lit int)]] =
      λu. ([[$]] in u) --> (u-{[[$]]},A_Status "unavail"),
                           (u,A_Status "avail");


   E_rd[[(Add exp$2 exp$3)]] =
      λu. let e1=E_rd[[exp$2]]u;e2=E_rd[[exp$3]]e1^1 in
```

```
    ([[$]] in e2^1) -->
              (e2^1-{[[$]]},A_Status "unavail"),
              (e2^1,A_Status "avail");

B_rd[[(Bool_Lit bool)]] =
   λu. ([[$]] in u) --> (u-{[[$]]},A_Status "unavail"),
                             (u,A_Status "avail");

B_rd[[(And b_exp$2 b_exp$3)]] =
   λu. let b1=B_rd[[b_exp$2]]u;b2=B_rd[[b_exp$3]]b1^1 in
         ([[$]] in b2^1) -->
              (b2^1-{[[$]]},A_Status "unavail"),
              (b2^1,A_Status "avail");

B_rd[[(Equal exp$1 exp$2)]] =
   λu. let e1=E_rd[[exp$2]]u;e2=E_rd[[exp$3]]e1^1 in
         ([[$]] in e2^1) -->
              (e2^1-{[[$]]},A_Status "unavail"),
              (e2^1,A_Status "avail");
```

Figure 4.10. Available expressions

P_ie is a specification that provides a set of all expressions used in the program as well as a map between each identifier in the program and the set of all expressions used in the program. This specification is easily written as a bottom-up analysis of the abstract syntax tree and we will assume that it is available. The syntactic categories exp and b_exp must be ordered externally to allow for the equivalence of two different instances of a syntactic object. The pre-defined *equal* operator checks for the same instance of a syntactic object.

4.5. Issues of precision and efficiency

A significant amount of effort in the design of analysis algorithms is spent on arriving at a compromise between the efficiency of an algorithm and the precision with which information can be obtained from the algorithm. In this section we will explore the implications of using denotational frameworks on the precision and efficiency of algorithms specified within this framework.

4.5.1. Precision

Semantic functions in analysis specifications define mappings between sets of possible states (in the abstract domain). Approximations occur when operations are used to arrive at finite and/or efficient representations for sets of possible states. In the example specifications of the previous section, the join operator was used to merge sets of possible states into a single set. In conditional statements, the information from both branches were merged together. In loop statements, the information at the beginning of the loop from successive iterations were merged together. Join operation can potentially introduce imprecision. For example, if the range of an integer variable was computed as [1..4] and [8..10] in two branches of a conditional statement, the merge at the end would result in the range [1..10] which includes the range [5..7], and hence represents states that never actually occur[25].

When aggregate values such as stores are used, this imprecision may become significant. For example, merging the two stores $\{x \rightarrow [1..1], y \rightarrow [1..1]\}$ and $\{x \rightarrow [10..10], y \rightarrow [10..10]\}$ results in the store $\{x \rightarrow [1..10], y \rightarrow [1..10]\}$. Note that the original stores represent a set of two possible states while the result represents a set of 100 possible states only two of which may actually occur. In practice, this can result in very crude approximations. Consider the program segments

(a) `a:=1;`**while** `a<4` **do** `a:=a + a` **od**

(b) `a:=1;b:=1;`**while** `a<4` **do** `a:=a + 1;b:=b + 1` **od**

The value of `a` at the end of the code segment (a) will be approximated by the range `[4,6]`, while the value of `b` at the end of the code segment (b) will be approximated by `[1,∞]`. Although the approximation in (a) may not look particularly poor, the approximation in (b) is not very useful.

Possible solutions to this problem essentially consist of avoiding, or at least delaying, the merging of information as long as possible. Suppose a branch statement was followed by a block denoted by the function f. From the following lemma it follows that delaying the merge until the function f is evaluated on information from both branches separately will result in more precise (or at least as precise) information than what is obtained by evaluating f with the merged information.

Lemma 3.1: *If F: A → B is a monotonic function, then*

$$\forall a_1, a_2 \in A.\ f(a_1 \sqcup a_2) \geq f(a_1) \sqcup f(a_2)$$

Proof trivially follows from the definition of least upper bound and monotonicity.

There are two identifiable methods that address this issue. We will call them *control separation* and *data separation*. In *control separation*, information originating at disjoint program points (such as branches

[25]Note that an approximation is safe as long as the states that actually occur are included.

of a conditional statement) is propagated as far as possible independent of any other information that may flow along the same path. Hence, a node may be traversed many times, once for each possible thread of control. This is illustrated by the continuation-style denotational specifications described in [41].

In continuation-style semantics, the semantic function from any node is passed to its continuation. Since, the two branches of a conditional statement have their own continuations (albeit identical), the nodes in the continuations are interpreted separately for each source of information flowing into them. As pointed out in [41] the continuation-style semantics is capable of deriving more precise information than the direct-style semantics that we have used in the examples but suffers from possible non-termination. Since information corresponding to a node is distributed in separate control threads, selective merging of information to ensure termination is difficult in continuation-style semantics.

The *Static Single Assignment* (SSA) graph approach in [64] can be viewed as a control separation approach (although not a denotational one) with selective merging. In SSA graph construction, special nodes called Φ nodes are introduced at merge points. The Φ nodes have the capability to differentiate between information flowing in from different arcs. However, information flowing in through the same arc in successive evaluations (such as loop traversals) are not kept separate. This can be used to ensure termination since only a finite number of control threads need to be followed.

Similar results can be obtained in the data separation approach by structuring the information as tuples where each component of the tuple corresponds to single control thread. Semantic functions are defined to operate on tuples[26]. It is equivalent to the control separation approach in that all threads of controls are followed simultaneously. Non-termination due to an infinite number of control threads translates to a tuple whose cardinality increases indefinitely. However, since information for each program point is available together across all possible control threads, tuple size can be controlled by selective merging of components. Larger the maximum cardinality allowed for a tuple, greater the number of merges delayed and consequently greater the precision. On the other hand, computation effort increases since each element of the tuple must be processed separately. The specifications in this chapter are examples of selective merging in which the maximum cardinality of the tuple is 1.

To demonstrate the use of selective merging we will describe a modification to the conditional range propagation specification where we use a different merging scheme to increase precision. This technique is motivated by the difference in the standard semantics specification and the analysis specification for the while loop. In standard interpretation, the boolean condition in the while loop is evaluated for each possible

[26]This corresponds to the "relational attribute" method in [26].

4.5.2. Efficiency

Evaluation of denotational specifications corresponds to traversal of the control flow graph in which a semantic function corresponding to a node is evaluated whenever that node is visited. Often during evaluation, the derivation of information at some program points affects only a portion of the control flow graph. However, before the affected nodes are evaluated all the intervening nodes involved in the traversal to the affected nodes must also be evaluated.

This problem can be somewhat alleviated by the incremental evaluation scheme based on function caching that we describe in the next chapter. Unfortunately, most analysis specifications use aggregate values such as stores, which can make such incremental schemes ineffective. In an interactive system such as SPARE used for rapid prototyping of analysis algorithms, this inefficiency is not critical. However, in obtaining executable analysis programs from the specifications for other applications, mechanisms must be provided to avoid redundant computations. We have described one such mechanism in [59]. This thesis will not explore this issue any further.

Chapter 5
SPARE prototype implementation

SPARE was designed to be used in conjunction with the Synthesizer Generator [52]. In this chapter, we will provide some details about the following aspects of the prototype implementation:

- The editor available for construction of specifications.
- The interface between SPARE and the Synthesizer Generator.
- Translation from SPARE specification language to the Synthesizer Specification Language (SSL).
- The interpreter for the specifications and the effectiveness of using function caching for incremental evaluation.

5.1. An overview

A schematic diagram of the components of the SPARE system and the interface with the Synthesizer Generator is provided in Fig.5.1. SPARE specifications are constructed using a syntax-directed editor. The

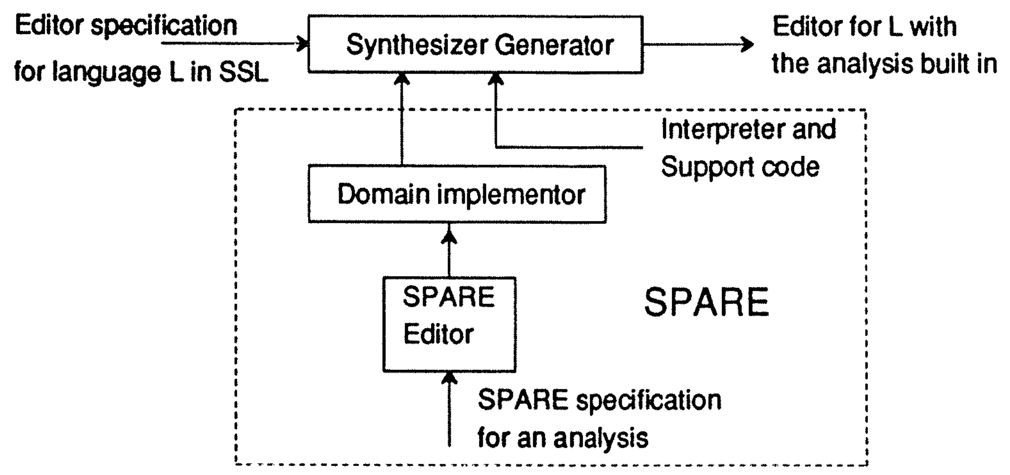

Figure 5.1. SPARE overview

SPARE editor translates function definitions into a representation that can be stored as attribute values in SSL specifications. The output of the editor is processed by the *domain implementor* which extracts information from domain declarations and generates SSL specifications for representations of domain values and operations defined on the domains. The SPARE interpreter is written as a function in SSL.

SSL specifications generated by the SPARE system and the interpreter definition are provided as input to the Synthesizer Generator along with an SSL specification for an editor for the target language. The output of the Synthesizer Generator is an editor for the target language with the specified analysis built in. This editor can be used to construct programs on which the analysis is performed and the results displayed. This facilitates testing of analysis algorithms without the need to provide implementations. Details about using the system are provided in the reference manual [60]. We will discuss some features of the various components of SPARE in the rest of the chapter.

5.2. SPARE Editor

The editor is constructed using the Synthesizer Generator. The SSL specification for the editor consists of about 2500 lines. The editor is syntax-directed and performs type checking and type inference as defined by the static semantics [61]. All expressions in a SPARE specification must be statically typed. If the type of the expressions cannot be inferred from context, the editor prompts the user to provide local typing information through type-casting (Section 3.1.4 in Chapter 3). Warning messages are output for type errors.

The specifications constructed using the editor can be written to files in three formats:

(1) The specification text: This is used to get a print copy of the specification and is created by using the write command with the *text* option.

(2) The specification structure: This allows specifications to be saved for future edits. It is created by using the write command with the *structure* option. Specifications saved in this format can be read back into the editor.

(3) The specification translation: This output consists of SSL specifications for function definitions and domain information for use by the *domain handler*. This file is created using the write command with the *text* option on the alternate unparsing scheme for the specification (see [52] for details).

In the translation, two attributes are declared for each nonterminal in the target language for which a semantic function is defined. One attribute is used to label the instance of a node in the abstract syntax tree. The other attribute is used to store the representation of the semantic function corresponding to that node. Attribute equations are provided in the translations to define these attributes for each production in the target language. Information from domain declarations is provided in a condensed form for use by the *domain* *implementor*.

5.3. Domain Implementor

The *domain implementor* is written in Pascal and consists of about 800 lines of source code. The *domain implementor* takes as input the translation provided by the editor. The domain information is stripped from the translation and the attribute declarations and equations are passed through without change. The domain information is used to generate SSL specifications to

- Define representations for values in the domains declared in the specification.
- Define the operations <*join, equal, bottom, top*> that represent the structure of the domains.

The domain value representations are tagged with domain names provided in the domain declaration. Although the tags are not required for correctness of the implementation, they are useful for two reasons:

(1) During the development of the system, tags were used to ensure that the static type checking was sufficient and did not allow any type errors to occur during execution.

(2) During use of the system, tags allow information from the analyses to be displayed with different schemes for distinct domains even if they have the same structure. For example, stores used to represent different information can be displayed in a manner that is natural to the information.

5.4. Interpreter

The interpreter is written in SSL and consists of about 900 lines of source code out of which about 300 lines support various operations on domains independent of any specification. The interpreter is written to be consistent with the evaluation rules specified in the dynamic semantics of the SPARE language [61].

SPARE is designed to be used for testing analysis algorithms rather than implementing them for use in other applications. Hence, the emphasis was more on correctness than efficiency of evaluation. For example, all domain values are currently represented using lists. More efficient implementations can lead to substantial improvements in the efficiency of analysis. However, we decided to include an incremental evaluation mechanism to explore the benefits of using such a mechanism for a functional language.

5.4.1. Incremental evaluation

There have been several studies on the use of function caching [36] to avoid redundant computation [30,38] in functional programs. The use of function caching as an incremental evaluation mechanism has been explored in [50]. Function caching is generally applicable for functions that are side-effect free and have no non-local accesses. There are two features in our language that can potentially create functions with side-effects.

Collect Option

The collection of intermediate evaluation can be considered as a side-effect of function evaluation. However, as noted in [38], such side-effects can be removed by considering the *collect cache*[29] as an extra parameter to all functions. As our implementation essentially treats a collect cache in this fashion, the collect mechanism does not create any problems.

Fixed-Point termination options

The use of termination options in fixed-point expressions create functions with side-effects. For example, if the bounded termination option is used in a fixed-point expression, the result of evaluation of such an expression depends on previous evaluations of that expression. Once again, this problem can be removed by considering the information related to termination (such as the number of previous evaluations, the arguments to the previous evaluation, etc.) as being included in an extra parameter to a fixed-point expression.

However, if structures that contain closures of fixed-point expressions are passed as parameters to other functions, then the test for equality of arguments becomes more complicated. Note that closures are created during the definition of a fixed-point expression and not during evaluation of that expression. Hence, information related to termination is not part of the closure and must be maintained dynamically in a separate environment. To test for equality of arguments to a function one must detect the presence of fixed-point expression closures and access the corresponding termination information for use in the test for equality. This leads to an inefficient computation for equality testing and negates the advantages from caching even when fixed-point expressions are not passed as parameters.

To avoid this problem, our implementation includes in the check for equality of arguments, the check for equality of termination information related to all fixed-point expressions active at that point. In practice, this results in the incremental evaluation being ineffective for function evaluations within fixed-point expressions. However, we find that the use of aggregate values as parameters to functions already defeats the incremental evaluation scheme inside fixed-point expressions and our decision to test for all activations has very little impact on the efficacy of the incremental scheme.

[29]We will call the cache used for collecting intermediate information, the *collect cache* to distinguish it from the cache used for incremental evaluation.

5.4.1.1. An example measurement

This section describes the procedure used for measuring the performance of the incremental scheme. We demonstrate the typical behavior of the incremental scheme using an editor generated with the specification for the conditional constant propagation algorithm in Chapter 4. The small program in Fig.5.2 is used as a starting point and the modifications in Fig.5.3 are successively applied to this program. The profiling mechanism available in the Synthesizer Generator is used to measure the performance of the interpreter both with and without the incremental scheme. We will summarize the results obtained by using several test programs in the next section.

```
(1)  program;
(2)  begin
(3)     a := 1;
(4)     b := a;
(5)     a := 3;
(6)     c := a;
(7)     if c = 3 then
(8)        d := 10;
(9)     else
(10)       d := 5;
(11)    fi;
(12)    while a = 3 do
(13)       a := a + 1;
(14)       e := c + d;
(15)    od;
(16)    f := a + b + c + d;
(17) end.
```

Figure 5.2. Test program

Mod. No.	Line affected	Old line	New line
1.	16	`f := a + b + c + d;`	`f := a + b + c + d;`
2.	16	`f := a + b + c + d;`	`f := b + c + d;`
3.	3	`a := 1`	`a := 2`
4.	7	**if** `c = 3` **then**	**if** `c = 5` **then**
5.	12	**while** `a = 3` **do**	**while** `a = 5` **do**

Figure 5.3. Modifications to the test program

Modification 1:

The line is replaced with an identical one. This forces a re-analysis of the program. This should give an idea of the fixed costs associated with the incremental scheme.

Modification 2:

The expression is replaced with a simpler expression. As the modification is at the end of the program, the incremental scheme should be effective.

Modification 3:

The constant assigned to a is changed. Interpretation without caching should not be affected. The caching scheme is not expected to perform well here since the change is reflected in inputs to all functions.

Modification 4:

The boolean condition is changed so that the false branch will always be taken. The caching scheme should be useful here since the program segment above the change is not affected.

Modification 5:

The boolean condition is changed so that the statements within the loop are never executed. This should demonstrate the usefulness of the caching scheme on primarily straightline code.

The metric used is the number of interpreter op-codes executed in the Synthesizer Generator. This metric is claimed to be a "fairly good" estimate of the amount of time spent for execution [52]. The results are pictured in Fig. 5.3.

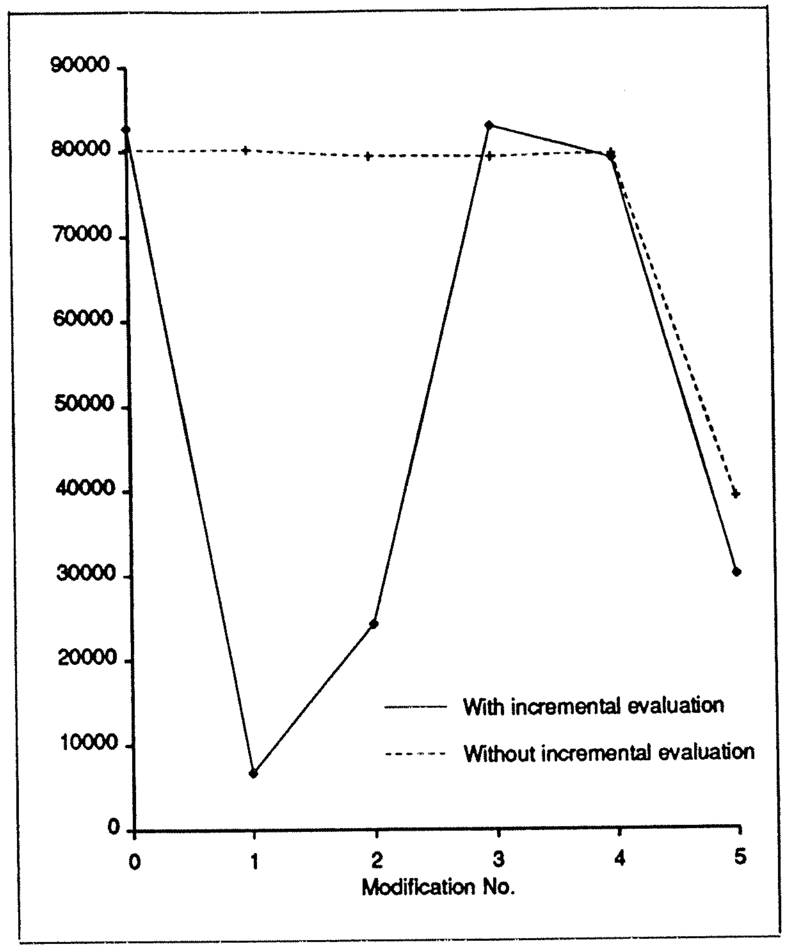

Figure 5.3. Efficacy of the cache scheme

Summary of measurement

Base Program:

Complete evaluation required for both interpreters. The overhead for creating the cache is about 3%.

Modification 1:

The incremental scheme is extremely effective. The fixed costs involved in cache lookups is about 8.5% of the cost of re-evaluation.

Modification 2:

The incremental scheme is still effective. The incremental evaluation saves about 60% of the cost of re-evaluation.

Modification 3:

The propagation of new information through aggregate values makes incremental evaluation ineffective. The reconstruction of the entire cache involves an overhead of 4.5%.

Modification 4:

The cost saved by incremental evaluation is less than 1%.

Modification 5:

The incremental evaluation avoids about 23% of the re-evaluation.

5.4.1.2. Summary of test results

We measured the effectiveness of the incremental scheme using an editor with conditional constant propagation as well as an editor with Use-Def analysis. The test programs ranged in size from 10 to 50 lines[30]. The measurements are summarized in Fig. 5.4.

First-time evaluation:

This column provides the range of overhead costs that the incremental scheme incurs when a program is evaluated for the first time. The overhead is for the creation of the cache. The larger cache generated for Def-Use analysis explains the higher overhead cost as compared to constant propagation.

Worst-case evaluation:

This column provides the range of overhead costs for re-evaluation using the incremental scheme when a program is modified so as to make the entire cache useless for re-evaluation. Combined lookup and update costs result in higher overhead costs than the first-time evaluation.

Analysis	First-time evaluation (overhead)	Worst-case re-evaluation with 0% cache hits (overhead)	Best-case re-evaluation with 100% cache hits (savings)
CCP	3-16%	4-30%	89-95%
Def-Use	21-34%	28-38%	95-98%

Figure 5.4. Effectiveness of incremental evaluation

[30]We expect the programs used for testing of analysis algorithms through SPARE to be within this range.

Best-case evaluation:

This column provides the range of savings for incremental re-evaluation relative to non-incremental evaluation when changes to the program do not affect the information obtained from the analysis. For example, replacing a constant with another does not affect the information obtained from Def-Use analysis. The small costs involved in this case arise from the equality testing and cache lookups.

Other observations:

(1) The effectiveness of the incremental-scheme depends on the precise nature of modifications and the structure of the program.

(2) Use of aggregate values as inputs to functions decreases the effectiveness of incremental schemes using function caching.

(3) Fixed-point evaluations are expensive and our incremental scheme helps only if the inputs to a fixed-point expression do not change. It is not helpful during the evaluation of the fixed-point itself.

Our experience with the incremental evaluation scheme suggests that function caching can be useful for evaluation of SPARE specifications.

Chapter 6
Formal verification

In addition to testing, formal verification can result in greater confidence about an analysis algorithm. The choice of a denotational framework for this thesis was motivated in part by the availability of standard proof techniques such as structural induction, fixed-point induction, etc. In this chapter, we provide the formalism necessary to express and verify the correctness of analysis algorithms expressed as SPARE specifications. Proofs of correctness are provided for some example specifications in Chapter 4. These proofs can be used as guidelines for construction of correctness proofs for other algorithms.

The correctness proofs provided in this Chapter are of two kinds. The proofs for flow analysis algorithms are based on reasoning about sequences of evaluations of semantic functions in both standard and analysis (abstract) interpretations. On the other hand, the proofs for the constant propagation and range propagation specifications follow the more traditional approach of reasoning about information computed by semantic functions.

In the proof scheme that we develop for the flow analysis algorithms we make certain assumptions:

(1) The formal semantics rules for the SPARE language do not specify the order of evaluation of terms within certain expressions. Although, we allow for such evaluations to take place in any order, we assume that these evaluations are performed sequentially and not interleaved with each other. This assumption implies that the correctness proofs for analysis specifications do not apply to implementations of the target language that perform certain operations in parallel. However, if such parallel evaluations are performed such that the semantics is always consistent with a sequential implementation, then the correctness proofs may still apply.

(2) We ensure that the begin and end of evaluations of semantic functions are well defined by assuming that the target language has only structured constructs and that semantic functions are not Curried. This assumption implies that the begin and end of evaluations of semantic functions have the structure of nested parentheses.

In Section 6.1, we provide the formalism necessary to express the correctness of flow analysis specifications. This formalism is used in Section 6.2 to provide the correctness proofs for flow analysis algorithms. Section 6.3 provides an useful result to relate **fix argument** expressions to the traditional fixed-point expressions. This result is used in the correctness proofs for conditional constant propagation specifications in Section 6.4.

6.1. Definitions and Notation

All specifications are written for a target language defined by a grammar $G = < \Sigma_n, \Sigma_t, \gamma, \Pi >$ where

Σ_n is the set of nonterminal symbols.

Σ_t is the set of terminal symbols.

$\gamma \in \Sigma_n$ is the goal symbol.

Π is the set of productions.

As our interest is in the analysis of programs that are syntactically correct, we will assume that programs are represented as derivation trees generated according to the grammar G. The nodes in the derivation tree are labeled using elements from the set $\Psi_t \cup \Psi_n$ where[31]

$$\Psi_t = \Sigma_t \times \mathbf{N} \text{ and } \Psi_n = \Sigma_n \times \mathbf{N}$$

The notion of a program point in analysis algorithms generally depends on the application. To abstract this notion we will use the following definition:

Definition 6.1: *(Program points)*

The set of program points PP_p *for a program* p *is defined as*

$$PP_p = \Psi_n \oplus \Psi_n.$$

The elements of the disjoint sum are denoted by

$$(\mathbf{begin}, s) \textit{ or } (\mathbf{end}, s)$$

where s *denotes an element in* Ψ_n.

We assume that only nonterminals correspond to program points of interest and there exist functions to map elements of PP_p to "program points" defined by the application. For example, the elements of PP_p can be uniquely mapped to statement numbers in an imperative program or to labeled expressions in a functional program.

Definition 6.2: *(Interpretation)*

An interpretation, I_S *of a program* p *under a SPARE specification* S *is an evaluation of the denotation of the program* $M_S [\![p]\!]$ *according to the formal semantics of SPARE.*

Definition 6.3: *(Execution trace)*

The execution trace of an interpretation I_S *of a program is a (possibly infinite) sequence of program points* $\pi_1, \pi_2, \cdots$ *where* $\pi_i = (\mathbf{begin}\ s)$ *and* $\pi_k = (\mathbf{end}\ s)$ *denote the begin and end of evaluation respectively of the body of a semantic function associated with the node* s. *Without loss of generality we assume that there is a*

[31]The symbol **N** denotes the set of natural numbers.

single instance of the goal nonterminal. Hence,

$$\pi_1 = (\textbf{begin}, (\gamma, 0))$$

If the interpretation terminates then the last program point $\pi_k = (\textbf{end}, (\gamma, 0))$.

If the denotation of a program contains expressions whose evaluation order is unspecified by the semantics of the SPARE language (e.g., expression union), an interpretation may correspond to more than one execution trace.

In general, it is not possible to determine statically the set of all execution traces that actually occur in some interpretation. However, one can consider possible execution traces assuming that every applicable rewrite rule for an expression is possible. In particular, each conditional expression gives rise to two possible execution traces while fixed-point expressions result in an infinite number of possible execution traces each corresponding to a distinct number of recursive unfoldings.

Definition 6.4: *(Possible execution trace)*

A sequence of program points is called a possible execution trace of an interpretation I_S *of a program* p *iff the sequence corresponds to a possible evaluation of the denotation of the program assuming that every expression can be evaluated with any of the applicable rewrite rules.*

Definition 6.5: *(Control graph)*

The control graph $CG_S(p)$ *of a program* p *with respect to a specification* S *is the set* PP_p *ordered by the relation* $\trianglelefteq$ *as follows:*

for all $\pi_1, \pi_2 \in CG_S(p)$, $\pi_1 \trianglelefteq \pi_2$ *iff* π_1 *occurs immediately before* π_2 *in some possible execution trace for an interpretation* I_S *of the program* p.

The element $\pi = (\textbf{begin}, (\gamma, 0))$ *is called the root of the control graph.*

Note that the ordering $\trianglelefteq$ is neither reflexive, transitive nor anti-symmetric.

Definition 6.6: *(Control chain)*

A sequence of program points $\pi_1, \pi_2, \cdots$ *is called a control chain of a control graph* $CG_S(p)$ *iff*

(1) π_1 *is the root of the control graph.*

(2) For all successive program points π_i *and* π_{i+1} *in the sequence,* $\pi_i \trianglelefteq \pi_{i+1}$.

By definition, it follows that every possible execution trace of an interpretation I_S of a program p is a control chain of the control graph $CG_S(p)$. In general, a control graph consists of control chains that do not correspond to any possible execution trace. As we are usually interested in proving that some property holds for all possible execution traces of an analysis interpretation, we will restrict the set of control chains considered for analysis specifications to *maximal chains* defined as follows:

Definition 6.7: *(Maximal chains)*

A chain X *in a control graph* $CG_S(p)$ *is called a maximal chain iff*

(1) *Every program point* $\pi \in PP_p$ *is in* X.

(2) *Every loop corresponding to a* **fix argument** *expression is unwound at least twice.*

The motivation for (1) comes from the observation that analysis algorithms usually visit every program point at least once. If the evaluation of a **fix argument** expression terminates due to the equality of the argument to two successive recursive calls, the semantics of the expression does not change if the expression is recursively unfolded once more before termination[32]. Hence, we assume that all **fix argument expressions** are recursively unfolded at least twice.

Definition 6.8: *(Control graph abstraction)*

A control graph $CG_S(p)$ *is said to be an abstraction of the control graph* $CG_{S'}(p)$ *iff*

If there is an occurrence of a program point π_1 *before the occurrence of a program point* π_2 *in some control chain of* $CG_{S'}(p)$*, then there is an occurrence of* π_1 *before* π_2 *in every maximal control chain of* $CG_S(p)$

To establish the control abstraction relation finitely, the following lemma is useful:

Lemma 6.1: *(Program point dependence)*

Let C *be some control chain of a control graph* $CG_S(p)$. *For all* $\pi_1, \pi_2 \in C$, *if there is an occurrence of* π_1 *before an occurrence of* π_2 *in* C, *then there is a control chain* C′ *of* $CG_S(p)$ *such that* $\pi_1, \pi_2 \in C'$ *and* π_1 *occurs before the second occurrence of* π_2.

Proof: If π_1 occurs before the second occurrence of π_2 in C, then C′ = C. Otherwise, C must be of the form

$$\alpha\,\pi_2\,\beta\,\pi_2\,\gamma\,\pi_1\,\delta\,\pi_2\,\cdots$$

where α, β, γ and δ are finite sequences of program points and $\pi_2 \notin (\gamma \cup \delta)$. By the definition of the control graph, the following must also be a control chain of $C_S(P)$:

$$\alpha\,\pi_2\,\gamma\,\pi_1\,\delta\,\pi_2\,\cdots$$

If π_1 occurs before the second occurrence of π_2 in the above sequence, then the proof is complete. Otherwise, the above step can be repeated a finite number of times until the required control chain is obtained.

[32]We assume that there are no side-effects and use of global variables. This assumption is not valid if the semantic functions evaluated within the fixed-point expression *use* information in the cache. As such an use is not a common occurrence for flow analysis specifications, we assume that correctness proofs for specifications that have such uses will be constructed differently from the general scheme that we provide here.

To establish control dependences between any two program points for *any path* problem specifications, we can use this lemma to consider only control chains that have at most three occurrences of every program point. For finite control graphs, only a finite number of such control chains are possible.

6.2. Flow analysis specifications

6.2.1. Use-Def analysis

This section provides the correctness proofs for the Use-Def analysis in Fig. 4.7. The label operator in SPARE provides an unique label for each node in the derivation tree. However, each node in the derivation tree corresponds to two program points — one at the beginning and one at the end of the structure. In this section, we assume that the label generated by SPARE always denotes the program point at the beginning of the structure.

An outline of the proof is as follows:

(1) If a definition site $\pi =$ (begin, (Assign i e)) reaches a statement s in some standard interpretation and s uses i, then there is an evaluation of the semantic function S_ud⟦s⟧x, x $\in$ Def_Store in every possible evaluation of P_ud⟦p⟧ such that $\pi \in$ S_ud⟦s⟧x$\downarrow$2.

(1.1) If a definition site $\pi =$ (begin, (Assign i e)) reaches a statement s in some standard interpretation, then there is an evaluation of the semantic function S_ud⟦s⟧x, x $\in$ Def_Store such that $\pi \in$ x[i] (Lemma 6.4).

(1.1.1) If there is an occurrence of a program point π_1 before a program point π_2 in some possible execution trace of P⟦p⟧, then in every possible execution trace of P_ud⟦p⟧, there is an occurrence of π_1 before π_2 (Lemma 6.2).

(1.1.2) If a definition at program point π is not killed in some segment $\pi_1, \cdots, \pi_2$ of some possible execution trace of P⟦p⟧, then there is a segment in every possible execution trace of P_ud⟦p⟧ starting with π_1 and ending with π_2 such that if π is included in a Def_Store available at π_1, then π is included in a Def_Store available at π_2 (Lemma 6.3).

(1.1.3) A program point $\pi =$ (begin, (Assign i e)) is included in a Def_Store available at $\pi' =$ (end, (Assign i e)).

(1.2) If there is an evaluation of the semantic function S_ud⟦s⟧x with x $\in$ Def_Store for some statement s and s uses an identifier i, then x[i] $\subseteq$ S_ud⟦s⟧x$\downarrow$2 (Lemma 6.7).

(2) An evaluation of the semantic function S_ud⟦s⟧x, x $\in$ Def_Store for the body of the program s, associates with s, a cache c such that, if there is an evaluation of the semantic function S_ud⟦s_1⟧x', x' $\in$ Def_Store for every s_1 occurring in s (including s itself), then S_ud⟦s_1⟧x'$\downarrow$2 $\subseteq$ c[s_1'label] (semantics of the collect option).

(3) The evaluation of P_ud⟦p⟧ for any program p returns the cache obtained in step (2) above.

Lemma 6.2: *(Control Abstraction)*

For any program $p \in prog$, *the control graph corresponding to the analysis* $CG_{ud}(p)$ *is an abstraction of the control graph corresponding to the standard semantics* $CG_s(p)$.

Proof:

As we are interested in dependences between statements in the program we will consider only those program points that correspond to statements[33].

We must show that if a program point π_1 occurs before π_2 in some control chain of $CG_s(p)$, then π_1 occurs before π_2 in every maximal control chain of $CG_{ud}(p)$. The proof is by structural induction. We consider control chains within each statement structure. As the statements have a single entry and a single exit, the dependences between program points in different structures are established through the entry and exit program points through transitivity.

The control graph schema for each statement is pictured in Figs. 6.1(a) and 6.1(b). Only the entry and exit nodes of the component structures need to be considered. As mentioned earlier, the dependences with program points inside the component structure are automatically established by transitivity.

It is trivial to verify the dependences in all control chains for assignment, conditional and composition statements. We will provide the proof for the loop statement.

We will consider only those control chains in the control graph for the loop statement that have at most two occurrences of every program point in them. We can then use Lemma 6.1 to argue that Lemma 6.2 is true for all other control chains. There are only three such control chains. They are pictured below with B and E denoting **begin** and **end** respectively. $< \cdots >^2$ denotes a sequence that is repeated twice.

(1) (B, (While b s)), (E, (While b s))

(2) (B, (While b s)), (B, s), (E, s), (B, (While b s)), (E, (While b s))

(3) <(B, (While b s)), (B, s), (E, s)>^{2}, (B, (While b s)), (E, (While b s))

There is only one maximal control chain that we need to consider for the analysis specification:

(1) <(B, (While b s)), (B, s), (E, s)>^{2}, (E, (While b s))

Although not required here, the proof can be easily extended to show that the lemma holds even when the program points corresponding to expressions are included in the control graph.

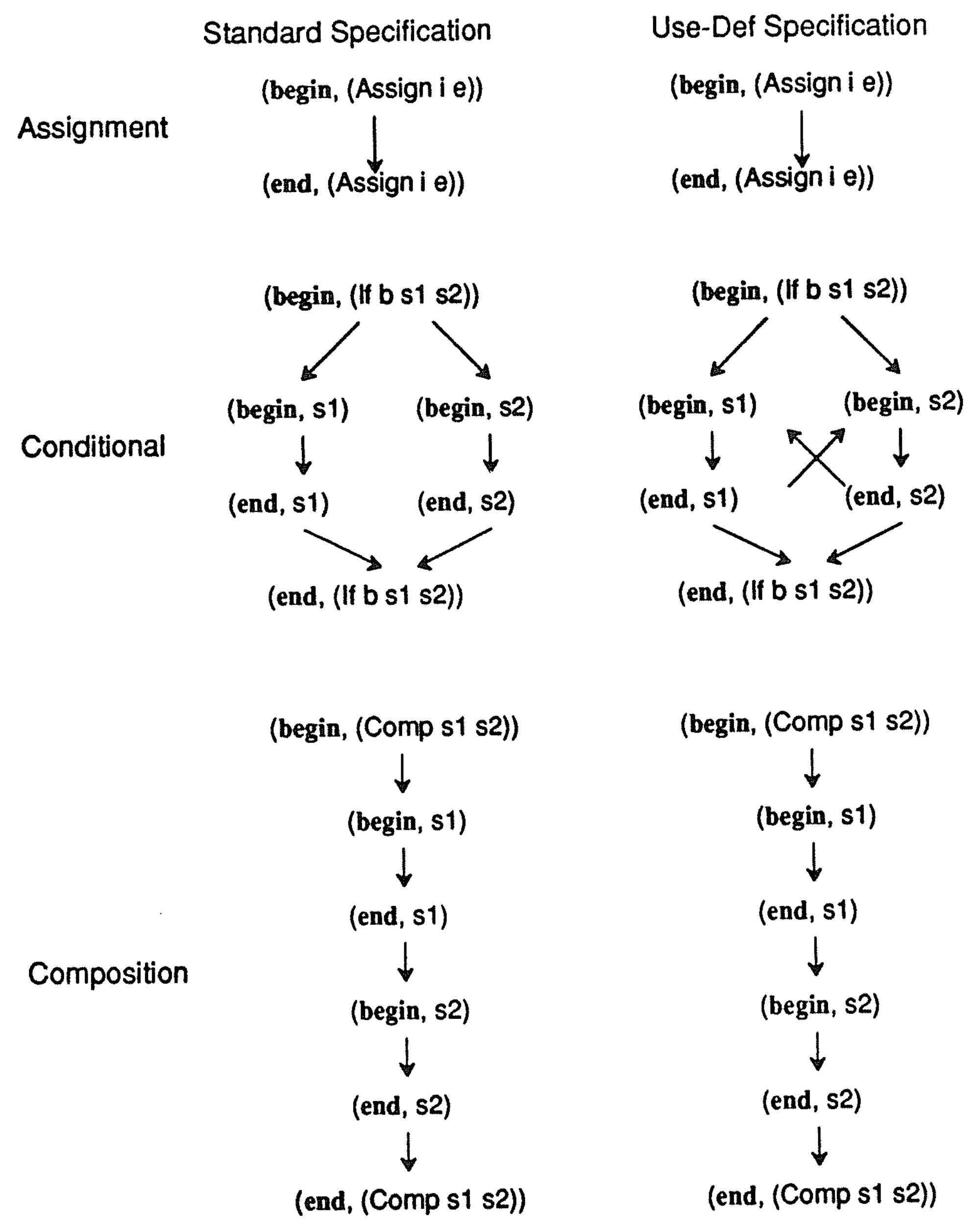

Figure 6.1(a). Use-Def control graphs

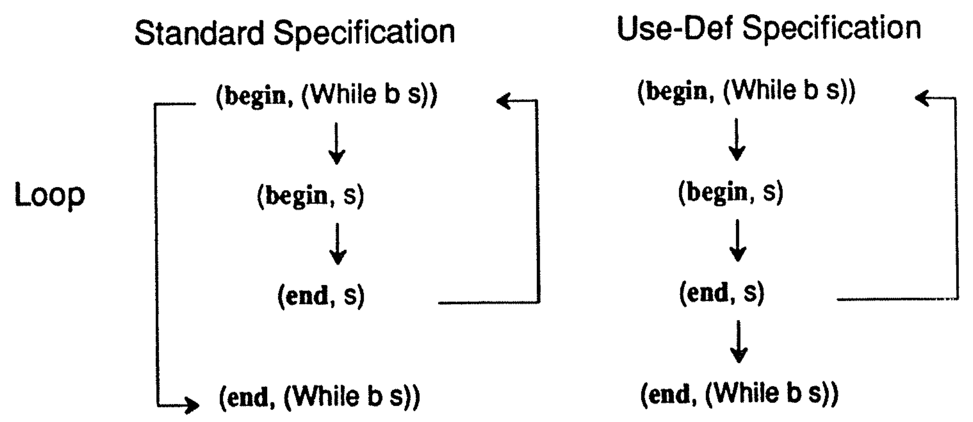

Figure 6.1(b). Use-Def control graphs

By comparing the above chain with the three control chains corresponding to the standard semantics specification, it is easy to verify that for every occurrence of program point π_1 before π_2 in any of the three control chains, there is an occurrence of π_1 before π_2 in the maximal chain corresponding to the analysis specification. From Lemma 6.1 it follows that this property holds for all control chains corresponding to the standard semantics of the loop statement. From the structural induction hypothesis and transitivity of the occurrence relation, it follows that this property is true for all control chains of the control graph for the loop statement when the control graph corresponding to the body of the loop is also included.

The lemma then follows from the definition of control graph abstraction.

Definition 6.9: *(π-free)*

Let $\pi_1, \cdots, \pi_k$ *be some sequence of program points of a program* p. *The sequence is said to be π-free iff*

(1) π *is of the form* (**begin**, (Assign i e)) *for some* i ∈ id, e ∈ exp *and*

(2) *There is no* j, $1 \le j < k$ *such that* π_j *is of the form* (**begin**, (Assign i e′)) *for some* e′ ∈ exp.

Definition 6.10: *(π-live)*

Let π *be a program point of the form* (**begin**, (Assign i e)) *for some* i ∈ id, e ∈ exp. *An occurrence of a program point* π_1 *in some possible execution trace of the analysis is said to be π-live iff for some statement* s

Case 1: π_1 *is of the form* (**begin**, s)

and in the corresponding evaluation of S_ud⟦s⟧x, x ∈ Def_Store, π ∈ x[i].

Case 2: π_1 *is of the form* (end, s)

and in the corresponding evaluation of S_ud⟦s⟧x, x ∈ Def_Store, $\pi \in$ S_ud⟦s⟧x↓1.

Lemma 6.3:

Let $\pi_1, \cdots, \pi_k$ *be a segment of some possible execution trace in the standard interpretation of a program* p. *If the sequence* $\pi_1, \cdots, \pi_k$ *is* π*-free for any* $\pi \in PP_p$*, then there is a segment* $\pi'_1, \cdots, \pi'_l$ *in every possible execution trace of the analysis interpretation such that*

(1) $\pi'_1 = \pi_1$ *and* $\pi'_l = \pi_k$.

(2) π'_1 *is* π*-live implies* π'_l *is* π*-live.*

Proof:

It can be easily verified that every possible analysis execution trace is a maximal chain of the corresponding control graph. Suppose there is an occurrence of π_1 before π_k in some possible standard execution trace. Since every possible standard execution trace is also a control chain of $CG_S(p)$, then from Lemma 6.2, there must be an occurrence of π_1 before π_k in every possible analysis execution trace for the analysis.

We will prove the second part of the lemma using structural induction on all possible standard execution traces in the evaluation of any statement.

The proof scheme for each statement consists of the following steps:

(1) For each statement, we will define a set of standard execution trace segments such that every segment in every possible standard execution trace is a concatenation of one or more segments from this set.

(2) We will show that the lemma is true for each segment in this set.

(3) We will show that corresponding to every possible concatenation of such segments, there is a corresponding concatenation of segments in every possible analysis execution trace such that the lemma is true.

Assignment Statement (Assign i e):

The segment (begin, (Assign i e)), (end, (Assign i e)) is the only segment possible for this statement. This segment is π-free for any π of the form (end, (Assign i' e) where $i' \neq i$. It can be easily verified by inspection of the semantic equation for the assignment statement, that the lemma is true for any such π.

Conditional statement (If b s1 s2):

Every segment in every possible standard execution trace for the conditional statement must be a concatenation of one or more of the following segments:

(1) The segment (begin, (If b s1 s2)), (begin, s1).

(2) The segment (begin, (If b s1 s2)), (begin, s2).

(3) The segment (end, s1), (end, (If b s1 s2)).

(4) The segment (end, s2), (end, (If b s1 s2)).

(5) All segments in every possible standard execution trace in the evaluation of s1 and s2.

From the structural induction hypothesis, the lemma is true for every segment in (5). It can be easily verified by inspection of the semantic equation that the lemma is true for segments (1) through (4). If σ_1 and σ_2 are any two segments in the above collection then there must be two corresponding segments σ'_1 and σ'_2 in every possible analysis execution trace such that the lemma is true. It is also easy to verify that if the concatenation of σ_1 and σ_2 is a segment in some possible standard execution trace then the concatenation of σ'_1 and σ'_2 is a segment in every possible analysis execution trace.

Composition statement (Comp s1 s2):

The proof is similar to the conditional statement using the following segments:

(1) The segment (begin, (Comp s1 s2)), (begin, s1).

(2) The segment (end, s1), (begin, s2).

(3) The segment (end, s2), (end, (Comp s1 s2)).

(4) All segments in every possible standard execution trace in the evaluation of s1 and s2.

Loop statement (While b s):

It can be easily verified that the lemma is true for each of the following segments:

(1) The segment (begin, (While b s)), (begin, s)

(2) The segment (end, s), (begin, (While b s))

(3) The segment (begin, (While b s)), (end, (While b s))

(4) All segments in every possible standard execution trace in the evaluation of s.

Every segment in a possible standard execution trace occurring within a single iteration of the loop (i.e. between two successive recursive calls) can be constructed using segments from the above list. It is easy to verify that for every such segment, there is a corresponding segment between every two successive recursive calls in every possible analysis execution trace such that the lemma is true.

As we mentioned at the beginning of the chapter, we can assume that in every evaluation of a fix argument expression, the termination due to the convergence of the argument (if it occurs) is delayed for at least one more recursive unfolding of the expression without affecting the semantics of the expression. Hence, every possible analysis execution trace for the loop statement must be of the form (begin, (While b s)) , (begin, s) , β' , (end, s) ,

(begin, (While b s)) , (begin, s) , β'' , (end, s) , $\cdots$

where β' and β'' are some possible analysis execution traces for the evaluation of s.

Every segment in every possible standard execution trace for the loop statement must be of the form

π_1 , α , (begin, (While b s)) , <(begin, s) , β , (begin, (While b s))>^{n} , (begin, s) , γ , π_2

where α, β and γ are finite sequences of program points that do not contain any occurrences of the program point (begin, (While b s)). <X>^{n} denotes n repetitions of the sequence X.

Case 1: n = 0

The segment π_1, $\cdots$, π_2 occurs within a single trace corresponding to a single iteration of the loop and hence there is a segment before the second occurrence of (begin, (While b s)) in every possible analysis execution trace for the loop such that the lemma is satisfied.

Case 2: n > 0

If the segment π_1, $\cdots$, π_2 is π-free for some π, then the two segments within it π_1 , α , (begin, (While b s)) and (begin, s) , γ , π_2 must be π-free. Since both these segments occur within a single iteration of the loop, there are corresponding segments π_1 , α' , $\bar{\pi}$ and $\bar{\pi}'$, γ' , π_2 in every possible analysis execution trace such that $\bar{\pi}$ is the second occurrence of (begin, (While b s)) and $\bar{\pi}'$ is the program point (begin, s) occurring immediately after $\bar{\pi}$. The concatenation of these two segments occurs in every possible analysis execution trace and satisfies the lemma for the segment π_1, $\cdots$, π_2 in every standard execution trace.

Hence, Lemma 6.3 follows by structural induction.

Lemma 6.4:

Let p *be any program. If there is standard interpretation such that a definition at* π_1 = (begin, (Assign i e)) *reaches a program point* π_2 *then there is an occurrence of* π_2 *in every possible analysis execution trace such that* π_2 *is* π_1*-live.*

Proof: Since the definition at π_1 reaches π_2, there is a segment π, $\cdots$, π_2 in some standard execution trace such that π = (end, (Assign i e)) and the segment is π_1-free. It is obvious that every occurrence of π is π_1-live in every possible analysis execution trace. Hence from Lemma 6.3, there is an occurrence of π_2 in every possible analysis execution traces and this occurrence is π_1-live.

Lemma 6.5:

$\forall$x $\in$ Def_Store. $\forall$e $\in$ exp. $\exists$i $\in$ Id. *if* e *uses* i *then*

$$x[i] \subseteq E\_ud[\![e]\!]x$$

Proof: Easily verified by inspection.

Lemma 6.6:

$\forall$x $\in$ Def_Store. $\forall$b $\in$ b_exp. $\exists$i $\in$ id. *if* b *uses* i *then*

$$x[i] \subseteq B\_ud[\![b]\!]x$$

Proof: Easily verified by inspection.

Lemma 6.7:

$\forall$x $\in$ Def_Store. $\forall$s $\in$ stmt. $\exists$i $\in$ id. *if* s *uses* i *then*

$$x[i] \subseteq (S\_ud[\![s]\!]x)\downarrow 2$$

Proof: Easily verified using Lemmas 6.5 and 6.6.

Lemma 6.8:

Let s *be any statement. For all* x $\in$ Def_Store. S_ud[[s]]x *terminates.*

Proof:

Proof is by structural induction. It is obvious for assignment, conditional and composition statements. In the proof of Lemma 6.3, we showed that every possible analysis execution trace for the loop statement must be of the form

(**begin**, (While b s)) , (**begin**, s) , β' , (**end**, s) ,
(**begin**, (While b s)) , (**begin**, s) , β'' , (**end**, s) , (**begin**, (While b s)) , $\cdots$

From the induction hypothesis, β' and β'' must be finite. Let $\bar{\pi}_i$ denote the ith occurrence of (**begin**, (While b s)) in some possible analysis execution trace. Consider all possible standard execution trace segments starting with a definition site inside s and ending at some π' = (**begin**, (While b s) such that the definition reaches π'.

If there are none, then all segments inside s are π-free for all $\pi \in PP_p$. This implies that $\bar{\pi}_2$ is π-live for exactly those $\pi \in PP_p$ for which $\bar{\pi}_1$ is π-live. From the definition of π-live, the argument to the second recursive call is the same as the argument to the first recursive call and from the semantics of the fix argument option, the evaluation terminates.

Let π = (**begin**, (Assign i e)) be a definition site inside s such that π reaches a program point π' = (**begin**, (While b s) in some standard interpretation. Then, there is a segment $\pi_1, \cdots, \pi'$ where π_1 = (**end**, (Assign i e)) in some possible standard execution trace and the segment is π-free. From Lemma 6.3, there is a segment in every possible analysis execution trace starting with π_1 and ending with π' such that π_1 is π-live implies π' is π-live. It is obvious that π_1 is always π-live in any possible analysis execution trace. Hence, the occurrence of π' is π-live.

Moreover, the segment $\pi_1, \cdots, \pi'$ occurs within some single iteration of the loop. In the proof of Lemma 6.3, we showed that the corresponding segment satisfying Lemma 6.3 for the segment $\pi_1, \cdots, \pi'$, occurs between every two successive recursive calls in every possible analysis execution trace. Therefore, $\bar{\pi}_i$ is π-live for all $i > 1$ in every possible analysis execution trace.

Hence, $\bar{\pi}_3$ is $\hat{\pi}$-live for exactly those $\hat{\pi}$ such that

(1) All possible standard execution traces for the evaluation of s are $\hat{\pi}$-free and $\bar{\pi}_2$ is $\hat{\pi}$-live or

(2) $\hat{\pi}$ is a definition inside s that reaches the end of s.

Since $\bar{\pi}_2$ is also $\hat{\pi}$-live for all $\hat{\pi}$ that satisfy (2), the argument to the third recursive call must be the same as the argument to the second recursive call. Hence, the evaluation of the expression terminates.

Theorem 6.1: *(Correctness of Use-Def Analysis)*

Let p *be any program.*

(1) P_ud⟦p⟧ *terminates.*

(2) *If in any execution trace corresponding to the evaluation of* P⟦p⟧, *a definition of a variable* i *at program point* π *reaches a statement* s *and* s *uses* i *then*

$$\pi \in (\text{P\_ud}[\![p]\!][(\textbf{begin}, s)])$$

Proof:

(1) Follows from Lemma 6.8.

(2) From Lemma 6.4,

$$\pi' = (\textbf{begin}, s) \text{ is } \pi\text{-live.}$$

From Lemma 6.7

$$\exists x \in \text{Def\_Store}.\ \pi \in (S[\![s]\!]x)\downarrow 2.$$

From the semantics of the collect option it follows that

$$\pi \in (\text{P\_ud}[\![p]\!][(\textbf{begin}, s)]).$$

6.2.2. Def-Use Analysis

Theorem 6.2: *(Correctness of Def-Use Analysis)*

Let p *be any program.*

(1) P_du⟦p⟧ *terminates.*

(2) *If in any execution trace corresponding to the evaluation of* P⟦p⟧, *a definition of a variable* i *at program point* π *is used by a statement* s,

$$(\textbf{begin}, s) \in (\text{P\_du}[\![p]\!][\pi])$$

Proof:

(1) Trivial using structural induction. There are no recursive functions.

(2) It can be easily shown that for every statement s ∈ stmt, there is an evaluation of S_du⟦s⟧x for some x ∈ I_Store. It is also easy to see that for all such evaluations x = P_ud⟦p⟧. Hence from theorem 6.4, if a definition at program point π reaches s and is used within it, then π ∈ x[(begin, s)].

Moreover, it can easily shown by induction that for any statement s, if the evaluation S_du⟦s⟧x = y for any x then for any program point π such that π ∈ x[(begin, s′)] where s′ is a statement within s

(begin, s′) ∈ y[π].

Since this is true for any statement, it is true for the statement that makes up the body of the program and hence the theorem follows.

6.2.3. Available expressions

The correctness of the specification for available expressions (Figure 4.10) is proved in a fashion very similar to the correctness of Use-Def analysis. In this section we will provide the formalism necessary to express the correctness and the correctness results without proof.

Lemma 6.9: *(Control Abstraction)*

For any program p ∈ prog, *the control graph for statements corresponding to the analysis* $CG_{rd}(p)$ *is an abstraction of the control graph corresponding to the standard semantics* $CG_s(p)$.

Definition 6.11: *(Expression-free)*

Let $\pi_1, \cdots, \pi_k$ *be any sequence of program points of a program* p. *The sequence is said to be* ξ*-free where* ξ ∈ (exp ∪ b_exp) *iff*

There is no j, 1 ≤ j ≤ k *such that* π_j *is of the form* (begin, ξ).

Definition 6.12: *(Expression-dead)*

Let ξ ∈ (exp ∪ b_exp) *be any expression. An occurrence of a program point* π_1 *in any possible analysis execution trace is said to be* ξ*-dead iff for any* s ∈ stmt, e ∈ exp, b ∈ b_exp

Case 1: π_1 *is of the form* (begin, s)

and in the corresponding evaluation of S_rd⟦s⟧x y, x ∈ Expr_Set, y ∈ I_Store, ξ ∈ x.

Case 2: π_1 *is of the form* (end, s)

and in the corresponding evaluation of S_rd⟦s⟧x y, x ∈ Expr_Set, y ∈ I_Store, ξ ∈ S_rd⟦s⟧x y.

Case 3: π_1 *is of the form* (begin, e)

and in the corresponding evaluation of E_rd⟦e⟧x, x ∈ Expr_Set, ξ ∈ x.

Case 4: π_1 *is of the form* **(end, e)**

and in the corresponding evaluation of E_rd[[e]]x, x ∈ Expr_Set, $\xi \in$ (E_rd[[e]]x)↓1).

Case 5: π_1 *is of the form* **(begin,** b)

and in the corresponding evaluation of B_rd[[b]]x, x ∈ Expr_Set, $\xi \in$ x.

Case 6: π_1 *is of the form* **(end,** b)

and in the corresponding evaluation of B_rd[[b]]x, x ∈ Expr_Set, $\xi \in$ (B_rd[[b]]x)↓1).

Lemma 6.10:

Let $\pi_1, \cdots, \pi_k$ *be some segment of some possible standard execution trace of a program* p. *If* $\pi_1, \cdots, \pi_k$ *is* ξ*-free for any* $\xi \in$ (exp ∪ b_exp), *then there is a segment* $\pi'_1, \cdots, \pi'_j$ *in every possible analysis execution trace such that*

(1) $\pi'_1 = \pi_1$ *and* $\pi'_j = \pi_k$

(2) π_1 *is* ξ*-dead implies* π_k *is* ξ*-dead.*

Lemma 6.11:

Let p *be any program. If there is a standard execution trace such that an expression* ξ *is unavailable at a program point* π *then there is an occurrence of* π *in every possible analysis execution trace such that* π *is* ξ*-dead.*

Lemma 6.12:

∀x ∈ Expr_Set. ∀e ∈ exp. ∃ξ ∈ (exp ∪ b_exp). *if* e *contains* ξ *then*

$\xi \notin$ (E_rd[[e]]x)↓1

Lemma 6.13:

∀x ∈ Expr_Set. ∀b ∈ b_exp. ∃ξ ∈ (exp ∪ b_exp). *if* b *contains* ξ *then*

$\xi \notin$ (B_rd[[b]]x)↓1

Lemma 6.14:

Let s *be any statement. For all* x ∈ Expr_Set. y ∈ I_Store. S_rd[[s]]x y *terminates.*

Theorem 6.3: *(Correctness of Available expressions Analysis)*

Let p *be any program.*

(1) P_rd[[p]] *terminates.*

(2) *If for any program point* π = **(begin,** ξ) *corresponding to some expression* $\xi \in$ (exp ∪ b_exp), P_rd[[p]][π] = "avail" *then in every possible execution trace corresponding to the evaluation of* P[[p]], *the expression* ξ *is available at* π.

6.3. Fixed point equivalence

In this section we prove that a common form of a fixed point expression using the argument termination option is equivalent to the fixed point expression without the use of the termination option under certain conditions. This allows the termination option to be ignored in fixed-point induction arguments used in the following sections.

Definition 6.13: *(Chain-bounded set)*

A partially ordered set D *is called chain-bounded iff all strictly increasing chains in* D *are finite.*

Lemma 6.15: *(Fixed point equivalence)*

Let $f_1 : D \rightarrow D_\perp$ *and* $f_2 : D \rightarrow D_\perp$ *be two functions denoted in the SPARE language by the expressions:*

$f_1 = \textbf{fix}\ f.\ \lambda x.\ f.g(x) + h(x)$

$f_2 = \textbf{fix argument}\ f.\ \lambda x.\ f.g(x) + h(x)$

where $g : D \rightarrow D_\perp$ *and* $h : D \rightarrow D_\perp$ *are continuous functions.*

If for all $x \in D.\ g(x) \geq x$ *and* D *is chain-bounded then*

$f_2(x)$ *terminates for all* $x \in D$ *and* $f_1 = f_2$.

Proof:

From the fixed point theorem,

$$f_1 = \text{fix } f = \bigsqcup_{i=0}^{\infty} (F^i(\perp))$$

where

$$F = \lambda f.\ \lambda x.\ f.g(x) + h(x)$$

For any $x_0 \in D$,

$$\begin{aligned} F^0(\perp)(x_0) &= \perp \\ F^1(\perp)(x_0) &= h(x_0) \\ F^2(\perp)(x_0) &= h(g(x_0)) + h(x_0) \\ &\cdots \\ F^i(\perp)(x_0) &= h(g^{i-1}(x_0)) + h(g^{i-2}(x_0)) + \cdots + h(x_0) \end{aligned}$$

It can be seen from above that

$\forall i>0.\ F^i(\perp)(x_0) = h(g^{i-1}(x_0)) + F^{i-1}(\perp)(x_0)$

Hence

$\forall i>0.\ \bigsqcup_{0}^{i} (F^i(\perp))(x_0) = F^i(\perp)(x_0)$

Since

$\forall x \in D.\ g(x) \geq x,$

it can be proved by induction that

$\forall i > 0.\ g^i(x) \geq g^{i-1}(x).$

Hence,

$x_0, g(x_0), g^2(x_0), \ldots, g^i(x_0)$

is a strictly increasing chain. Since every strictly increasing chain in D is finite (D is chain-bounded from the hypothesis to the lemma), there exists a finite n such that $g^n(x_0) = g^{n-1}(x_0)$.

Hence,

$$\begin{aligned} F^{n+1}(\perp)(x_0) &= h(g^n(x_0)) + h(g^{n-1}(x_0)) + \cdots + h(x_0) \\ &= h(g^{n-1}(x_0)) + h(g^{n-1}(x_0)) + \cdots + h(x_0) \\ &= h(g^{n-1}(x_0)) + h(g^{n-2}(x_0)) + \cdots + h(x_0) \\ &= F^n(\perp)(x_0) \end{aligned}$$

It can be easily shown that

$$\forall k > n.\ F^k(\perp)(x_0) = F^n(\perp)(x_0).$$

Hence,

$$f_1(x_0) = \bigsqcup_{i=0}^{\infty} (F^i(\perp))(x_0) = F^n(\perp)(x_0)$$

Now from the operational definition of the fixed point expression as a recursive function

$$f(x) = f(g(x)) + h(x)$$

we get

$$f_2(x_0) = f(x_0) = \lim_{i \to \infty} G_i(x_0)$$

where

$$\begin{aligned} G_1(x_0) &= f(g(x_0)) + h(x_0) \\ G_2(x_0) &= f(g^2(x_0)) + h(g(x_0)) + h(x_0) \\ &\cdots \\ G_i(x_0) &= f(g^i(x_0)) + h(g^{i-1}(x_0)) + \cdots + h(x_0) \end{aligned}$$

We have shown earlier that there exists a finite n such that

$$g^n(x_0) = g^{n-1}(x_0)$$

Consider,

$$G_{n-1}(x_0) = f(g^{n-1}(x_0)) + h(g^{n-2}(x_0)) + \cdots + h(x_0)$$

$$G_n(x_0) \quad = \quad f(g^n(x_0)) + h(g^{n-1}(x_0)) + h(g^{n-2}(x_0)) + \cdots + h(x_0)$$

It can be easily verified that

$$\forall k > n.\ G_k(x_0) = G_n(x_0)$$

Hence

$$f_2(x_0) = \lim_{i \to \infty} G_i(x_0) = G_n(x_0)$$

The arguments to the recursive call in $G_n(x_0)$ is the same as the argument to the recursive call in the previous evaluation. From the semantics of the argument termination option the recursive call in $G_n(x_0)$ is replaced by $\perp_D$. This implies that $f_2(x_0)$ terminates. Moreover,

$$\begin{aligned} f_2(x_0) &= \perp_D + h(g^{n-1}(x_0)) + h(g^{n-2}(x_0)) + \cdots + h(x_0) \\ &= h(g^{n-1}(x_0)) + h(g^{n-2}(x_0)) + \cdots + h(x_0) \\ &= f_1(x_0) \end{aligned}$$

Since this holds for any $x_0 \in D$, $f_2(x)$ terminates for all $x \in D$ and $f_1 = f_2$.

6.4. State abstraction algorithms

The information provided by conditional constant propagation and range propagation algorithms are abstractions of states that may occur during standard interpretation. The semantic functions in analysis specifications are similar to semantic functions in a standard semantics specification except that they are defined on abstract domains and use approximation mechanisms to ensure termination. The proofs for these specifications are provided by relating information computed by analysis (abstract) semantic functions to information computed by the corresponding standard semantic functions.

For our meta-discussion of properties of semantic functions, it is necessary to denote non-termination of semantic functions. When it is not clear from context we use $\perp\!\!\!\perp$ to denote non-termination of function evaluations to distinguish it from $\perp$ that may denote values in semantic domains. We ensure that semantic functions are total in the meta-discussion by assuming that all semantic domains are lifted, with $\perp\!\!\!\perp$ as the least element.

Finally, we use $F \vdash G$ to denote that an evaluation of the semantic function F, if it terminates, involves an evaluation of the semantic function G.

6.4.1. Conditional constant propagation

We first prove the correctness of the naive algorithm in Fig. 4.3 (Chapter 4) relative to the standard semantics in Fig. 4.2. The abstraction function that maps elements of Prog_Store to elements in Cstore is defined pointwise using an injection from the domain of lifted integers to the lattice of constant propagation, Con. Since every element in $\text{integer}_\perp$ is mapped to itself in Con, we will not use the abstraction function explicitly in the proofs in this section.

For all $\breve{x} \in \text{Cstore}$ and $x \in \text{Prog\_Store}$, $\breve{x} \geq x$ should be read as $\breve{x} \geq_{\text{Con}} \alpha(x)$ where α is the abstraction function.

Lemma 6.16:

Let $e \in \text{exp}$ *be any expression.* $\forall \breve{x} \in \text{Cstore}.\ \forall x \in \text{Prog\_Store}.$ *if* $\breve{x} \geq x$ *then*

$$\text{E\_ccp}\llbracket e \rrbracket \breve{x} \geq \text{E}\ \llbracket e \rrbracket x$$

Proof: Easily verified by inspection.

Lemma 6.17:

Let $b \in \text{b\_exp}$ *be any boolean expression.* $\forall \breve{x} \in \text{Cstore}.\ x \in \text{Prog\_Store}.$ *if* $\breve{x} \geq x$ *then*

$$\text{Bt\_ccp}\llbracket b \rrbracket \breve{x} = \perp \Rightarrow \text{B}\ \llbracket b \rrbracket x = \text{false}$$

Proof: Easily verified by inspection.

Lemma 6.18:

Let $b \in \text{b\_exp}$ *be any boolean expression.* $\forall \breve{x} \in \text{Cstore}.\ x \in \text{Prog\_Store}.$ *if* $\breve{x} \geq x$ *then*

$$\text{Bf\_ccp}\llbracket b \rrbracket \breve{x} = \perp \Rightarrow \text{B}\ \llbracket b \rrbracket x = \text{true}$$

Proof: Easily verified by inspection.

Lemma 6.19:

Let $s \in \text{stmt}$ *be any statement.*

$\forall \breve{x} \in \text{Cstore}.$ *if* $\text{S\_ccp}\llbracket s \rrbracket \breve{x}$ *terminates then*

$$\text{S\_ccp}\llbracket s \rrbracket \breve{x} \geq \breve{x}$$

Proof:

The proof is by structural induction. The lemma can be easily proved for assignment, conditional and composition statements. We prove the lemma for a loop statement through fixed-point induction using the induction hypothesis. We ignore the termination option for this lemma and show its equivalence later. The induction hypothesis is

$$P(f) = \forall \breve{x} \in \text{Cstore}.\ f(\breve{x}) = \perp \vee f(\breve{x}) \geq \breve{x}$$

For the base case, it is obvious that $P(\lambda x.\ \perp)$ holds. For the induction hypothesis assume that $P(f)$ holds. We need to show that $P(F(f))$ holds where

$$F = \lambda f.\lambda x.\ f(\text{S\_ccp}\llbracket \text{stmt} \rrbracket(\text{Bt\_ccp}\llbracket \text{b\_exp} \rrbracket x)) + \text{Bf\_ccp}\llbracket \text{b\_exp} \rrbracket x$$

For any $\breve{x} \in \text{Cstore}$

$$F(f)(\breve{x}) = f(\text{S\_ccp}\llbracket \text{stmt} \rrbracket(\text{Bt\_ccp}\llbracket \text{b\_exp} \rrbracket \breve{x})) + \text{Bf\_ccp}\llbracket \text{b\_exp} \rrbracket \breve{x}$$

If $f(S\_ccp[\![stmt]\!](Bt\_ccp[\![b\_exp]\!]\bar{x})) = \perp\!\!\!\perp$ then $F(f)(\bar{x}) = \perp\!\!\!\perp$. Suppose that

$$f(S\_ccp[\![stmt]\!](Bt\_ccp[\![b\_exp]\!]\bar{x})) \neq \perp\!\!\!\perp$$

Case 1: $Bf\_ccp[\![b\_exp]\!]\bar{x} = \perp$

It can be shown that $Bt\_ccp[\![b\_exp]\!]\bar{x} = \bar{x}$. From the structural induction hypothesis and the fixed-point induction hypothesis we have

$$F(f)(\bar{x}) = f(S\_ccp[\![stmt]\!](Bt\_ccp[\![b\_exp]\!]\bar{x})) \geq \bar{x}$$

Case 2: $Bf\_ccp[\![b\_exp]\!]\bar{x} \neq \perp$

It can be shown that $Bf\_ccp[\![b\_exp]\!]\bar{x} = \bar{x}$. Then

$$F(f)(\bar{x}) = f(S\_ccp[\![stmt]\!](Bt\_ccp[\![b\_exp]\!]\bar{x})) + \bar{x} \geq \bar{x}.$$

This completes the proof.

Lemma 6.20:

Let $s \in stmt$ *be any statement.*

$$\forall \bar{x} \in Cstore.\ S\_ccp[\![s]\!]\bar{x} \textit{ terminates.}$$

Proof:

Proof is by structural induction. It is trivial for assignment, conditional and composition statements. Consider

$$\textbf{fix argument } F.\ \lambda x.\ F(S\_ccp[\![stmt]\!](Bt\_ccp[\![b\_exp]\!]x)) + Bf\_ccp[\![b\_exp]\!]$$

If at any time during the evaluation of this expression, $Bt\_ccp[\![b\_exp]\!]x$ evaluates to $\perp$ then the expression terminates since all the functions are declared to be strict. Suppose that $Bt\_ccp[\![b\_exp]\!]x$ always evaluates to x during the evaluation. Then the above expression is of the form

$$\textbf{fix argument } F.\ \lambda x.f(g(x)) + h(x)$$

where

$$g(x) = S\_ccp[\![stmt]\!](Bt\_ccp[\![b\_exp]\!]x)$$

From the structural induction hypothesis g(x) terminates for all x. Hence from Lemma 6.19, $g(x) \geq x$ for all x. Moreover, for any program p, the domain Cstore is chain-bounded. Therefore, from Lemma 6.15, the fixed-point expression always terminates. This completes the proof.

Lemma 6.21:

Let $s \in stmt$ *be any statement.* $\forall \bar{x} \in Cstore.\ \forall x \in Prog\_Store.$ *if* $\bar{x} \geq x$ *then*

$$S\_ccp[\![s]\!]\bar{x} \geq S[\![s]\!]x$$

Proof:

If $S\llbracket s\rrbracket x = \perp$, then the Lemma is trivially true. Suppose that $S\llbracket s\rrbracket x \neq \perp$. We use structural induction on elements of stmt. The proof follows trivially from Lemmas 6.16, 6.17 and 6.18 for assignment, conditional and composition. We will provide the proof for the loop statement using simultaneous fixed-point induction on the two functionals. We will ignore the termination option in the fixed-point induction and use the Lemma 6.15 to show the equivalence later.

We use the induction predicate

$$P(f, \bar{f}) = \forall \tilde{x} \in \text{Cstore}.\ \forall x \in \text{Prog\_store}.\ \bar{f}(\tilde{x}) \geq f(x)$$

For the basis step, it is obvious that $P(\lambda x.\ \perp, \lambda x.\ \perp)$ holds. For the induction step we use the induction hypothesis $P(f, \bar{f})$. We need to show that $P(F(f), \bar{F}(\bar{f}))$ holds, where:

$F = \lambda f.\lambda x.\ B\llbracket b\_exp\rrbracket x \rightarrow f(S\llbracket stmt\rrbracket x),\ x$

$\bar{F} = \lambda f.\lambda x.\ f(S\_ccp\llbracket stmt\rrbracket(Bt\_ccp\llbracket b\_exp\rrbracket x)) + Bf\_ccp\llbracket b\_exp\rrbracket x$

For any $x_0 \in \text{Prog\_Store}$ and $\tilde{x}_0 \in \text{Cstore}$ such that $\tilde{x}_0 \geq x_0$,

$F(f)(x_0) = B\llbracket b\_exp\rrbracket x_0 \rightarrow f(S\llbracket stmt\rrbracket x_0),\ x_0$

$\bar{F}(\bar{f})(\tilde{x}_0) = f(S\_ccp\llbracket stmt\rrbracket(Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0)) + Bf\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0$

Case 1: $Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0 = \perp$.
From Lemma 6.17 $B\llbracket b\_exp\rrbracket x_0 = \text{false}$. Also it can be shown that $Bf\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0 = \tilde{x}_0$. Then

$$\bar{F}(\bar{f})(\tilde{x}_0) = \tilde{x}_0 \geq x_0 = F(f)(x_0)$$

Case 2: $Bf\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0 = \perp$
From Lemma 6.18 $B\llbracket b\_exp\rrbracket x_0 = \text{true}$ and $Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0 = \tilde{x}_0$. By the structural induction hypothesis

$$S\_ccp\llbracket stmt\rrbracket(Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0) \geq S\llbracket stmt\rrbracket x_0.$$

By the fixed-point induction hypothesis it follows that

$$\bar{F}(\bar{f})(\tilde{x}_0) = \bar{f}(S\_ccp\llbracket stmt\rrbracket(Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0)) \geq f(S\llbracket stmt\rrbracket x_0) = F(f)(x_0)$$

Case 3: $Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0 \neq \perp$ and $Bf\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0 = \perp$
It can be shown that $Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0 = \tilde{x}_0$. As in case 2, $\bar{f}(S\_ccp\llbracket stmt\rrbracket(Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0)) \geq f(S\llbracket stmt\rrbracket x_0)$.
Hence,

$$\bar{F}(\bar{f})(\tilde{x}_0) = \bar{f}(S\_ccp\llbracket stmt\rrbracket(Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x}_0)) + \tilde{x}_0 \geq f(S\llbracket stmt\rrbracket x_0) + x_0 \geq F(f)(x_0)$$

This completes the proof by fixed-point induction. To show the correctness of the fixed-point expression with the termination option, note that if at any time during the evaluation of the expression $Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x}$ becomes $\perp$ the evaluation terminates and the termination option has no effect. Suppose an evaluation terminates due to the termination option. Then since in each recursive call $Bt\_ccp\llbracket b\_exp\rrbracket \tilde{x} = \tilde{x}$, by the structural induction hypothesis, we have $g(\tilde{x}) = S\_ccp\llbracket stmt\rrbracket \tilde{x} \geq \tilde{x}$. Hence, by Lemma 6.15 this evaluation is equivalent to the evaluation of the fixed-point expression without the termination option.

$\forall i \in \mathrm{id}.\ \mathrm{P\_ccp}\llbracket p \rrbracket[i] = n \wedge n \in \mathrm{integer}$ *then* i *is a constant in all evaluations of* $\mathrm{P}\llbracket p \rrbracket$.

Proof:

(1) Follows from Lemma 6.20.

(2) Let p be any program p = (Prog s) where $s \in \mathrm{stmt}$. Then

$\mathrm{P\_ccp}\llbracket p \rrbracket = \mathrm{S\_ccp}\llbracket s \rrbracket(\perp_{\mathrm{Cstore}}[\,])$ and

$\mathrm{P}\llbracket p \rrbracket = \mathrm{S}\llbracket s \rrbracket(\perp_{\mathrm{Prog\_Store}}[\,])$

Since

$$\perp_{\mathrm{Cstore}}[\,] = \perp_{\mathrm{Prog\_Store}}[\,]$$

from Lemma 6.19 we have,

$$\mathrm{S\_ccp}\llbracket s \rrbracket(\perp_{\mathrm{Cstore}}[\,]) \geq \mathrm{S}\llbracket s \rrbracket(\perp_{\mathrm{Prog\_Store}}[\,])$$

Hence

$$\mathrm{P\_ccp}\llbracket p \rrbracket = \perp \Rightarrow \mathrm{P}\llbracket p \rrbracket = \perp\!\!\perp \vee \mathrm{P}\llbracket p \rrbracket = \perp$$

(3) Using Lemma 6.20 and an argument similar to the one above we can show that

If $\mathrm{P}\llbracket p \rrbracket \neq \perp\!\!\perp$ and $\exists i \in \mathrm{id}.\ \exists e \in \mathrm{exp}.\ \mathrm{P}\llbracket p \rrbracket \vdash \mathrm{S}\llbracket \mathrm{Assign}\ i\ e \rrbracket x$ then

$$\mathrm{P\_ccp}\llbracket p \rrbracket[i] \geq \mathrm{E}\llbracket e \rrbracket x.$$

If an identifier i is assigned to two integer values n_1 and n_2 in any evaluation of $\mathrm{P}\llbracket p \rrbracket$, then

$\mathrm{P\_ccp}\llbracket p \rrbracket[i] \geq n_1$ and $\mathrm{P\_ccp}\llbracket p \rrbracket[i] \geq n_2$.

For the above conditions to be simultaneously satisfied, we must have

$\mathrm{P\_ccp}\llbracket p \rrbracket[i] = \top$.

Since $\top \notin \mathrm{integer}$, the theorem follows.

Modified conditional constant propagation

Lemma 6.23: *(Collection lemma)*

$\forall p \in \mathrm{prog}.\ \exists i \in \mathrm{id}.\ \exists e \in \mathrm{exp}.\ \exists \hat{x} \in \mathrm{Cstore}.$

$(\mathrm{P\_ccp}\llbracket p \rrbracket \vdash \mathrm{S\_ccp}\llbracket \mathrm{Assign}\ i\ e \rrbracket \hat{x} \wedge \hat{x} \neq \perp) \Rightarrow \mathrm{P\_ccp}\llbracket p \rrbracket[i] \geq \mathrm{E\_ccp}\llbracket e \rrbracket \hat{x}$

Proof follows from the semantics of the collect option.
It can also be easily verified that Lemmas 6.16, 6.17, 6.18 and 6.20 hold for the modified specification.

Lemma 6.24:

Let $s \in \mathrm{stmt}$ *be any statement. If for any* $x \in \mathrm{Prog\_Store}$, $\mathrm{S}\llbracket s \rrbracket x$ *terminates then*

$\exists s' \in \mathrm{stmt}.\ \exists x' \in \mathrm{Prog\_Store}.\ (\mathrm{S}\llbracket s \rrbracket x \vdash \mathrm{S}\llbracket s' \rrbracket x') \Rightarrow$

$$\forall \hat{x} \in \mathrm{Cstore}.\ \exists \hat{x}' \in \mathrm{Cstore}.\ \hat{x} \geq x \Rightarrow \mathrm{S\_ccp}\llbracket s \rrbracket \hat{x} \vdash \mathrm{S\_ccp}\llbracket s' \rrbracket \hat{x}' \wedge \hat{x}' \geq x'$$

Proof: We prove this by structural simplification.

Assignment statement:

Trivially true since, the evaluation of an assignment statement does not involve an evaluation of any other statement.

Conditional statement:

Let $x \in \text{Prog\_Store}$ and $\bar{x} \in \text{Cstore}$. Consider the evaluation of $S[\![\text{If b s2 s3}]\!]x$ and $S\_ccp[\![\text{If b s2 s3}]\!]\bar{x}$ for $b \in \text{b\_exp}$, $s2, s3 \in \text{stmt}$.

Case 1: $Bt\_ccp[\![b]\!]\bar{x} = \perp$.

We can show that $Bf\_ccp[\![b]\!]\bar{x} = \bar{x}$. From Lemma 6.17 $B[\![b]\!]x = \text{false}$. This implies that there is no evaluation of $S[\![s2]\!]x'$ for any $x' \in \text{Prog\_Store}$. However, it is obvious that the lemma holds for the evaluations of $S[\![s3]\!]x$ and $S\_ccp[\![s3]\!](Bf\_ccp[\![b]\!]\bar{x})$.

Case 2: $Bf\_ccp[\![b]\!]\bar{x} = \perp$.

Similar to Case 1 with evaluations for s2.

Case 3: $Bt\_ccp[\![b]\!]\bar{x} = \bar{x}$ and $Bf\_ccp[\![b]\!]\bar{x} = \bar{x}$.

Since both branches in the analysis are always evaluated, Case 1 and Case 2 imply that regardless of which branch is evaluated in the standard interpretation, the lemma is satisfied.

Composition statement:

Let $x \in \text{Prog\_Store}$ and $\bar{x} \in \text{Cstore}$. Consider the evaluation of $S[\![\text{Comp s2 s3}]\!]x$ and $S\_ccp[\![\text{Comp s2 s3}]\!]\bar{x}$ for $s2, s3 \in \text{stmt}$.

It is obvious that the evaluations of $S[\![s2]\!]x$ and $S\_ccp[\![s2]\!]\bar{x}$ satisfy the Lemma. From Lemma 6.20, $S\_ccp[\![s2]\!]\bar{x} \geq S[\![s2]\!]x$ when $\bar{x} \geq x$. Hence the Lemma is satisfied by the evaluations of $S[\![s1]\!](S[\![s2]\!]x)$ and $S\_ccp[\![s1]\!](S\_ccp[\![s2]\!]\bar{x})$.

While statement:

Let f and $\bar{f}$ denote the recursive functions, $S[\![\text{While b s}]\!]$ and $S\_ccp[\![\text{While b s}]\!]$ respectively. Let f_i and $\bar{f}_i$ denote the ith recursive call in the evaluation of f and $\bar{f}$ respectively. Let $\mathbf{S}$, $\mathbf{B}$, $\bar{\mathbf{S}}$, $\mathbf{B}_t$ and $\mathbf{B}_f$ denote $S[\![s]\!]$, $B[\![b]\!]$, $S\_ccp[\![s]\!]$, $Bt\_ccp\ [\![\ b\ ]\!]$ and $Bf\_ccp[\![b]\!]$ respectively. Ignoring the termination option, the ith and i+1th calls are related as follows:

$$f_i(x) = \mathbf{B}(x) \rightarrow f_{i+1}(\mathbf{S}(x)),\ x$$

$$\bar{f}_i(\bar{x}) = \bar{f}_{i+1}(\bar{\mathbf{S}}(\mathbf{B}_t(\bar{x})) + \bar{x}) + \mathbf{B}_f(\bar{x})$$

We will show by induction on i that if there is a recursive call $f_i(x_i)$ in the evaluation of $f(x)$ then for all $\bar{x} \geq x$, there is a recursive call $\bar{f}_i(\bar{x}_i)$ in the evaluation of $\bar{f}(\bar{x})$ such that $\bar{x}_i \geq x_i$. In other words, we will use the predicate

$$P(i) = \forall x \in \text{Prog\_Store}.\ f(x) \vdash f_i(x_i) \Rightarrow \forall \bar{x} \geq x.\ \exists \bar{x}_i.\ \bar{f}(\bar{x}) \vdash \bar{f}_i(\bar{x}_i) \wedge \bar{x}_i \geq x_i$$

For the base case, the hypothesis $\bar{x} \geq x$ establishes the predicate for $i=0$. Assume that $P(i)$ is true. We must show that $P(i+1)$ holds.

It is clear from above that there is an evaluation of $f_{i+1}(x_{i+1})$ iff $\mathbf{B}(x_i)$ is true. If $\mathbf{B}(x_i)$ is false, then the predicate trivially holds. Suppose that $\mathbf{B}(x_i)$ is true, then from Lemma 6.17, we can show that for all $\bar{x}_i \geq x_i$,

$\mathbf{B}_t(\bar{x}_i) = \bar{x}_i$.

The arguments to the i+1th recursive call are then given by

$x_{i+1} = \mathbf{S}(x_i)$ and $\bar{x}_{i+1} = \bar{\mathbf{S}}(\bar{x}_i) + \bar{x}_i$

From Lemma 6.21, we have $\bar{x}_{i+1} \geq x_{i+1}$. Hence, the predicate holds for i+1. From induction, the predicate holds for all $i \geq 0$.

Since $\bar{\mathbf{S}}(\bar{x}_i) + \bar{x}_i \geq \bar{x}_i$ and Cstore is chain-bounded, for each $\bar{x}' \in$ Cstore, there is a finite n such that the argument to the nth recursive call in the evaluation of $\bar{f}(\bar{x}')$ is the same as the argument to the n+1th recursive call (and consequently for all calls after it). The use of the termination option avoids further recursive calls after the nth call. Since $\bar{x}'_k = \bar{x}'_n$ for all $k \geq n$, from the above induction result we conclude that:

If there is a recursive call $f_i(x_i)$ in the evaluation of $f(x)$ then for all $\bar{x} \geq x$, there is a recursive call $\bar{f}_j(\bar{x}_j)$ in the evaluation of $\bar{f}(\bar{x})$ such that $\bar{x}_j \geq x_i$.

Further, in the ith recursive call of f there is an evaluation of $S[\![s]\!]x_i$ iff $B[\![b]\!]x_i$ is true. Since $\bar{x}_j \geq x_i$, using Lemma 6.17 it is clear that there is an evaluation of $S\_ccp[\![s]\!]\bar{x}_j$. This proves the lemma for the while statement.

The proof of the lemma for any statement then follows by successive application of the simplification argument.

Theorem 6.5: *(Correctness of modified conditional constant propagation)*

For all programs $p \in$ prog

(1) $P\_ccp[\![p]\!]$ *terminates.*

(2) *If* $P[\![p]\!] \neq \perp\!\!\!\perp$ *and*

$\forall i \in id.\ P\_ccp[\![p]\!][i] = n \wedge n \in$ integer *then* i *is a constant in all evaluations of* $P[\![p]\!]$.

Proof:

(1) Follows from Lemma 6.20.

(2) Follows from Lemma 6.24 and the collection lemma.

6.4.2. Conditional range propagation

The correctness proofs for the range propagation is established in a fashion very similar to that of constant propagation. Hence, we state without proof the results required for the correctness and the correctness theorem.

Definition 6.14: *(Abstraction function)*

The abstraction function α : Prog_Store $\rightarrow$ Store *is defined as the function*

$$\alpha(s) = \lambda i.\ \textit{if}\ s[i] = \perp\ \textit{then}\ \perp\ \textit{else}\ (s[i], s[i])$$

Lemma 6.25:

Let $e \in$ exp *be any expression.* $\forall \breve{x} \in$ Store. $\forall x \in$ Prog_Store. *if* $\breve{x} \geq \alpha(x)$ *then*

$$\text{E\_crp}[\![e]\!]\breve{x} \geq \alpha(\text{E}\ [\![e]\!]x)$$

Lemma 6.26:

Let $b \in$ b_exp *be any boolean expression.*

$\forall x \in$ Prog_Store. *if* B $[\![b]\!]x$ = true *then*

$$\forall \breve{x} \in \text{Store}.\ \breve{x} \geq \alpha(x) \Rightarrow \text{Bt\_crp}[\![b]\!]\breve{x} \geq \alpha(x)$$

Lemma 6.27:

Let $b \in$ b_exp *be any boolean expression.*

$\forall x \in$ Prog_Store. *if* B $[\![b]\!]x$ = false *then*

$$\forall \breve{x} \in \text{Store}.\ \breve{x} \geq \alpha(x) \Rightarrow \text{Bf\_crp}[\![b]\!]\breve{x} \geq \alpha(x)$$

Lemma 6.28:

Let $s \in$ stmt *be any statement.*

$$\forall \breve{x} \in \text{Store}.\ \text{S\_crp}[\![s]\!]\breve{x}\ \textit{terminates}.$$

Lemma 6.29:

Let $s \in$ stmt *be any statement.* $\forall \breve{x} \in$ Store.$\forall x \in$ Prog_Store. *if* $\breve{x} \geq \alpha(x)$ *then*

$$\text{S\_crp}[\![s]\!]\breve{x} \geq \alpha(\text{S}\ [\![s]\!]x)$$

Lemma 6.30: *(Collection lemma)*

$\forall p \in$ prog. $\exists i \in$ id. $\exists e \in$ exp. $\breve{x} \in$ Store.

$$(\text{P\_crp}[\![p]\!] \vdash \text{S\_crp}[\![\text{Assign } i\ e]\!]\breve{x} \wedge \breve{x} \neq \perp) \Rightarrow \text{P\_crp}[\![p]\!][i] \geq \text{E\_crp}[\![e]\!]\breve{x}$$

Lemma 6.31:

Let $s \in \text{stmt}$ *be any statement. If for any* $x \in \text{Prog\_Store}$, $S[\![s]\!]x$ *terminates then*

$$\exists s' \in \text{stmt}.\ \exists x' \in \text{Prog\_Store}.\ (S[\![s]\!]x \vdash S[\![s']\!]x') \Rightarrow$$
$$\forall \hat{x} \in \text{Store}.\ \exists \hat{x}' \in \text{Store}.\ \hat{x} \geq \alpha(x) \Rightarrow S\_crp[\![s]\!]\hat{x} \vdash S\_crp[\![s']\!]\hat{x}' \wedge \hat{x}' \geq \alpha(x')$$

Theorem 6.6: *(Correctness of conditional range propagation)*

For all programs $p \in \text{prog}$

(1) $P\_crp[\![p]\!]$ *terminates.*

(2) *If* $P[\![p]\!] \neq \perp\!\!\!\perp$ *and*

$$\exists i \in \text{id}.\ \exists e \in \text{exp}.\ \exists x \in \text{Prog\_Store}.\ P[\![p]\!] \vdash S[\![\text{Assign } i\ e]\!]x \text{ then}$$
$$P\_crp[\![p]\!][i] \geq \alpha(E[\![e]\!]x).$$

6.5. Summary

The high-level nature of the specification enables the development of proofs without the need to consider irrelevant implementation details. The denotational framework allows proofs to be constructed in a modular fashion as a set of independent proofs for each production (i.e., language construct).

Specifications for related algorithms are often very similar and differ mainly in the domains used in the specifications. Proofs for such algorithms can be constructed from common proof schemes avoiding duplication of effort.

The proofs provided in this chapter for flow analysis specifications consist of two steps:

(1) Verification of the consistency of the control flow established by the analysis with the control flow established by the standard semantics.

(2) Verification of the consistency of the data flow in the analysis with the desired property of the standard semantics.

The goal of such a separation is to allow the first step to be performed without considering the nature of the particular flow analysis problem. In our example specifications, the verification of control abstraction required only the call relationships among semantic functions. We believe that when analysis algorithms are expressed as abstract interpretations, such a verification can be automated in certain cases. We will discuss this possibility in Chapter 7 where we suggest some extensions to this work.

Chapter 7
Conclusions

This chapter consists of four sections. The first section places this thesis in perspective with related studies. The second section describes limitations of denotational frameworks. Some issues for further research in this area are briefly described in the third section, and we summarize the thesis and draw conclusions from the work in the last section.

7.1. Related work

Although we are not aware of any published research with similar motivation as ours, there are several studies that relate to this work. These studies basically fall into three categories:

(1) Studies that use denotational semantics notation for expression of specific program analysis techniques.

(2) Studies that explore denotational approaches to program analysis specification.

(3) Studies that facilitate implementation of denotational specifications.

We will discuss a few specific examples in each category.

7.1.1. Expression of a particular technique

The studies in this category are generally concerned with expression of a technique rather than implementations for that technique. Hence, the specifications in these studies often use informal notation and conventions that are difficult to translate into implementations. A typical example is the use of ellipses ($\cdots$) which can be tricky to formalize. Such notation and informal conventions prevent these studies from translating automatically into applications of SPARE.

Hudak and Young [24]:

This study uses a denotational framework to specify a collecting interpretation for functional languages. This construction was motivated by a desire to avoid the use of power domains in analysis of programs in higher-order languages. The collecting interpretation takes as a parameter a semantics (standard or alternative) for the language. Every semantic equation in the collecting interpretation specification is of the form

$$\bar{\mathbf{M}}[\![\mathsf{l.e}]\!] \cdots = \mathsf{update}(\mathsf{c}, \mathsf{l}, \mathsf{f}(\mathbf{M}[\![\mathsf{l.e}]\!] \cdots, \cdots))$$

where

— $\bar{\mathbf{M}}$ is the collecting semantic function for expressions,

— **M** is the semantic function in the semantic specification provided as a parameter to the collecting interpretation,

— l.e is an expression in the target language with label l,

— c is a "cache" that maps expression labels to sets of denotable values in the parametric semantics,

— f is a function of the value (i.e., "meaning") of the labeled expression computed using the parametric semantic function.

— update is a function to update the cache, defined as

$$\text{update} : \text{cache} \rightarrow \text{label} \rightarrow D \rightarrow \text{cache} = \lambda c.\, \lambda l.\, \lambda d.\; c + \bot[\{d\}/l]$$

where D is the domain of denotable values in the parametric specification.

The collecting interpretation returns a "cache" that maps expression labels to results of computations obtained using the parametric semantic functions. The environment in which **M** is evaluated does not depend on the collecting interpretation at all. In other words, the semantics defined by the parametric specification is independent of the collecting mechanism.

On the other hand, the collecting interpretation uses the values computed by the parametric semantic functions. However, any approximation made in the collecting interpretation involves approximations to the sets of values computed by the parametric semantic functions and not to the domain of denotable values (D) in the parametric specification. This implies that semantic functions in the collecting interpretation do not have to be monotonic in D to guarantee a fixed-point. Consequently, the ordering in D is irrelevant to fixed-point computations in the collecting interpretation. Thus the collecting interpretation framework in [24] avoids the need for powerdomains, which are used to capture the ordering of base domains in powerset domains.

The collecting interpretation specification is essentially equivalent to a SPARE specification of the parametric semantics specified using the collect option along with the powerset option. The collecting interpretation in [24] is, in fact, an operational definition of such a collect option. From a pragmatic point of view, this framework does not provide mechanisms to guarantee termination of analysis specifications (such as strictness analysis provided as an example in this study). As we pointed out in Section 4.2.3, specification of a termination mechanism in a compositional framework when recursive procedures (or functions in this case) are involved may be difficult.

Cartwright and Felleisen [9]:

This study derives semantic definitions for program dependence graphs from generalizations of denotational semantics of the programming language. A denotational semantics specification (for a small imperative language) is transformed to arrive at a denotational definition (called the *demand semantics*) that exposes the data and control dependences between statements in the program. The *demand semantics* is then subjected to a staging analysis [28]. A staging analysis decomposes a denotational definition into two definitions — a semantic definition that constructs an intermediate representation for a program, and a semantic definition that provides the meaning of the intermediate representation. Intuitively, the first definition corresponds to a *compiler* that translates a program to an intermediate representation and the second definition corresponds to an *interpreter* for the intermediate representation. A staging analysis on the *demand semantics* produces a semantic definition for a *compiler* that constructs dependence trees. Dependence trees can be collapsed into program dependence graphs.

This study relates to our thesis in two ways:

(1) As a potential application for SPARE, it suggests the possibility of obtaining implementations that derive dependence trees for programs. However, the different formulations for the *update* operator (lazy and lackadaisical evaluations) used in [9] cannot be expressed directly in SPARE. A possibility is to extend the store domain constructor to allow for these different formulations and provide separate specifications for each such formulation.

(2) The staging analysis on the *demand semantics* produces a specification for standard interpretation of a dependence tree (and thus indirectly an interpretation of an equivalent program dependence graph). By replacing this interpretation with a non-standard interpretation, it may be possible to specify analysis algorithms that operate on program dependence graphs as opposed to control flow graphs used by SPARE specifications.

7.1.2. Denotational frameworks

The studies in this category demonstrate the use of denotational frameworks for the specification of analysis algorithms. These studies use the facilities generally used in denotational semantics of programming languages and suffer from the following deficiencies described in Chapter 2:

- The specification of the collection mechanism tends to obscure the analysis specification.
- The problem of formal specification of lattices used by analysis algorithms is not addressed.
- Correctness is usually established with respect to a standard collecting interpretation. However, non-standard formulations must be used to verify the correctness of analyses (e.g., flow analyses) that are not simple abstractions of a standard interpretation.

Nielson [41-43]:

In [41], guidelines are provided for the use of a denotational framework for specification of analysis algorithms for imperative programs. The concept of "occurrences" is used to associate information with program points. Both direct and continuation style semantics are used. The need to provide an operational specification of the collection as well as use of non-standard notation results in complex specifications.

Later studies by Nielson [42,43] have gravitated towards the notion of providing alternative interpretations for a single denotation of the language to derive different analysis techniques. In other words, abstract interpretation of a denotational definition of a program rather than the denotational definition of an abstract interpretation of a program. Although a considerable theoretical foundation has been provided for such an approach, development of practical analysis algorithms requires consideration of precision, efficiency, etc., and the issues involved in accommodating such considerations in the theoretical foundation have not been addressed.

Donzeau-Gouge [14]:

This study uses the language DSL supported by the SIS system [37] rather than the traditional notation of denotational semantics. As a result, the algorithms specified in [14] have been tested using SIS. Formal verification is also provided for the example specifications. The notion of "occurrence" is used to label program points and the collection is defined operationally. Continuation semantics is used in all specifications. The specifications for constant propagation and available expressions in this thesis can be contrasted with the corresponding specifications in [14] to realize the benefits of our framework not only in the clarity of the specification but in reducing the complexity of the formal verification as well.

7.1.3. Implementation of denotational specifications

The studies in this category deal with the issues involved in generating implementations from denotational semantics of programming languages. Knowledge gained from these studies can be used to provide efficient implementations for the SPARE language if it is to be used in other applications (e.g. compiler generators). Extensions proposed to the specification language in these studies are based on providing efficient implementations rather than ease of expression.

Mosses (SIS) [37]:

SIS was the first compiler generation system based on denotational semantics. The specifications are written in a language called DSL, a variant of the traditional notation used in denotational semantics. The evaluation is through lambda expression reduction using normal order semantics. Although this system is capable of executing virtually any denotational specification, the generated compilers are very large and slow.

Bahlke and Snelting (PSG) [6]:

The PSG system also uses a lambda expression reduction mechanism with normal order semantics and consequently also suffers from efficiency problems. However, PSG was designed to provide an interactive environment and allows incremental evaluation in cases where the derivation tree for a program is constructed top-down through successive refinements. Incompletely specified programs can be executed until an unexpanded construct is encountered. Execution is resumed as soon as the construct is expanded.

Paulson [45]:

The compiler generator in this study is similar to SIS except that certain efficiency considerations result in a more efficient system. The update operation is provided as an atomic operation. This feature makes it possible to treat stores and environments as association lists or even, in some cases, random access arrays. The major departure from SIS is in the separation of the evaluation into static ("compile time") and dynamic ("run time") components. Using a graph representation of λ-expressions, several combinator reductions and β-reductions are performed during the static evaluation.

Appel [5]:

Several studies [51, 62] provide combinators in the specification language to reflect more closely the operational capabilities of a von Neumann machine. The use of these combinators enables generation of code that is comparable in efficiency to code generated by a conventional compiler. Most of the combinators are directed towards reducing the inefficiencies involved in handling of stores and environments. However, the use of such combinators results in a lack of generality as compared to systems such as SIS. Appel's work [5] restores some generality through extensive compile-time evaluation as well as through introduction of new combinators that are natural to both von Neumann machines as well as to semantics of algorithmic languages.

7.2. A limitation of denotational frameworks

The compositional nature of denotational specifications places some constraints on the underlying representation of programs. In this thesis all specifications were provided as interpretations over control flow graphs. This limitation implies that analysis algorithms that use program structures such as static single assignment graphs or program dependence graphs, are not directly expressible in denotational frameworks. Further research is required to allow such analysis algorithms to be specified either indirectly or non-compositionally[35].

In Section 4.2.3, we noted the difficulty in expressing termination criteria when inter-procedural flow was involved. This is a natural consequence of the fact that termination of inter-procedural analyses cannot be expressed compositionally.

7.3. Further research

Possible issues for future research in this area fall into 3 categories:

(1) Extensions to the meta-language.

(2) Verification schemes for analysis specifications.

(3) Evaluation schemes for analysis specifications.

7.3.1. Language extensions

The most obvious extension is to allow continuations to handle analysis of programs written in languages with unstructured branching statements. Another extension is to provide an alternative semantics to the SPARE language incorporating lazy evaluation of semantic functions. This alternative interpretation can be useful in specification of analyses (such as [9]) that require lazy evaluation.

The standard semantics specification of the target language in conjunction with the formal semantics of SPARE provides the minimum constraints that a standard implementation for the target language must satisfy for the information derived from the analysis to be meaningful. These constraints may exclude standard interpretations that perform certain optimizing code transformations. The information obtained by a program analysis specification may no longer be valid if the transformations are not reflected in the program analysis. This is essentially the same problem noted by Hennessy [19] in debugging optimized programs. If one could provide transformation rules for the specification language that correspond to the transformation rules in the source language, a specification for an optimized program could be automatically obtained from the specification for an un-optimized program. The transformation scheme is pictured in Fig. 7.1.

[35]The collecting interpretation of [24] is an example of a specification that is denotational but not compositional.

$$\begin{tikzcd}
\text{Source Program} \arrow[r, "\text{Program Transformation}"] \arrow[d] & \text{Optimized Program} \arrow[d] \\
\text{Analysis specification} \arrow[r, "\text{Specification transformation}"'] & \text{A ? B} \\
& \text{(analysis specification for optimized program)}
\end{tikzcd}$$

Figure 7.1. Analysis of programs under optimization transformations

In a standard interpretation, it is necessary to ensure that the specification A is "equivalent" to B. In program analysis specification however, it is sufficient to show that A provides a "safer" approximation than B. It would be useful to identify source transformations that make such a derivation possible.

7.3.2. Verification schemes

One of the major goals of standardizing the expression of analysis algorithms is the potential for developing proof schemes that cover a number of related analysis algorithms. This avoids duplication of effort in developing correctness proofs for each of the algorithm specifications. By separating common aspects of a class of analysis algorithms, it may be possible to either provide common proofs or automate the derivation of such proofs.

The correctness proofs for flow analysis algorithms in Chapter 6 used the abstraction relation between the control graphs defined by the standard semantics and the analysis specification. If analysis algorithms are expressed as abstract interpretations (which implies that corresponding to every semantic function in the standard semantics there is an abstract semantic function in the analysis), the control dependences between program structures can essentially be determined by call relationships between semantic functions.

If the semantic functions are not curried or passed as parameters, then the process of deriving control graphs can be automated. The formal semantics of SPARE determines the call relationships. By providing dependence rules for each expression constructor in the SPARE language, the control graph schema for any specification can be automatically derived. By enumerating a finite set (as a consequence of Lemma

6.1 in Chapter 6 for any path problems) of possible execution paths through each program structure[36], the verification of control abstraction can be automated.

7.3.3. Evaluation schemes

It is possible to adopt the SPARE language for use in other applications such as derivation of analysis implementations for use in compilers. These applications require efficient translations for the SPARE specification language. Ideas from studies on efficient evaluators [5,45,46,51] for denotational specifications may prove to be useful here.

In Chapter 5, we indicated the limitations of our incremental evaluation scheme for evaluation of fixed-point expressions. Incremental evaluation schemes for fixed-point expressions can be extremely useful for our work as well as many other applications.

7.4. Summary

Although the notion of using denotational frameworks for specification of analysis algorithms is not new, none of the previous studies have explored the issues involved in providing an environment for the development of program analysis algorithms. A practical development environment must allow for testing as well as verification of analysis algorithms.

Chapters 1 and 2 provided an insight into the pragmatic and theoretical issues behind the use of a denotational framework for specification of analysis algorithms. Chapter 3 described the design of a specification language to address these issues. A formal semantic definition was also provided for this language. Several analysis algorithm specifications were provided in Chapter 4 to demonstrate the clarity and conciseness of specifications as well as the flexibility afforded by our meta-language design. A tool designed and built to demonstrate the feasibility of a denotational framework was described in Chapter 5. This tool can be used to derive automatic implementations from analysis specifications. Used in conjunction with the Synthesizer Generator, it provides an effective environment for testing of analysis algorithms. Chapter 6 provided guidelines for developing formal proofs of correctness for analysis specifications.

This thesis bridges the gap between theory and practice in the design of analysis algorithms. We have demonstrated that it is possible to arrive at a design that incorporates features to address both theoretical and pragmatic considerations. We feel that such an approach can ease the development of correct program analysis algorithms.

[36]This is essentially the method used in the proof of Lemma 6.2 in Chapter 6.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Prinicples, Techniques and Tools,* Addison-Wesley, Reading, Mass. (1986).

2. F. E. Allen, "Program Optimization," *Annual Review of Automatic Programming* **5** pp. 239-307 Pergamon, (1969).

3. B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," pp. 1-11 in *Proc. 15th ACM Symp. on Principles of Programming Languages,* (January 1988).

4. A. W. Appel, "Semantics-Directed Code Generation," pp. 315-324 in *Proc. 12th ACM Symp. on Principles of Programming Languages,* (January 1985).

5. A. W. Appel, "Compile-time evaluation and code generation for semantics-directed compilers," Tech. Report CMU-CS-85-147, Carnegie-Mellon University (1985). Ph.D. Thesis

6. R. Bahlke and G. Snelting, "The PSG - programming System Generator," pp. 28-33 in *Proc. 8th ACM SIGPLAN Symp. on Language Issues in Programming Environments,* (June 1985).

7. J. C. Beatty, "Register Assignment Algorithm for Generation of Highly Optimized Object Code," *IBM Journal of Research and Development* **18**(1) pp. 20-39 (1972).

8. G. Birkhoff, *Lattice Theory,* American Mathematical Society Colloquium Publications (1967).

9. R. Cartwright and M. Felleisen, "The Semantics of Program Dependence," *Sigplan Notices* **24**(7) pp. 13-27 (June 1989). Proc. SIGPLAN '89 Conf. on Programming Language Design and Implementation

10. D. Clement, "The Natural Dynamic Semantics of Mini-Standard ML," pp. 67-81 in *TAPSOFT '87 (Proceedings of the International Joint Conference on Theory and Practice of Software Development) Volume 2,* (March 1987).

11. P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," pp. 238-252 in *Proc. 4th ACM Symp. on Principles of Programming Languages,* (January 1977).

12. J. Despeyroux, "Proof of translation in Natural Semantics," *Logic in Computer Science,* pp. 193-205 (June 1986).

13. T. Despeyroux, "Executable specification of static semantics," in *Semantics of data types,* (June 1984).

14. V. Donzeau-Gouge, "Denotational Definition of Properties of Program Computations," pp. 343-379 in *Program Flow Analysis: Theory and Applications,* Prentice-Hall, Englewood Cliffs, NJ (1981).

15. R. Farrow, K. Kennedy, and L. Zucconi, "Graph Grammars and Global Program Flow Analysis," in *Proc. 17th IEEE Symp. on Foundations of Computer Science*, (1975).

16. J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. on Programming Languages and Systems* 9(3) pp. 319-349 (July 1987).

17. C. N. Fischer and R. J. LeBlanc, *Crafting a compiler*, Benjamin-Cummings, Menlo Park, CA (1988).

18. R. Harper, R. Milner, and M. Tofte, "The Semantics of Standard ML - Version 1," ECS-LFCS-87-36, University of Edinburgh (Aug 1987).

19. J. Hennessy, "Symbolic debugging of optimized code," *ACM Transactions on Programming Languages and Systems* 4(3) pp. 323-344 (1982).

20. S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence analysis for pointer variables," *Sigplan Notices* 24(7) pp. 28-40 (June 1989). Proc. SIGPLAN '89 Conf. on Programming Language Design and Implementation

21. S. Horwitz, J. Prins, and T. Reps, "Integrating Non-Interfering Versions of Programs," pp. 133-145 in *Proc. 15th ACM Symp. on Principles of Programming Languages*, (January 1988).

22. S. Horwitz, J. Prins, and T. Reps, "On the adequacy of program dependence graphs for representing programs," pp. 146-157 in *Proc. 15th ACM Symp. on Principles of Programming Languages*, (January 1988).

23. P. Hudak, "A semantic model of reference counting and its abstraction," pp. 45-62 in *Abstract Interpretation of Declarative Languages*, ed. S. Abramsky, C. Hankin,Ellis Horwood, West Sussex (1987).

24. P. Hudak and J. Young, "A Collecting Interpretation of Expressions (Without Powerdomains)," pp. 107-118 in *Proc. 15th ACM Symp. on Principles of Programming Languages*, (January 1988).

25. J. Hughes, "Analysing strictness by abstract interpretation of continuations," pp. 63-102 in *Abstract Interpretation of Declarative Languages*, ed. S. Abramsky, C. Hankin,Ellis Horwood, West Sussex (1987).

26. N. D. Jones and S. S. Muchnik, "Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra," pp. 380-393 in *Program flow analysis: Theory and applications*, Prentice-Hall (1981).

27. N. D. Jones and A. Mycroft, "Data flow analysis of applicative programs using minimal function graphs.," pp. 296-306 in *Proc. 13th ACM Symp. on Principles of Programming Languages*, (January 1986).

28. U. Jorring and W. L. Scherlis, "Compilers and staging transformations," pp. 86-96 in *Proc. 13th ACM Symp. on Principles of Programming Languages,* (January 1986).

29. G. Kahn, "Natural Semantics," *Proc. of Symp. on Theoretical Aspects of Computer Science,* (Feb 1987).

30. R. M. Keller and M. R. Sleep, "Applicative caching," *ACM TOPLAS* **8**(1) pp. 88-106 (Jan 1986).

31. K. Kennedy, "A survey of data flow analysis techniques," pp. 5-54 in *Program Flow Analysis: Theory and Applications,* Prentice-Hall, Englewood Cliffs, NJ (1981).

32. G. A. Kildall, "A Unified Approach to Global Program Optimization," *ACM Symp. on Principles of Programming Languages,* pp. 194-206 (1973).

33. D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe, "Dependence graphs and compiler optimization," pp. 207-218 in *Proc. 8th ACM Symp. on Principles of Programming Languages,* (January 1981).

34. Z. Manna, S. Ness, and J. Vuillemin, "Inductive methods for proving properties of programs," *ACM SIGPLAN Notices* **7**(1) pp. 27-50 (1972).

35. T. J. Marlowe and B. G. Ryder, "Properties of data flow frameworks: A unified model," LCSR-TR-103, Rutgers University (April 1988).

36. D. Michie, " "Memo" functions and machine learning," *Nature* **218** pp. 19-22 (1968).

37. P. Mosses, "SIS - Semantics Implementation System. Reference Manual and User Guide," Report DAIMI MD-30, Aarhus University (August 1979).

38. J. Mostow and D. Cohen, "Automating program speedup by deciding what to cache," *Proc. of the Ninth Int. Joint Conf. on Artificial Intelligence* **1** pp. 165-172 (1985).

39. K. Mulmuley, "Full abstraction and semantic equivalence," Ph.D. thesis, Carnegie-Mellon University (1985).

40. A. Mycroft, "Abstract Interpretation and Optimising Transformations for Applicative Programs," Ph. D. Thesis, University of Edinburgh (December 1981).

41. F. Nielson, "A denotational framework for data flow analysis," *Acta Informatica* **18** pp. 265-287 (1982).

42. F. Nielson, "Expected forms of data flow analyses," pp. 172-191 in *Programs as data objects,* Springer-Verlag, Berlin (1985).

43. F. Nielson, "Abstract interpretation of denotational definitions," pp. 1-20 in *Proc. STACS 1986,* Springer-Verlag, Berlin (1986).

44. A. Pal, "Generating Execution Facilities for Integrated Programming Environments," Tech. Report #676, University of Wisconsin-Madison (1986). Ph.D. thesis

45. Lawrence Paulson, "A Semantics-Directed Compiler Generator," pp. 224-233 in *Proc. 9th ACM Symp. on Principles of Programming Languages*, , Albuquerque, New Mexico (January 25-27, 1982).

46. U. F. Pleban, "Compiler Prototyping Using Formal Semantics," pp. 94-105 in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, , Montreal, Canada (June 1984).

47. G. D. Plotkin, "LCF considered as a programming language," *Theoretical Computer Science* **5** pp. 223-255 (1977).

48. G. D. Plotkin, "A structural approach to operational semantics," DAIMI FN-19, Aarhus University (Sept 1981).

49. D. Prawitz, *Natural deduction - A proof-theoretical study*, Almqvist & Wiksell, Stockholm (1965).

50. W. W. Pugh, "Incremental computation and the incremental evaluation of function programs," Tech. Report 88-936, Cornell University (1988). Ph.D. thesis

51. J. Raoult and R. Sethi, "Properties of a notation for combining functions," *JACM* **30**(3) pp. 595-611 (1983).

52. T. Reps and T. Teitelbaum, *The Synthesizer Generator Reference Manual*, Springer-Verlag, New York (Third Edition, 1988).

53. B. K. Rosen, "High-Level Data Flow Analysis," *CACM* **20**(10) pp. 712-724 (1977).

54. B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," pp. 12-27 in *Proc. 15th ACM Symp. on Principles of Programming Languages*, (January 1988).

55. D. A. Schmidt, *Denotational Semantics - A methodology for language development*, Allyn and Bacon, Boston (1986).

56. R. Sethi, "Control Flow Aspects of Semantics-Directed Compiling," *ACM Transactions on Programming Languages and Systems* **5**(4)(1983).

57. J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, Cambridge, Massachusetts, and London, England (1977).

58. R. D. Tennent, "The denotational semantics of programming languages," *Comm. of the ACM* **19** pp. 437-452 (1976).

59. G. A. Venkatesh, "A framework for construction and evaluation of high-level specifications for program analysis techniques," *Sigplan Notices* **24**(7) pp. 1-12 (June 1989). Proc. SIGPLAN '89 Conf.

on Programming Language Design and Implementation

60. G. A. Venkatesh and C. N. Fischer, "SPARE: Reference Manual," Tech. Report #850, University of Wisconsin-Madison (June 1989).

61. G. A. Venkatesh and C. N. Fischer, "SPARE : Formal Semantics," Tech. Report #873, University of Wisconsin-Madison (1989).

62. M. Wand, "A semantic prototyping system," *Sigplan Notices* **19**(6) pp. 213-221 (June, 1984). Proc. of the Sigplan '84 Sym. on Compiler Construction

63. B. Wegbreit, "Property extraction in well-founded property sets," *IEEE Trans. on Software Engineering* **SE-1**(3) pp. 270-285 (Sept 1975).

64. M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," Tech. Report #CS-88-02, Brown University (1988).

65. M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).