Semantics-based Program Analysis via Symbolic Composition of Transfer Relations

Christopher Colby

August 16, 1996

CMU-CS-96-162

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Peter Lee, Chair
Robert Harper
John Reynolds
Patrick Cousot, École Normale Supérieure

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Keywords: program analysis, abstract interpretation, symbolic execution, program verification, compilers, debugging, operational semantics, functional languages, imperative languages

To M and D.

Abstract

The goal of program analysis is to determine automatically properties of the run-time behavior of a program. Tools of software development, such as compilers, program-verification systems, and program-comprehension systems, are in large part based on program analyses. Most semantics-based program analyses model the run-time behavior of a program as a trace of execution states and compute a property of these states. Typically, this property is drawn from a predetermined language of semantic information, such as aliasing descriptions or types of values. The standard methodology of program analysis is to construct the property as a fixed point, a single execution step at a time. We explain that these ubiquitous methodological choices—the *a priori* choice of the describable program properties and the use of a fixed-point computation—have some fundamental limitations and can result in poor precision.

In this dissertation, we present a different approach to semantics-based program analysis. Our methodology is based on *transfer relations* that precisely describe the changes between the state of memory one point during execution and the state of memory at some later point in the execution. We isolate a language TR of concise computer-representable presentations of transfer relations. We also give an algorithm $\oplus$ that, given two transfer relations from TR, symbolically constructs a third transfer relation in TR that is semantically equivalent to their relational composition. An analysis designer begins by describing the operational semantics of a source language as a set of TR-terms that precisely describe the atomic steps of execution. Then an analysis algorithm repeatedly applies $\oplus$ to build a precise run-time description of any finite control path of interest.

We show that TR is expressive enough to describe a wide variety of source-language features, including heap-allocated mutable data structures, arrays, pointers, and first-class functions. We then explain how our analysis methodology overcomes some current limitations of program analysis. The transfer relations themselves are useful program properties and would be difficult or impossible to formulate with classical approaches to program analysis. But we also describe some classes of analysis applications that are based on transfer relations. For instance, we explain that the classical limitation of program analysis to build a property a single execution step at a time can result in dramatic loss of precision, but may be overcome by using $\oplus$ to compose multiple steps before applying a classical analysis. Furthermore, we show how to compute precise properties of loops symbolically, avoiding the inevitable imprecision of a fixed-point computation.

Acknowledgments

At long last, it is my extreme pleasure to thank my advisor, Peter Lee. Peter has been a great mentor, colleague, and friend during my time at Carnegie Mellon, and I am quite fortunate to have found in an advisor such uncanny patience and understanding. I could not have asked for a better role model than Peter, and I owe him a lifelong gratitude.

I am very happy indeed to thank Robert Harper. During my time as a graduate student, Bob's steadfast devotion to uncompromising standards has been a constant inspiration of the highest order, and my life is far richer for having known him. It is also a great pleasure to thank John Reynolds for his role on my committee. In many ways, John and Bob understood my work better than I did, and I am sure that I will return to their valuable insights many times. I am deeply honored and fortunate to have known them both.

I owe a most profound debt to Patrick and Radhia Cousot, who hosted me for a year at the Laboratoire d'Informatique of École Polytechnique and at École Normale Supérieure. In the eleventh hour, they reminded me of much that I had forgotten. I offer Patrick and Radhia my deepest gratitude for their invaluable gifts: a tireless sponsorship, a belief in my abilities, a beacon of scientific passion and integrity, and an extraordinary year in Paris. I also thank Patrick for serving on my committee.

Many people helped make me feel welcome during my year on the Quartier Latin's Place de la Contrascarpe. I would like to thank Régis Cridlig, Éric Goubault, Thomas and Sandra Jensen, Ian Mackie, Bruno Monsuez, and Arnaud Venet. They all made my stay in Paris very special. A special thanks goes to Alain Deutsch, whose work opened my eyes to what program analysis can be. I later received similar inspirations from the work of Philippe Granger.

It is my great pleasure to thank Andrew Tomkins and Glen Wilk. Glen and I have been close friends for the better half of my life. He has always shown a keen interest in my career, and he has been a solid foundation of encouragement. Andrew and I entered Carnegie Mellon together and immediately became great friends. Andrew is amazingly kind and generous, and he as much as anyone has helped me make it through graduate school. Glen and Andrew are both wonderful people, and they are the best friends anyone could ever hope to have.

I would also like to thank Rob Adams, Jill Colby, Scott Colby, Kathleen Downey, Scott Draves, Ben Fine, Sue Lee, Mark Leone, Bryan Loyall, Chris Okasaki, Jay Sipelstein, Steve Weeks, Chuck and Penny Wilk, and the countless others who joined me as traveling companions on my journey.

Finally, I would like to thank my parents, Jim and Gloria. Nothing that I have achieved would have been remotely possible without their unconditional love, encouragement, and support, unwavering through both good times and bad.

Contents

Part I

Introduction

Chapter 1

Some Topics in Program Analysis

The goal of program analysis is to determine automatically at compile time some properties about the run-time behavior of a program. There are several major applications of program analysis.

- **Compiler support**. It is reasonably straightforward to implement a correct compilation of a program from a high-level language to machine code, but it is not as easy to implement a high-quality compilation. This is because the program may have a specialized run-time behavior that the compiler could exploit, but this run-time behavior may not be easy to detect from a simple examination of the code. Therefore, the compiler must invoke a program analysis to uncover this run-time behavior. For instance, most compilers use *data-flow analysis* (e.g., [KU76], [MJ81]) and *alias analysis* (e.g., [CWZ90], [Lan91], [Deu94]) to enable classic optimizations such as common-subexpression elimination, copy propagation, and hoisting of loop-invariant computations [ASU86]. Similarly, some compilers for languages with first-class functions use a *control-flow analysis* (e.g., [JM79], [Shi91]) to construct a conservative control graph. Compiler support is far and away the most common application of program analysis.

- **Program verification**. One would like to check statically that a program will behave properly at run time. For instance, an analysis might verify that a C program never attempts to dereference a dangling pointer; or if it cannot verify a property that strong, it might at least isolate a small number of potential trouble spots in the code. Also, strongly typed languages such as Standard ML [MTH90] verify at compile time that a program is well-typed and thus completely eliminate any possibility of a type error at run time. Furthermore, static type-checking reveals at compile time a remarkable percentage of programmer errors.

- **Program comprehension**. A subject that has been gaining interest in recent years is the use of program analysis to aid the human understanding of code. For instance, the work in *static debugging* [Bou93a, Bou93b] allows the user to specify various kinds of pre- and post-conditions at different points in the program, and then calculates the corresponding information about the ranges of numeric variables. Also, *program slicing* (e.g., [HRB90], [FRT95]) isolates the parts of a program that contribute to or depend on a particular variable in the program chosen by the user.

This dissertation presents some new developments in the theory of program analysis. By "theory of program analysis" we mean that we are concerned less with specific analysis problems or specific applications of program analysis, and more with generic semantic tools that are powerful and yet easy to apply to a variety of real programming languages and analysis tasks.

To put our goals into perspective, we compare them to the goals of abstract interpretation. Abstract interpretation [CC77] is a general theory of semantics-based program analysis—so general and wide-ranging that the theory itself intentionally does not provide explicit support for particular language features, such as data structures and functions, or particular applications, such as alias or data-shape analysis. A powerful methodology has been constructed around this theory [CC79, Cou81, Cou90, CC92a, CC92b, CC92d, CC92c, CC94, CC95], including a wide range of techniques for designing numeric lattices [Kar76, CH78, Gra89, Gra91a, Gra91b]. But when faced with a specific analysis task for a specific programming language, the analysis designer is left largely on his own to cope with the overwhelming generality of the framework. With a deep understanding and skillful use of the methodology, the results can be spectacular, such as the storeless alias analysis of Deutsch [Deu92, Deu94]. But after 20 years, much of the staggering potential of abstract interpretation still remains largely untapped.

In contrast, our methodology is designed around real language features, such as pointers, heap-allocated data structures, arrays, assignment, and to a lesser extent first-class functions. Consequently, although our framework does not have the same level of generality as abstract interpretation, it is more straightforward to apply our tools to real languages and real analysis tasks. We aim to strike a balance between analysis theory and analysis design. One of our goals is to bring some of the power of semantics-based analysis techniques closer to the user.

To accomplish this, we have taken a step back in order to consider the task of program analysis from a fresh perspective. This new perspective has uncovered some fundamental limitations in the current methodology of program analysis—limitations that are manifest in real analyses. By largely reworking semantics-based program analysis from the beginning, this dissertation provides some technical answers to these basic limitations.

1.1 Limitations of Single-step Abstract Interpretation

We begin with an anecdote. Imagine that you are asked to report the sum to two decimal places of the following list of numbers:

$$\begin{array}{l} 2.5548 \\ 1.3475 \times 10^{-1} \\ 9.971 \\ 3.802 \times 10^{-3} \\ 2.388 \times 10^{2} \\ 5.262 \end{array}$$

Consider these two different approaches:

- Algorithm A: Compute the exact sum of all six numbers and then round that sum to two decimal places.
- Algorithm B: Begin with 0, and then add the first number, round to two decimal places, add the second number, round to two places, add the third, round again, and so forth.

Algorithm A is the procedure that naturally comes to mind for this task, and of course it returns the correct answer of 256.73. In contrast, Algorithm B reports a result of 256.71, which is close but not correct. Why would anyone choose this second approach? One can imagine that the reduction in computation effort is worth the potential for accumulated rounding error.

In fact, these two algorithms are just the endpoints of a spectrum of possibilities. For instance, one could first compute the precise sum of adjacent pairs of numbers in the list, yielding a list of three exact partial sums:

$$\begin{array}{llllll} 2.5548 & + & 1.3475 \times 10^{-1} & = & 2.68955 \\ 9.971 & + & 3.802 \times 10^{-3} & = & 9.974802 \\ 2.388 \times 10^{2} & + & 5.262 & = & 2.44062 \times 10^{2} \end{array}$$

Then apply Algorithm B to this list, yielding a better but still not exact 256.72. This suggests a general approach of rounding only every so often during the accumulation of the sum, where Algorithm A is the extreme that rounds only at the very end, while Algorithm B is the other extreme that rounds after every single number in the list.

This simple discourse on how to compute rounded sums illustrates by analogy a remarkably important limitation of program analyses. As a very simple example, consider the following program.

```
while n > 0 do
{
  y := x − 3;
  x := y + 5;
  n := n − 1
}
```

Suppose that at any point during an execution of this program, the variable bindings are described by an environment

$$\rho \ \in \ \mathsf{Env} \ = \ \mathsf{Var} \to \mathsf{Int}$$

Say we wish to determine the variable bindings at the termination of this program, given an environment ρ_0 describing the initial bindings. Because this program always terminates (as long as `n`, `x`, and `y` are bound in ρ_0), we can in fact just execute the program and return the final environment as the answer.

By analogy, an entire environment ρ corresponds to a real number, and the execution of a single step of the program (which may modify the environment) corresponds to the accumulated addition of one number in the list.[1] Thus, executing the program corresponds to accumulating the exact sum of a list of real numbers (starting with 0). The length of the list is the total number of execution steps, which in this case is always finite, but may be quite long.

But suppose that all we want to know about each variable at the end of execution is information about its sign, expressed as one of the following properties of integers ordered by implication (in other words, sets of integers ordered by inclusion).

$$\begin{tikzcd}
& & \texttt{int} & & \\
& \texttt{nonpos} \arrow[ur] & & \texttt{nonneg} \arrow[ul] & \\
\texttt{neg} \arrow[ur] & & \texttt{zero} \arrow[ul] \arrow[ur] & & \texttt{pos} \arrow[ul] \\
& & \texttt{none} \arrow[ull] \arrow[u] \arrow[urr] & &
\end{tikzcd}$$

Given an environment ρ, one can *abstract* ρ by a *sign environment* $\hat{\rho}$ such that $(\hat{\rho}\,x)$ is the sign (either `neg`, `zero`, or `pos`) of $(\rho\,x)$ for all variables x.

$$\hat{\rho} \ \in \ \widehat{\mathsf{Env}} \ = \ \mathsf{Var} \to \mathsf{Sign}$$

By analogy, $\hat{\rho}$ corresponds to the “rounding” of ρ. Again, we can just execute the program and “round” the final environment to a sign environment. This is analogous to Algorithm A, and it will always return the strongest properties.

It is well known, however, that this process is infeasible in general. For one, the program may take a long time to execute. Even worse, we may not know the exact initial environment ρ_0. Finally, some programs do not terminate, and even if we do know ρ_0 beforehand, it is impossible to determine effectively if the execution will eventually halt. So in general we must settle for some approximation of the result.

The standard approach to program analysis is essentially to perform Algorithm B, abstracting at each step. For our example program, the first three steps would produce the following

[1] This analogy has the disadvantage that a real number corresponds to both an environment and a single-step transformation between environments; it is crucial to distinguish these two very different concepts.

sign environments from the initial sign environment shown below:

```
                       $[\mathtt{x}, \mathtt{y}, \mathtt{n} \mapsto \mathtt{pos}, \mathtt{pos}, \mathtt{nonneg}]$
**while** $\mathtt{n} > 0$ **do**
{                      $[\mathtt{x}, \mathtt{y}, \mathtt{n} \mapsto \mathtt{pos}, \mathtt{pos}, \mathtt{pos}]$
  $\mathtt{y} := \mathtt{x} - 3$;
                       $[\mathtt{x}, \mathtt{y}, \mathtt{n} \mapsto \mathtt{pos}, \mathtt{int}, \mathtt{pos}]$
  $\mathtt{x} := \mathtt{y} + 5$;
                       $[\mathtt{x}, \mathtt{y}, \mathtt{n} \mapsto \mathtt{int}, \mathtt{int}, \mathtt{pos}]$
  $\mathtt{n} := \mathtt{n} - 1$
}
```

Before the first assignment, all that is known about `x` is that it is positive, but the analysis must calculate $\mathtt{x} - 3$ to determine the value of `y`. The exact answer is the set of all positive integers decremented by 3, which is the set $\{-2, -1, 0, \ldots\}$, but the abstraction "rounds" that set to the smallest enclosing element in the sign lattice, which is `int`. Now, in the next step, all that is known about `y` is that it is an integer, and so $\mathtt{y} + 5$ is the set of all integers incremented by 5, which is again the set of integers. So the abstract value of `x` in the next step is `int`.

However, a little bit of thought reveals that `x` is actually guaranteed to be positive after the second assignment. The reason the analysis has already lost this information is because of the abstraction, or "rounding error", between the two assignments. If the set $\{-2, -1, 0, \ldots\}$ had not been abstracted to `int`, then its increment by 5 in the next step would yield the set $\{3, 4, \ldots\}$, whose abstraction is `pos`. Thus, there are two ways to achieve better results.

1. Do not abstract between the first and second assignments.
2. Abstract after every step as usual, but beforehand enrich the lattice of integer properties with an element corresponding to $\{-2, -1, 0, \ldots\}$, so that the abstraction of this intermediate property loses no information.

The first approach seems promising, but is not in the current repertoire of program analysis techniques. Most of this dissertation develops a a general foundation that one may use for this approach; we will return to it shortly.

The second approach seems absurd from a practical standpoint and troublesome from a theoretical standpoint. It clearly does not generalize. For instance, elements such as $\{-2, -1, 0, \ldots\}$ are clearly ad hoc and dependent on the particular run-time behavior of a program. Probably many new elements would be needed for a reasonably sized program, and for anything more sophisticated than a sign analysis, the space from which these elements may be chosen becomes much more complex and rich. Even if one could isolate a small set of useful specialized elements with which to enrich the property lattice for a given program, it seems difficult to determine which properties would be the most useful without actually running the program itself. Nevertheless, there are examples of practical program analyses that essentially use this idea in limited capacity for the lack of any other solution; we give an example at the end of this section.

To continue with the analysis of this program, there is the additional complication that the execution length (corresponding to the length of the list of numbers) may be unbounded, and so an analysis will typically use some "folding" strategy, usually at every program point, and compute the solution as an iterative fixed-point calculation. For example, the next step above is to calculate $\texttt{pos} - 1 = \texttt{nonneg}$ for n, and then to join the resulting sign environment with the old environment at the loop entry, weakening the properties of x and y to int. The analysis reaches the following fixed point after a second iteration through the loop:

$$\begin{array}{llcll}
 & [\mathtt{x},\mathtt{y},\mathtt{n} & \mapsto & \texttt{int}, \texttt{int}, \texttt{nonneg}] & \nwarrow \\
\textbf{while}\ \mathtt{n} > 0\ \textbf{do} & & & & | \\
\{ & [\mathtt{x},\mathtt{y},\mathtt{n} & \mapsto & \texttt{int}, \texttt{int}, \texttt{pos}] & | \\
\quad \mathtt{y} := \mathtt{x} - 3; & & & & | \\
 & [\mathtt{x},\mathtt{y},\mathtt{n} & \mapsto & \texttt{int}, \texttt{int}, \texttt{pos}] & | \\
\quad \mathtt{x} := \mathtt{y} + 5; & & & & | \\
 & [\mathtt{x},\mathtt{y},\mathtt{n} & \mapsto & \texttt{int}, \texttt{int}, \texttt{pos}] & | \\
\quad \mathtt{n} := \mathtt{n} - 1 & & & & | \\
 & [\mathtt{x},\mathtt{y},\mathtt{n} & \mapsto & \texttt{int}, \texttt{int}, \texttt{nonneg}] & \nearrow \\
\} & & & & \\
 & [\mathtt{x},\mathtt{y},\mathtt{n} & \mapsto & \texttt{int}, \texttt{int}, \texttt{zero}] &
\end{array}$$

The last environment is the answer. But the most precise answer (corresponding to the "correct" rounded sum) is

$$[\mathtt{x},\mathtt{y},\mathtt{n} \mapsto \texttt{pos}, \texttt{int}, \texttt{zero}].$$

As we have suggested, the reason that the analysis reported the final sign of x as int instead of pos is because it used the equivalent of Algorithm B, which is the extreme approach of abstracting at every step. Algorithm A is at the other extreme, which as we have explained is uncomputable for program analysis. But what about the intermediate approach of "rounding only every so often"? To understand how that applies to program analysis, consider rewriting the program to use a parallel assignment:

$$\begin{array}{l}
\textbf{while}\ \mathtt{n} > 0\ \textbf{do} \\
\{ \\
\quad \mathtt{x},\mathtt{y},\mathtt{n} := \mathtt{x} + 2, \mathtt{x} - 3, \mathtt{n} - 1 \\
\}
\end{array}$$

Now apply the approach of Algorithm B:

$$\begin{array}{llcll}
 & [\mathtt{x},\mathtt{y},\mathtt{n} & \mapsto & \texttt{pos}, \texttt{int}, \texttt{nonneg}] & \nwarrow \\
\textbf{while}\ \mathtt{n} > 0\ \textbf{do} & & & & | \\
\{ & [\mathtt{x},\mathtt{y},\mathtt{n} & \mapsto & \texttt{pos}, \texttt{int}, \texttt{pos}] & | \\
\quad \mathtt{x},\mathtt{y},\mathtt{n} := \mathtt{x} + 2, \mathtt{x} - 3, \mathtt{n} - 1 & & & & | \\
 & [\mathtt{x},\mathtt{y},\mathtt{n} & \mapsto & \texttt{pos}, \texttt{int}, \texttt{nonneg}] & \nearrow \\
\} & & & & \\
 & [\mathtt{x},\mathtt{y},\mathtt{n} & \mapsto & \texttt{pos}, \texttt{int}, \texttt{zero}] &
\end{array}$$

This returns the most precise answer possible. Note how this approach was able to determine the precise result for `x`. Before the assignment, `x` is `pos`. So, $x + 2$ is the set $\{3, 4, \ldots\}$, which is then abstracted to `pos`.

Technically, we are still abstracting after every step and using the same sign analysis to do it, but by rewriting the three sequential instructions into a single parallel instruction, we are in effect abstracting only after every third step. Recall that in our analogy, a real number corresponds both to an environment (the accumulated result) and a single-step transition between environments (an element of the list). Here, the transitions are done by assignment statements, so this transformation from multiple sequential statements to a single parallel statement corresponds to adding groups of adjacent numbers in the list before applying Algorithm B.

The above is of course merely a toy example. But it is not hard to find examples in real program analyses that suffer from this same phenomenon of abstracting after every step. For instance, Ghiya and Hendren describe in [GH96] a shape analysis that attempts to determine whether data structures in a C program are trees, dags, or graphs. Their paper describes a difficulty with their analysis:

> If a data structure temporarily becomes dag-like or cyclic and then becomes tree-like again, shape analysis cannot detect this, and continues to report its shape as dag-like or cyclic. The benchmark *reverse* that recursively swaps [the children of] a binary tree represents this case.

Although shape analysis for C is quite a bit more complex than a sign analysis for a simple arithmetic while-loop language, it turns out that the difficulty that Ghiya and Hendren described is *precisely* the same phenomenon that caused the sign analysis above to fail to detect that `x` is always positive. The fundamental reason that they cannot detect those *temporary* changes of shape is that they abstract at every step. In their case, they abstract a C memory state by a "direction matrix" and an "interference matrix"; and whereas our problem in the program above was that our lattice of sign properties could not precisely express the set $\{-2, -1, 0, \ldots\}$ that came up after the second step, their problem is that their abstract store cannot express many of the possible forms of non-tree or non-dag shapes that may arise *temporarily* during execution.

This is a problem not just with Ghiya and Hendren's shape analysis. At the same conference, Sagiv, Reps, and Wilhelm presented a shape analysis that attempts to address these issues [SRW96]. They point out that:

> The third and fourth common list-manipulation operations—splicing a new element into a list and removing an element from a list—can, in many cases, be handled accurately by our shape-analysis algorithm, *even if shape-nodes temporarily become shared!*

But they, too, abstract after every single step. In order to achieve good results for some programs that temporarily alter data shapes, they instead chose to design a rather unusual abstraction of a memory state that *can* actually express certain kinds of temporary shape alterations that might arise in common programs. In spirit, their solution is item 2 on page 7. As we suggested there, this approach does not generalize very well and is necessarily limited at the outset; this is indeed the case for their analysis. Because of their specialized lattice design, their analysis determines very little information about any program that allocates at least one pointer that is at some point shared (pointed to by more than one distinct location in memory) and not itself the binding of any variable. Clearly, this eliminates a great many programs from consideration—for instance, any program that creates a doubly-linked list or any kind of dag-like structure. In contrast, the Ghiya/Hendren analysis is not nearly so limited.

Our claim is that the methodology of abstracting at every step is a ubiquitous and serious limitation of current program-analysis methodology. To understand why, we will revisit abstract interpretation, the root of semantics-based program analysis, in Chapter 8. This dissertation will provide a solution, which we will outline in Section 1.3.

1.2 Overuse of Abstraction and Fixed-point Computation

Our discussion of the sign analysis in the previous section centered around how to deal with a single loop iteration. We only touched upon the "folding" process that was necessary to deal effectively with the unbounded execution length of the program. The issue of how to cope with infinite execution sequences is of primary importance in program analysis, and almost all analyses use a similar technique of computing a fixed point over an abstract semantic domain (sign environments in the above example).

Our claim that this technique is rather ubiquitous and yet not well suited for many analysis tasks. The cause of this state of affairs is, perhaps surprisingly, strongly related to the cause of the problem described in the previous section: that analyses cannot take multiple steps of execution between abstraction. Fortunately, the solutions to these two problems are closely related, as well, and in this dissertation we develop the foundations for both.

In Chapter 8 we will see that the foundation of semantics-based program analysis is based on an observation that a semantics of a language is usually expressed using a fixed point whose iterative calculation corresponds in some sense to the execution steps of the program. For instance, consider the common form of operational semantics as a transition system, in which program execution is modeled by the single-step transitions from machine state to machine state. This kind of semantics is particularly useful for program analysis because it expresses many intensional details of execution that might be of interest to analyze; one might say that it is "close to the iron", in comparison to a more extensional semantics such as a standard denotational model that only maps program input to program output. We will say more about this in Chapter 4.

For now, we are not so much concerned with the appropriateness of a particular semantic model for the purpose of program analysis, but rather we wish to illustrate that semantic models of programming languages typically use fixed points that reflect program execution. For example, a transition system of a particular program P will have a binary transition relation

$$\longmapsto \subseteq \mathsf{State} \times \mathsf{State}$$

specifying the pairs of states that may be adjacent in an execution of that program. Then the semantics $\mathcal{M}[\![P]\!]$ of program P is defined as an unfolding of this relation into a set of unbounded sequences (where $\vec{\psi}.\psi$ denotes the extension of state sequence $\vec{\psi}$ by state ψ):

$$\frac{\vec{\psi}.\psi \in \mathcal{M}[\![P]\!] \quad \psi \longmapsto \psi'}{\vec{\psi}.\psi.\psi' \in \mathcal{M}[\![P]\!]}$$

If this rule is solved inductively from a base set of initial states, its iterative solution yields all finite execution prefixes.[2]

One can rephrase the iterative solution of the above rule as the repeated application of a function

$$\mathcal{S}[\![P]\!] \in \mathcal{P}(\mathsf{State}^*) \rightarrow \mathcal{P}(\mathsf{State}^*)$$

that, given a partial solution of $\mathcal{M}[\![P]\!]$, applies the above rule once to enlarge $\mathcal{M}[\![P]\!]$ by a single execution step.

A program analysis based on this transition-system semantics must analyze these potentially unbounded sequences. For instance, suppose that in our sign analysis above, a state comprises a control point specifying the line of the program to be executed next and an environment specifying the current variable bindings.

$$\mathsf{State} \quad = \quad \mathsf{CtrlPoint} \times \mathsf{Env}$$

The analysis that we described informally above can now be formalized as an iteration of

$$(\alpha \circ \mathcal{S}[\![P]\!] \circ \gamma) \in \widehat{\mathsf{State}} \rightarrow \widehat{\mathsf{State}}$$

until a fixed point is reached, where

$$\begin{array}{rcl} \gamma & \in & \widehat{\mathsf{State}} \rightarrow \mathcal{P}(\mathsf{State}^*) \\ \alpha & \in & \mathcal{P}(\mathsf{State}^*) \rightarrow \widehat{\mathsf{State}} \end{array}$$

and

$$\widehat{\mathsf{State}} \quad = \quad \mathsf{CtrlPoint} \rightarrow \widehat{\mathsf{Env}}.$$

Here, a member of $\widehat{\mathsf{State}}$ is a table of abstract environments indexed by control point, just as we showed next to the program in the examples of Section 1.1. The function γ, given such a

[2] There are similar ways to express the infinite executions of a program via coinduction, but for the sake of simplicity we leave the reader to [CC92b] for a discussion. We do note, however, that the use of coinduction for program analysis is powerful technique, especially for the analysis of errors, that is currently not well appreciated. For examples, see [Bou93a].

table $\hat{\Psi}$, describes the set of all execution sequences whose states satisfy the properties given in $\hat{\Psi}$. The function α abstracts a set of execution sequences by a table giving the strongest sign properties of the states in those sequences.

In the analogy of Section 1.1 in which we compared program execution and analysis to the accumulation of the rounded sum of a list of numbers, a set of execution sequences corresponds to an "exact" real number, and a member of $\widehat{\mathsf{State}}$ corresponds to a "rounded" real number. The function γ is given a rounded number representing the accumulated sum at some point in the middle of the list. Conceptually, γ "coerces" this number into an exact number by adding zeroes onto the end. Then $\mathcal{S}[\![P]\!]$ corresponds to adding the next number in the list to this sum, and α rounds the resulting sum, usually losing information. The program analysis repeats this process until it reaches a fixed point (which does not have a clear analog in our list-sum anecdote).

Almost every kind of program analysis is based on a similar notion of fixed-point calculation over an abstraction of the properties of interest. This is not always apparent, because many analysis frameworks, such as data-flow analysis [MJ81], type inference [KMP84], and constraint-based analysis [Hei92, AWL94], are phrased in terms of systems of equations or inference rules. But most of these frameworks reduce to a fixed-point calculation whose iterations correspond in some sense to abstract execution steps of the program. Abstract interpretation is a fixed-point-based theory that unifies these seemingly disparate approaches.

In Section 1.1 we explained that this methodology of abstracting after every step can cause severe precision problems with the analysis. In our small while-loop example, we illustrated this problem by rewriting the three individual assignments in the loop body as a single parallel assignment. In Chapter 8 we will go further into that topic, but for now we suggest that a *multi-step* program analysis might amount to finding the fixed point of

$$(\alpha \circ \mathcal{S}[\![P]\!] \circ \mathcal{S}[\![P]\!] \circ \mathcal{S}[\![P]\!] \circ \gamma) \in \widehat{\mathsf{State}} \to \widehat{\mathsf{State}}$$

instead of the above function that takes only a single step between applications of the abstraction function α. The problem is that there is no general methodology to develop program analyses that have this kind of flexibility. But we have developed such a methodology, which we outline in Section 1.3.

Now we may make the following key insight. Once one has a methodology to perform any number of steps between abstractions, the need to perform the abstractions and compute the fixed point often evaporates.

For instance, shape analyses are often concerned with detecting computations that are *shape-preserving*. It is common for the success or failure of a shape analysis to be measured by how well it analyzes routines such as list-insert, list-delete, node swapping, and so forth. For instance, one would like to determine that a routine that destructively inserts a node into a linked list preserves the invariant that the structure upon which it operates has the shape of a list. Routines such as these typically take more than one instruction, but still a finite number of them. Why would they need an iterative fixed-point calculation to compute their shape-preserving properties? The answer is that they do not, but because the present methodology of program analysis does not offer any way to combine multiple execution steps, a shape analysis has no choice but to perform a global fixed point as we did for the sign properties in Section 1.1.

1.3 An Introduction to Our Methodology

This dissertation develops a foundation for a new methodology of program analysis that addresses the problems that we have described above. This foundation is based on a semantic methodology of programming languages in which it is possible to compute a simple term describing the net effect of any given finite execution path.

In Section 1.2 we suggested that an operational semantics based on a transition system between execution states is particularly useful for program analysis. For the example program in Section 1.1, an execution state was a pair of a control point and an environment. In general, environments are not expressive enough because they cannot express pointers and other kinds of mutable data structures.

In order to address a wide variety of languages, we introduce the notion of a *store*. A store is similar to an environment in that it maps variables to values, but it also maps *references* to values. A reference is a pair of two values; the reference (v, v') is written $v.v'$ and represents component v' of data structure v. Actually, it is convenient to think of a store as a graph whose nodes are values and whose edges are labeled by values. Then v is the root node of some data structure (record, pointer, array, and so on), and its outgoing edges point to its mutable subcomponents, labeled by their names v' (field names, the C "*" token, integer array indices, and so forth). An *l-value* is an object that may be dereferenced in a store; it is either a variable x or a reference $v.v'$. A store is then a map from l-values to values.

$$\begin{aligned} \sigma \ \in \ \mathsf{Store} &= \mathsf{Lval} \to \mathsf{Val} \\ \mathsf{Lval} &= \mathsf{Var} \cup (\mathsf{Val} \times \mathsf{Val}) \end{aligned}$$

The set Val of values is left unspecified because different languages will need different values. We consider this parameterized notion of a store, however, to be common to all languages.

More specifically, the techniques in this dissertation apply to any language in which the execution(s) of a program can be expressed as a transition relation

$$\longmapsto \subseteq \mathsf{State} \times \mathsf{State}$$

where

$$\mathsf{State} = \mathsf{CtrlPoint} \times \mathsf{Store}$$

for some set CtrlPoint of static control points and some set Val of values.

Usually, $\longmapsto$ is defined by meta-rules that specify how the individual pieces of program syntax induce transitions. For instance, one might imagine the following rule for variable assignments.

$$(x := e;\ t, \sigma) \ \longmapsto \ (t, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma])$$

Here, e is a basic expression, and $\mathcal{E}[\![e]\!]\sigma$ denotes the value to which e evaluates in store σ. The core idea of our technique is to replace these meta-rules with *computer-representable composable descriptions.*

Our first observation is the isomorphism

$$\mathcal{P}(\mathsf{State} \times \mathsf{State}) \;\simeq\; \mathsf{CtrlPoint} \times \mathsf{CtrlPoint} \to \mathcal{P}(\mathsf{Store} \times \mathsf{Store}).$$

This means that a transition relation $\longmapsto$ is equivalent to a table of binary relations on stores, indexed by pairs of control points. We write the (C, C') entry in this table as $\overset{C,C'}{\longmapsto}$, and this relation defines the possible store changes in a single step from C to C'. Thus,

$$(C, \sigma) \;\longmapsto\; (C', \sigma') \quad \text{iff} \quad \sigma \overset{C,C'}{\longmapsto} \sigma'.$$

For example, one can rewrite the above meta-rule as

$$\sigma \overset{(x{:=}e;\,t),t}{\longmapsto} \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]$$

or, alternatively, as the definition

$$\overset{(x{:=}e;\,t),t}{\longmapsto} = \{(\sigma, \sigma') \mid \sigma' = \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\}.$$

We call a binary relation on stores a *transfer relation.* A transfer relation describes a way in which a store evolves during execution. For example, $\overset{C,C'}{\longmapsto}$ is a transfer relation that describes how the store changes in a single step from C to C'. A nice property of transfer relations is that one may compose them to express multiple steps of execution. For instance,

$$\overset{C,C'}{\longmapsto}; \overset{C',C''}{\longmapsto}$$

is a transfer relation that expresses how a store changes in an execution that begins at control point C, progresses in one step to C', and then progresses in the next step to C''. Here, the symbol ";" is the relation composition operator. In this manner, one can build the transfer relation for any finite control path.

Above, we said that our central approach is to replace the meta-rules of the transition system with computer-representable composable relations: computer-representable because they will be directly manipulated and examined by a program analysis, and composable because we want a flexible way of processing multiple execution steps in the analysis before abstracting the result, as we explained in the example of Section 1.1 and more generally in Section 1.2.

Let us examine this more closely. As we explained in Section 1.2, an algorithm for analyzing program P works by iteratively applying an abstract step function

$$(\alpha \circ \mathcal{S}[\![P]\!] \circ \gamma) \in \widehat{\mathsf{State}} \to \widehat{\mathsf{State}}$$

where $\widehat{\mathsf{State}}$ is a set of abstract properties of state sequences (such as the signs of the numeric values occurring in the states), and the application of this function to $\hat{\Psi} \in \widehat{\mathsf{State}}$ applies the transition relation $\longmapsto$ to extend by one step every execution sequence consistent with $\hat{\Psi}$ (as given by γ) and abstracts the resulting set of execution sequences with α, in general losing information (i.e., weakening the property) in the process.

However, this function cannot be *implemented* in these three stages. It is not possible for a program-analysis algorithm to manipulate the probably infinite sets of states or state sequences. Instead, a program analysis performs this three-stage operation in a monolithic fashion, where α and γ are "baked into" the transition relation $\longmapsto$ that forms the core of $\mathcal{S}[\![P]\!]$.

For example, consider again our example meta-rule for variable-assignment transitions:

$$(x := e;\ t, \sigma) \longmapsto (t, \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma])$$

The program analysis designer will hand-design an algorithm that "abstractly" performs these transitions. For instance, if $\widehat{\mathsf{State}}$ is the set of tables of sign environments indexed by control point, as given in Section 1.2, then a straightforward algorithm to compute $(\alpha \circ \mathcal{S}[\![P]\!] \circ \gamma)$ will be hard-wired to propagate the sign property of expression e at control point $(x := e;\ t)$ to variable x at control point t for each variable assignment in P. This makes intuitive sense—the algorithm is "abstractly interpreting" the variable assignments. But of course the analysis designer should justify these intuitions by proving that the algorithm actually implements this function.

Note that:

1. To apply an existing analysis to a different language, one must separately hand-design a new algorithm for the meta-rules of that language. This is an engineering disadvantage.
2. Because the abstraction is "baked into" the analysis algorithm, there is no way to perform multiple execution steps abstracting the result. This is a more serious disadvantage because, as we have explained, it can have devastating effects on the quality of the analysis.

We now consider a different methodology to address these issues. Consider the meta-rule shown above as the single-step transfer relation

$$\overset{(x:=e;\,t),t}{\longmapsto} = \{(\sigma, \sigma') \mid \sigma' = \sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]\}.$$

Imagine a universal computer-representable language of these single-step transfer relations; for instance the above relation might be written as

$$\boxed{x \mapsto e}.$$

Then, given some analysis task such as sign analysis or shape analysis, one could implement a universal "back-end" that analyzes this language of transfer relations. Thus, to apply the analysis to a particular programming language, one merely expresses its semantics in terms of this language of single-step transfer relations instead of the usual meta-rule formulation.

Imagine further that this computer-representable language of transfer relations is closed under composition. For instance, the two successive variable assignments

$$\begin{aligned} \overset{(y{:=}x{-}3;\ x{:=}y{+}5;\ t),(x{:=}y{+}5;\ t)}{\longmapsto} &= \boxed{y \mapsto x - 3} \\ \overset{(x{:=}y{+}5;\ t),t}{\longmapsto} &= \boxed{x \mapsto y + 5} \end{aligned}$$

might be symbolically composed as follows

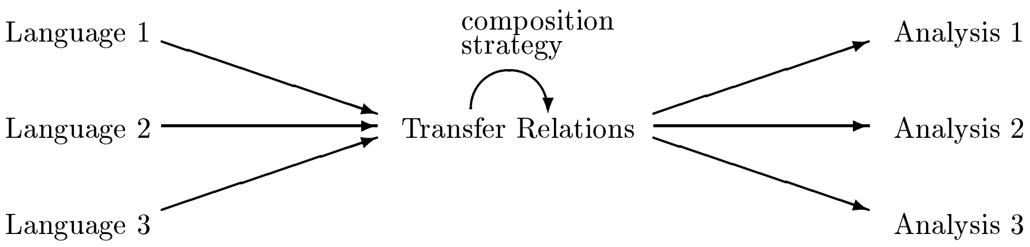
$$\boxed{y \mapsto x - 3};\boxed{x \mapsto y + 5} = \boxed{x, y \mapsto x - 3, x + 2}$$

to yield a computer representation of this two-step execution segment. Then, because the analysis back-end is designed to analyze *any* member of the language of transfer relations, it has maximum flexibility to perform any number of steps *before* abstracting. We demonstrated the benefits of the above example in Section 1.1; there, we magically rewrote the source program, but now we are moving toward a universal language-independent methodology of transfer relations.

Of course, the example immediately above is quite simple, as it does not involve important language features such as arrays, pointers, mutable data structures, or conditionals. The following question remains. Is there a computer-representable language of transfer relations closed under composition that is both

- expressive enough to handle a wide variety of imperative and applicative language features, and
- simple enough to be the target of a wide variety of important program analyses, such as alias, shape, and value analyses?

The answer is yes, and this language of transfer relations is largely the subject of Part II. This leads to the following general methodology of program analysis.



Given a language and an analysis task, one first describes the semantics of the language in terms of single-step transfer relations. Then, guided by a strategy to suit the analysis task and particular program at hand, some of these transfer relations are composed into bigger steps, similar to our rewriting of the example program in Section 1.1. Finally, the particular analysis problem uses these multi-step transfer relations in a manner appropriate to the task. In some cases, such as the sign analysis of Section 1.1 it is appropriate to apply an abstract interpretation to compute an abstract fixed point of these transfer relations. In other cases, such as the analysis of shape-preserving properties of data-structure maintenance routines, it may be more appropriate to extract the property of interest directly from the multi-step transfer relations, without designing any abstraction or performing any fixed-point computation.

Note that the compositions occur at the language-independent stage of transfer relations, so although they are sometimes analogous to rewriting source-program instructions, as in Section 1.1, that is not always the case. Also note that the analyses are now defined in terms of transfer relations instead of source programs. This means that large parts of an analysis do not have to be reimplemented for different languages. In general, however, reengineering is necessary because the language transfer relations will be parameterized by a set of primitive operations, and those may change from source language to source language.

1.4 Overview of the Dissertation

- Part II presents the language of transfer relations and the basic algorithms to compose them and manipulate them, and explains the general procedure for modeling the dynamic semantics of a programming language with transfer relations.

- Part III shows how to model a variety of imperative and applicative language features with transfer relations.

- Part IV expands on Section 1.1 and Section 1.2 by sketching some ideas for how to design program analyses around transfer relations.

- Part V concludes.

Part II

Foundations

Chapter 2

Stores and Transfer Relations

The foundation of our study is the *store*. A store is a model of an instantaneous state of the memory during program execution. As a program executes, it will at various points examine variables, data structures, stack frames, and so on, and it will at other points change the values of variables, alter the components of data structures, allocate new data structures, create new stack frames, and so forth. All of these operations are modeled as examinations or alterations of the store. Intuitively, there is one global store that evolves during program execution. But semantically, this "global store" is modeled as a *trace* of stores. Every time the program takes another step, another store is added to the sequence. If that execution step modified the store, then the modification will be reflected in the latest store. Otherwise, the latest store will just be a copy of the previous one. In this way, the program leaves a trace of stores.

Now consider the task of analyzing a program's execution. Ideally, one would actually let the program run, leaving its trace of stores behind. Then, when the program is done, one could go back to that trace and analyze everything that happened during that execution. The trace of stores is the entire execution history, and with perfect knowledge of that history all questions about the program's run-time behavior could be answered. This is sometimes called profiling.

This approach to program analysis has some serious problems.

- The execution may not terminate, thus leaving behind an infinite trace of stores. So it is impossible in general to run a program and then perform a post-mortem analysis on its trace.

- If the initial store (initial data, values of free variables, and so on) is unknown, then it doesn't make sense to analyze the execution trace of just one execution. One would have to analyze one execution from all possible initial stores, and in general there are an infinite number of them.

- Even if the initial store is fixed and the program terminates, the execution may have a large number of steps, and it is not feasible to record the entire store at every step.

Program analysis is largely the study of how to cope with these issues. The usual approach begins with the observation that if there were an efficient way to represent in a computer some interesting but infinite sets of stores, then some interesting questions about a program's run-time behavior could be answered, or at least approximated, automatically. These representations of infinite store sets can be thought of as *store properties*, and program analysis thus becomes the computation of properties of the stores that can arise during some execution from a store satisfying some initial property. For instance, in Section 1.1 we gave an example of an analysis that determines a sign property of integer-valued variables at every syntactic point in the program.

Our approach is to begin not by examining the stores themselves, but *how stores change over the course of the execution trace.* Suppose that a program analyzer were omnipotent and could examine and answer any questions about the execution traces, even infinite ones, from all possible initial stores. One question of interest might involve examining *pairs* of stores at different points along the trace, to see what the differences are between the first and the second. This would provide information about what happened during the interval of execution between those points. Now reconsider the problems listed above:

- The execution may never terminate and thus leave behind an infinite trace. But even so, there may be an infinite number of finite *intervals* during the execution that exhibit the same pattern of how the store at the beginning of the interval relates to the store at the end. In fact, this is the case with a loop in the program; each interval corresponds to a single iteration. If this pattern can be isolated, then it is not necessary to examine the entire infinite trace. An example of such a pattern is a loop invariant. But this general concept goes beyond loop invariants. For instance, one may relate the store at any point during a loop or recursion with the store k iterations later for a given k.

- Even if the initial store is unknown, there may be a commonality in the *change* between any initial store and the store at some later point in the trace. This is similar to the situation with loops; a potentially unbounded number of trace intervals share a common net effect between their initial and final stores. Related to this idea is the use of weakest preconditions to describe the semantics of loops [Dij76, Wan77].

- As a practical matter, even if the initial store is fixed and the program terminates, isolating the patterns in the trace provides a hope of making the analysis feasible in practice.

Such a pattern or commonality in the way one store evolves into another is simply a *relation between the initial and final stores.* We call these *transfer relations.* It turns out that there is a simple *language* of transfer relations that covers all the patterns that arise during program executions. Also, there are ways to compute these transfer relations and use them to reason about the executions. In this chapter, we introduce the our model of stores and give the language of transfer relations.

2.1 Stores

We make the fundamental assumption that during program execution, any instantaneous state of the memory can be modeled by a *store*. A store is parameterized by the following disjoint sets.

$$\begin{array}{rcll} x & \in & \mathsf{Var} & \text{a set of variables} \\ v & \in & \mathsf{Val} & \text{a set of values} \end{array}$$

A store is then a function from *l-values* to values.

$$\begin{array}{rcrcll} \sigma & \in & \mathsf{Store} & = & \mathsf{Lval} \rightarrow \mathsf{Val} & \text{stores} \\ w & \in & \mathsf{Lval} & = & \mathsf{Var} \cup (\mathsf{Val} \times \mathsf{Val}) & \text{l-values} \end{array}$$

We parameterize a store by Val because we would like to develop a semantic framework that is suitable for a wide variety of programming languages and analysis tasks. However, we will require that Val include the booleans `true` and `false` and a special value `undef`.

$$\mathtt{true}, \mathtt{false}, \mathtt{undef} \quad \in \quad \mathsf{Val}$$

The most natural notion of a store is a partial function, mapping exactly the l-values that are defined to their respective values. But instead, for technical reasons in Chapter 3, we require a store to be a total function, mapping all of the "undefined" l-values to the distinguished value `undef`. Throughout this dissertation, `undef` refers to an undefined or error value. In Chapter 4, we will discuss further the treatment of errors.

The "l" in l-value means "location". Intuitively, an l-value represents a location in memory that might be written or mutated as well as read. There are two kinds of l-values. The first kind is simply a variable. The second kind is called a *reference*; it is a pair of a value $v \in \mathsf{Val}$, representing a data structure, and a value $v' \in \mathsf{Val}$, representing an index into a mutable component of that data structure. The l-value $(v, v') \in \mathsf{Lval}$ is written $v.v'$.

Intuitively, a *value* represents the contents of a single mutable memory location—or in other words, the contents of an l-value. A value might be a simple object such as an integer or a boolean, or it might be a compound object, such as a tuple or vector. In the latter case, however, the compound object must be immutable because it represents the contents of a single mutable memory location. So, for instance, one should not model a (mutable) Scheme [ReC86] cons cell (1 . 2) with a single value, but rather use three values: one for the cons cell itself, one for 1, and one for 2.

A store is then a function from l-values to values that describes the contents of the memory. For instance, if v is the cons cell in the previous paragraph, the store would map the references $v.\mathtt{car}$ and $v.\mathtt{cdr}$ to 1 and 2, respectively. Intuitively, a program execution begins in some initial store σ_0 describing the initial state of memory, input data, and so forth, and then continually modifies the memory while it is executing, producing a sequence of evolving stores $\sigma_1, \sigma_2, \sigma_3, \ldots$ corresponding to the steps of the execution.

We stress again the crucial concept that l-values represent the *mutable* memory locations. Some programming languages include data structures that are not mutable—for instance, the tuples, records, and vectors in Standard ML [MTH90]. One would probably model these objects simply as compound values rather than breaking them up into their components and indexing those components by separate l-values in the store.

Example 1 *Consider the C programming language. A value $v \in \mathsf{Val}$ might correspond to any of the different kinds of C data types:*

- *An integer, real number, or character. In this case, v would be that value.*
- *A pointer. In this case, v would be a token representing the pointer itself. In addition, there would be a value $* \in \mathsf{Val}$, and the l-value $v.*$ would represent the memory location to which the pointer refers. A store would then map $v.*$ to the contents of the pointer.*
- *A struct. In this case, v would be a token (pointer) representing the root of the struct. In addition, there would be a value $f \in \mathsf{Val}$ for each field name f in the structure, and the l-value $v.f$ would represent the memory location of field f of the struct. A store would then map $v.f$ to the contents of that field of the struct.*
- *An array. In this case, v would be a token (pointer) identifying the array. In addition, every non-negative integer n would be in Val, and the l-value $v.n$ would represent the memory location of the nth array element. A store would then map $v.n$ to the contents of the nth element of the array.*

The above example illustrates that for some programming languages, the set Val of values might include, in addition to the base values of the language, a set of pointers to represent mutable data structures. In some operational semantics, these are called "locations" or "heap values" [MFH95] and are just taken from an arbitrary infinite set.

Again, we stress that a store is a *total* function. This is not intuitive, because at any time during an execution of a program in any reasonable programming language, there will only be a finite amount of data actually allocated and accessible by the rest of the execution. But this is why we require that Val include the distinguished value `undef` to represent the undefined value. The intended use of stores is to model the state of memory during an execution of a computer program. If an l-value $w \in \mathsf{Lval}$ is undefined in the memory then the store σ modeling that memory state would map w to `undef` (i.e., $(\sigma\, w) = \texttt{undef}$). Therefore, the fact that we require stores to be total functions is not a limitation of expressiveness. However, for minor technical reasons concerning the symbolic composition of transfer relations in Chapter 3, it will be convenient for stores to be total functions.

Stores as graphs

It is sometimes helpful to think of a store σ as a graph with directed labeled edges. The set of nodes is

$$\mathsf{Val} \cup \{\bullet\}$$

where $\bullet$ is a distinguished root node not in Val. The set of labeled directed edges is

$$\{\, \bullet \xrightarrow{x} v \;\mid \sigma\, x = v\} \cup \{\, v \xrightarrow{v'} v'' \mid \sigma(v.v') = v''\}.$$

Note the following properties of any store graph.

- Because a store is a function rather than a general relation, no two outgoing edges of the same node can have the same label.
- Node $\bullet$ has no incoming edges, and all its outgoing edges are labeled with variables.

At any particular time during program execution, all the l-values that are undefined in the store at that time will point to `undef`. Because of this choice of stores as total functions, a store graph will in general be infinite. We ignore this technical detail, as our perspective of stores as graphs is solely for expository purposes.

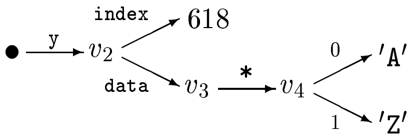
Example 2 *Again, consider the C programming language. Assume the set* Val *includes C integers and C characters.*

- *If at some point during an execution, the variable* `x` $\in \mathsf{Var}$ *is bound to a pointer to a location containing the integer* 42*, then the store* σ *at that point of execution would contain the following path from the root node:*

  $$\bullet \xrightarrow{\texttt{x}} v_1 \xrightarrow{*} 42$$

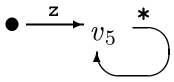
  Here, $v_1 \in \mathsf{Val}$ *represents the pointer itself.*

- *If in addition,* `y` *is bound to a struct with a field* `index`*, which is the integer* 618*, and with a field* `data`*, which points to a two-element array whose elements are the chars* `'A'` *and* `'Z'`*, then* σ *would also contain the following paths from the root node:*

  Here, $v_2 \in \mathsf{Val}$ *represents the struct,* $v_3 \in \mathsf{Val}$ *represents the char-array pointer, and* $v_4 \in \mathsf{Val}$ *represents the char array.*

- *If in addition,* `z` *is bound to a pointer that dereferences to itself, then* σ *would also contain the following subgraph:*

  Here, $v_5 \in \mathsf{Val}$ *represents the pointer itself.*

These three subgraphs of σ describe precisely the data that is reachable from variables `x`*,* `y`*, and* `z`*, respectively, at this point of execution.*

In the next section, we study ways to generate a value from other values in a store. This is done with *primitive operations.*

2.2 Primitive Operations

Our framework is parameterized by a set Primop of *primitive operations.* Each operation $p \in \mathsf{Primop}$ has an arity, which may be zero or more. A primitive operation describes a way in which zero or more values evaluate to a single value. The phrase

$$p(v_1, \ldots, v_n) \hookrightarrow_\sigma v$$

means that the n-ary primitive operation $p \in \mathsf{Primop}$ applied to the values $v_1, \ldots, v_n \in \mathsf{Val}$ in store $\sigma \in \mathsf{Store}$ evaluates to value $v \in \mathsf{Val}$. There are several distinct classes of primitive operations, which we characterize below.

All primitive operations must satisfy the following condition.

Condition 1 (Definedness of primitives) *For any n-ary primitive operation $p \in \mathsf{Primop}$, for any n values $v_1, \ldots, v_n \in \mathsf{Val}$, and for any store $\sigma \in \mathsf{Store}$, there is at least one value $v \in \mathsf{Val}$ such that $p(v_1, \ldots, v_n) \hookrightarrow_\sigma v$. In other words,*

$$\forall p, v_1, \ldots v_n, \sigma. \exists v.\ p(v_1, \ldots, v_n) \hookrightarrow_\sigma v$$

This condition states that primitive operations must be defined everywhere. Conceptually, this requirement is analogous to the requirement that a store is defined everywhere (i.e., for all l-values). The condition is required for minor technical reasons in the development to follow. However, as we explained about stores, this condition does not limit the expressiveness of the framework because Val includes `undef` representing the "undefined value".

Indeed, the two main parameters of our framework—the set Val of values and the set Primop of primitive operations with associated evaluation relation—truly go hand-in-hand. This will come out in Chapter 5 when we describe the design of a programming language using our development.

It is the parameterization of the framework by the set of primitive operations that makes this methodology particularly flexible and useful for a variety of applications. Yet, it is not the case that we are factoring *all* of the important semantics concepts out along with the primitive operations. This is because our concept of a primitive operation is a *computation without store modification.* The encapsulation of such operations as the main parameter of the analysis framework turns out to be quite useful and powerful.

2.2.1 Deterministic and context-independent primitive operations

It will be convenient to introduce some terms for some different classes of primitive operations.

Definition 1 (Deterministic and nondeterministic primitive operations) *A primitive operation $p \in \mathsf{Primop}$ is said to be* deterministic *if*

$$p(v_1, \ldots, v_n) \hookrightarrow_\sigma v$$

and

$$p(v_1, \ldots, v_n) \hookrightarrow_\sigma v'$$

implies that $v = v'$. Otherwise, p is said to be nondeterministic.

A typical programming language will need only deterministic primitive operations, but certain applications of the framework will make use of nondeterministic operations, and so we include them in the general framework.

Definition 2 (Context-independent and -dependent operations) *A primitive operation is said to be* context-independent *if for any stores σ and σ' and values $v_1, \ldots, v_n, v$,*

$$p(v_1, \ldots, v_n) \hookrightarrow_\sigma v \iff p(v_1, \ldots, v_n) \hookrightarrow_{\sigma'} v.$$

In other words, the evaluation of p does not depend on the store. In this case, we may use the abbreviated form

$$p(v_1, \ldots, v_n) \hookrightarrow v$$

for evaluation. Otherwise, p is said to be context-dependent.

The simplest kinds of primitive operations are deterministic context-independent operations. These will come up so often that it is worth introducing a definition just for them.

Definition 3 (Simple primitive operations) *If primitive operation p is both deterministic and context-independent, then it is said to be* simple.

2.2.2 Examples of primitive operations

We present some examples of each kind of primitive operations. Although they are just examples for the moment, some of them will play a major role in the development to follow. This section assumes that

$$p(v_1, \ldots, v_n) \hookrightarrow_\sigma \texttt{undef}$$

unless otherwise defined below. (Recall that `undef` $\in \mathsf{Val}$ is represents the undefined value.) We also assume that Val includes the integers.

Simple operations

Recall that simple operations are operations that are both deterministic, in that they evaluate to only a single value, and context-independent, in that their evaluation does not depend on the store.

Example 3 *Each value $v \in \mathsf{Val}$ specifies a nullary primitive operation $v \in \mathsf{Primop}$ that evaluates to v:*

$$v() \hookrightarrow v$$

Example 4 *Here are some standard arithmetic primitive operations found in programming languages. In these definitions, n and n' are integer values.*

$$\begin{aligned} \texttt{+}(n, n') &\hookrightarrow (n + n') \\ \texttt{-}(n, n') &\hookrightarrow (n - n') \\ \texttt{*}(n, n') &\hookrightarrow (n \times n') \\ \texttt{<}(n, n') &\hookrightarrow (n < n') \\ \texttt{>}(n, n') &\hookrightarrow (n > n') \end{aligned}$$

Example 5 *Below are boolean operations for conjunction, equality, and inequality. We will use the first two (`&` and `=`) internally in our analysis framework.*

$$\begin{aligned} \texttt{\&}(\texttt{true}, v) &\hookrightarrow v \\ \texttt{\&}(v, \texttt{true}) &\hookrightarrow v \\ \texttt{\&}(\texttt{false}, v) &\hookrightarrow \texttt{false} \\ \texttt{\&}(v, \texttt{false}) &\hookrightarrow \texttt{false} \end{aligned}$$

$$\begin{aligned} \texttt{=}(v, v') &\hookrightarrow (v = v') \\ \texttt{<>}(v, v') &\hookrightarrow (v \neq v') \end{aligned}$$

Note that there are cases in which these operations are given `undef` and evaluate to a boolean value. Intuitively, these are error cases that are allowed to "run wild", and so this choice is reasonable. We will discuss this more in Chapter 4.

Example 6 *The operation `if` implements conditional expressions.*

$$\begin{aligned} \texttt{if}(\texttt{true}, v, v') &\hookrightarrow v \\ \texttt{if}(\texttt{false}, v, v') &\hookrightarrow v' \end{aligned}$$

Example 7 *Example 1 demonstrated the need for pointer values to correspond to the roots of mutable data structures. Suppose that for every natural number n there is a pointer $\langle n \rangle \in \mathsf{Val}$, and `ptr` is a unary primitive operation that casts an integer n to the pointer $\langle n \rangle$.*

$$\texttt{ptr}(n) \hookrightarrow \langle n \rangle$$

We will return to `ptr` in Chapter 5.

Nondeterministic context-independent operations

These operations may evaluate to more than one value, but do not depend upon the store. These operations are similar to types, and this similarity provides some intuition about why we include the possibility of nondeterministic operations. After all, one of the major applications of program analysis is to infer types of data objects and expressions.

Example 8 *The nullary operation* `pos` *may evaluate to any positive integer:*

$$\texttt{pos}() \hookrightarrow n \qquad \textit{if } n \in \{1, 2, \ldots\}$$

Example 9 *The nullary operation* `bool` *may evaluate to any boolean:*

$$\begin{array}{l}\texttt{bool}() \hookrightarrow \texttt{true}\\ \texttt{bool}() \hookrightarrow \texttt{false}\end{array}$$

Deterministic context-sensitive operations

These operations always evaluate to a single value, but depend on the store in which the evaluation occurs.

Example 10 *The binary operation* `deref` *dereferences edges in a store. Given values v and v', it evaluates in store σ to the value to which σ binds the l-value $(v.v')$:*

$$\texttt{deref}(v, v') \hookrightarrow_{\sigma} \sigma(v.v')$$

This last example is important, which we will see in the next section.

The next example hints at an application of our framework to the analysis of the shapes of data structures, and also illustrates the point that our notion of primitive operations does not need to be limited to operations that might be available in a programming language.

Example 11 *The unary operation* `tree`*, when evaluated in store σ with value v, evaluates to* `true` *if the subgraph of σ rooted at v (possibly representing the root of some data structure) and not including node* `undef` *is a tree (in graph-theoretic terms). Otherwise, it evaluates to* `false`*.*

Although Primop is a parameter of our analysis framework, we demand that it include the following operations described above:

$$\texttt{true}, \texttt{false}, \texttt{\&}, \texttt{=}, \texttt{if} \quad \in \quad \mathsf{Primop}$$

The first two simply provide a way of denoting booleans as expressions. (Recall that the booleans were the only objects other than `undef` that we demand to be members of Val.) The second three are used internally in the transfer-relation composition algorithm in Chapter 3. We further remark that any nontrivial application of our methodology will need `deref` in order to build expressions that can perform general examinations of the store.

2.3 Expressions and L-expressions

Our framework is based upon the study of binary relations between stores, called transfer relations, that describe how a store at one point of an execution evolves into a store at some future point of the execution. We want to write down these transfer relations, represent them in a computer, and analyze them with algorithms. In order to do this, it turns out that we will need two languages of computer-representable terms, one to describe the nodes in a store and one to describe the edges in a store. Later, we will use these two languages to develop a language of transfer relations.

Elements of the first language are called *expressions*; given a store $\sigma \in \mathsf{Store}$, an expression $e \in \mathsf{Exp}$ denotes one or more values $v \in \mathsf{Val}$. If e denotes v in σ, then we say that "*e evaluates to v in σ*". The same expression may evaluate to different values in different stores.

Elements of the second language are called *l-expressions*; given a store $\sigma \in \mathsf{Store}$, the l-expression $l \in \mathsf{Lexp}$ denotes one or more l-values $w \in \mathsf{Lval}$. If l denotes w in σ, then we say that "*l evaluates to w in σ*". The same l-expression may evaluate to different l-values in different stores.

The syntax of the language of expressions and l-expressions is parameterized by a set Primop of primitive operations, described in Section 2.2, and has the following inductive definition.

$$\begin{array}{lllll} e & \in & \mathsf{Exp} & ::= & x \mid p(e_1, \ldots, e_n) & \text{expressions} \\ l & \in & \mathsf{Lexp} & ::= & x \mid e.e' & \text{l-expressions} \\ \\ p & \in & \mathsf{Primop} & & & \text{primitive operations (given)} \\ x & \in & \mathsf{Var} & & & \text{variables (given)} \end{array}$$

There are two types of expressions:

- A variable $x \in \mathsf{Var}$. This expression evaluates in store $\sigma \in \mathsf{Store}$ to the (unique) value $v \in \mathsf{Val}$ such that $\bullet \xrightarrow{x} v$ is an edge in σ. In other words, x evaluates in σ to $(\sigma\, x)$.
- An application of an n-ary primitive operation $p \in \mathsf{Primop}$ to n expressions $e_1, \ldots, e_n \in \mathsf{Exp}$. This expression evaluates in store $\sigma \in \mathsf{Store}$ to value $v \in \mathsf{Val}$ if expression e_i evaluates in store σ to value v_i, for $i \in \{1, \ldots, n\}$, and p applied to $(v_1, \ldots, v_n)$ evaluates in store σ to v.

The phrase

$$p$$

denotes the nullary primitive application $p()$. The phrase

$$e\, p\, e'$$

denotes the binary primitive application $p(e, e')$.

There are two types of l-expressions:

- A variable $x \in \mathsf{Var}$. This l-expression evaluates to itself in any store, and can be thought of informally as the dangling edge $\bullet \xrightarrow{x}$.
- A reference expression $e.e'$. This l-expression evaluates in store $\sigma \in \mathsf{Store}$ to the l-value $v.v' \in \mathsf{Lval}$ if e evaluates in σ to value $v \in \mathsf{Val}$ and e' evaluates in store σ to value $v' \in \mathsf{Val}$. This l-value can be thought of informally as the dangling edge $v \xrightarrow{v'}$.

Formally, the interpretations of expressions and l-expressions are given by the following relations.

- The phrase $e \vdash_\sigma v$ means that the expression e evaluates in store σ to value v.
- The phrase $l \vdash_\sigma w$ means that the l-expression l evaluates in store σ to l-value w.

Because a variable is both an expression and an l-expression, this notation may seem ambiguous. But in the former case, the right-hand side will be a value, and in the latter case it will be an l-value.

The following rules inductively define these relations.

$$x \vdash_\sigma (\sigma\, x) \qquad \frac{e_i \vdash_\sigma v_i \quad p(v_1, \ldots, v_n) \hookrightarrow_\sigma v}{p(e_1, \ldots, e_n) \vdash_\sigma v} \qquad \text{expression evaluation}$$

$$x \vdash_\sigma x \qquad \frac{e \vdash_\sigma v \quad e' \vdash_\sigma v'}{(e.e') \vdash_\sigma (v.v')} \qquad \text{l-expression evaluation}$$

The following lemma states that every expression (l-expression) evaluates to at least one value (l-value).

Lemma 1 (Definedness of expressions and l-expressions) *For any expression $e \in \mathsf{Exp}$, for any store $\sigma \in \mathsf{Store}$, there is at least one value $v \in \mathsf{Val}$ such that $e \vdash_\sigma v$. For any l-expression $l \in \mathsf{Lexp}$, for any store $\sigma \in \mathsf{Store}$, there is at least one l-value $w \in \mathsf{Lval}$ such that $l \vdash_\sigma w$. In other words:*

- $\forall e \in \mathsf{Exp}, \sigma \in \mathsf{Store}. \exists v \in \mathsf{Val}.\ e \vdash_\sigma v$
- $\forall l \in \mathsf{Lexp}, \sigma \in \mathsf{Store}. \exists w \in \mathsf{Lval}.\ l \vdash_\sigma w$

Proof: From Condition 1 on primitive operations and by straightforward induction on expression and l-expression evaluation. □

It is important to distinguish expressions and l-expressions that do not contain any applications of nondeterministic primitive operations.

Definition 4 (Deterministic expressions and l-expressions) *For expression $e \in \mathsf{Exp}$ (respectively, l-expression $l \in \mathsf{Lexp}$), if neither e nor any subexpression of e (respectively, if no subexpression of l) is an application of a nondeterministic primitive operation, then we say that e (respectively, l) is* deterministic*. The phrase* $\mathrm{determ}(e)$ *(respectively,* $\mathrm{determ}(l)$*) denotes this fact.*

The following lemma states that deterministic expressions and l-expressions always evaluate to *exactly* one value and l-value, respectively.

Lemma 2 (Deterministic expressions and l-expressions) *For any deterministic expression $e \in \mathsf{Exp}$, for any store $\sigma \in \mathsf{Store}$, there is exactly one value $v \in \mathsf{Val}$ such that $e \vdash_\sigma v$. For any deterministic l-expression $l \in \mathsf{Lexp}$, for any store $\sigma \in \mathsf{Store}$, there is exactly one l-value $w \in \mathsf{Lval}$ such that $l \vdash_\sigma w$. In other words,*

- $\mathrm{determ}(e) \Rightarrow \forall \sigma \in \mathsf{Store}.\, \exists! v \in \mathsf{Val}.\ e \vdash_\sigma v$
- $\mathrm{determ}(l) \Rightarrow \forall \sigma \in \mathsf{Store}.\, \exists! w \in \mathsf{Lval}.\ l \vdash_\sigma w$

Proof: From Lemma 1, from Definition 1, and from straightforward induction on expression and l-expression evaluation. □

If all primitive operations in Primop are deterministic, then all expressions are deterministic. But even if there are nondeterministic primitive operations in Primop, it will be important to distinguish deterministic expressions for the symbolic evaluation of certain primitive operations in Chapter 3.

At first it may seem as if our language of expressions is too restrictive; why not allow arbitrary l-expressions instead of just variables. The reason is that one may treat the l-expression $e.e'$ as an expression by using the `deref` primitive operation that we introduced in the previous section. Consider the C expression `*x`. On the left-hand side of an assignment statement, `*x` refers to a memory location, or an l-value in our terms. But on the right-hand side, it refers to the contents of that memory location, or a value in our terms. But C has a uniform syntax to handle both cases; they are both expressions. In contrast, in our framework the term on the left-hand side would be an l-expression—namely, `x.*` (where `*` $\in \mathsf{Val}$ as in Example 1)—whereas the term on the right-hand side would be an expression—namely, the primitive application `deref(x,*)`.

Therefore, `deref` is a rather distinguished primitive operation in that it provides the ability to examine the store beyond the level of variables. It is likely that one will almost certainly need it in *any* analysis application for any language. Therefore, inspired by the above discussion, we introduce a special syntax for it. The term

$$e.e'$$

will, depending on the context in which it appears, refer to either the l-expression $e.e'$ or the primitive-application expression $\texttt{deref}(e, e')$.

2.4 Simple Transfer Relations

Our central philosophy is that it is advantageous to analyze *relations* between two stores. These relations are called *transfer relations*. The idea of studying transfer relations is a paradigm shift from most analysis frameworks, as the focus is usually on reasoning about properties of, or sets of, individual stores. Yet a program fragment relates initial stores to final stores, and so it seems intuitive to study these relations.

2.4.1 Only some relations are natural

At first blush, it seems as if the set of transfer relations is the set

$$\mathcal{P}(\mathsf{Store} \times \mathsf{Store})$$

of binary relations between stores. But such an unrestricted notion of transfer relation has two related disadvantages.

- For analysis purposes, we will want to design computer representations of transfer relations (ideally, concise representations), and it is impossible to do so for general binary relations between stores.
- Earlier, we gave the intuition that for the purpose of static program analysis, a transfer relation corresponds to a fragment of the execution of some program in some programming language. But there are many binary relations between stores that would never come from any such execution fragment. Indeed, there are many such relations that are not even computable. These relations are unnatural in that they do not arise during program execution, and they are thus of no use for reasoning about programs.

The key is to identify the kinds of transfer relations that might actually come about as part of a computer program's execution. Fortunately, there is a class of such relations that is sufficiently expressive and yet conducive to automatic reasoning and analysis. We will demonstrate this in later chapters, where we model real programming languages with transfer relations and then use those relations to analyze source programs.

Abstractly, apart from any particular language or program, one basic kind of transfer relation is a relation that updates a store graph by assigning or changing the node to which an edge in the graph points. These relations can describe dynamic actions in a programming language that modify memory in some way. We already have both the language of expressions to denote nodes and the language of l-expressions to denote edges. An assignment relation is then described by an l-expression, denoting an edge to be assigned or reassigned, and an expression, denoting a node to which the edge must point.

Another basic kind of transfer relation is a relation that simply filters through stores that satisfy a certain property and rejects the stores that do not. This is the most basic kind of conditional operation, and as such will be necessary to express the dynamic behavior of most programming languages. Again, we can use the language of expressions to specify these properties, and so a filter relation is described by an expression.

One can then build bigger relations from these basic relations.

2.4.2 Building natural transfer relations

We wish to describe a set

$$\mathsf{TrRel} \subset \mathcal{P}(\mathsf{Store} \times \mathsf{Store})$$

of *natural* transfer relations. By natural, we mean informally that it is reasonable to imagine that the transfer relation in question might correspond to a fragment of an execution of some program in some programming language. In other words, suppose that at a certain point in the middle of an execution of some program in some language, store σ_1 describes the state of the memory at that point. As the execution continues from that point, it will produce a sequence of evolving stores $\sigma_2, \sigma_3, \ldots$ corresponding to the steps of the execution. Then for each $n \geq 0$, there there should be some transfer relation $\Delta \in \mathsf{TrRel}$ that relates store σ_1 to store σ_n; in other words,

$$\sigma_1 \, \Delta \, \sigma_n.$$

The idea is to keep the set TrRel as small as possible, but still large enough that one could model the operational semantics of realistic programming languages using only these relations. Fortunately, this is quite easy to do in a rather satisfactory and intuitive manner.

We will define the set TrRel inductively.

- There are four types of basic relations in TrRel:
  - The *empty relation* $\emptyset$.
  - The *identity relation* $\bullet$, which relates only between identical stores. Formally:

    $$\sigma \bullet \sigma$$

  - The *assignment relation* $\boxed{l \mapsto e}$ where $l \in \mathsf{Lexp}$ and $e \in \mathsf{Exp}$. If l-expression l evaluates to l-value w in σ and expression e evaluates to value v in σ, then $\boxed{l \mapsto e}$ updates store σ by assigning w to v. Formally:

    $$\frac{l \vdash_\sigma w \quad e \vdash_\sigma v}{\sigma \boxed{l \mapsto e} (\sigma[w \mapsto v])}$$

    Here, $\sigma[w \mapsto v]$ is the store that maps w to v and is otherwise identical to σ. Formally, it is defined as follows:

    $$(\sigma[w \mapsto v])\, w' = \begin{cases} v & \text{if } w = w' \\ \sigma\, w' & \text{otherwise} \end{cases}$$

    Note that if $\neg\mathrm{determ}(l)$ (i.e., if a nondeterministic primitive operation appears in l) then l may evaluate to more than one l-value (and similarly for e), and so $\boxed{l \mapsto e}$ can relate a store on the left to several different stores on the right.

– The *filter relation* $\boxed{e?}$ where $e \in \mathsf{Exp}$, which relates a store σ to itself only if expression e evaluates to the value `true` in σ. Formally:

$$\frac{e \vdash_\sigma \texttt{true}}{\sigma\,\boxed{e?}\,\sigma}$$

- If $\Delta, \Delta' \in \mathsf{TrRel}$, then their relational composition $\Delta; \Delta'$ (alternatively, $\Delta' \circ \Delta$) is in TrRel. We use the standard definition of relational composition:

$$\frac{\sigma\,\Delta\,\sigma' \quad \sigma'\,\Delta'\,\sigma''}{\sigma\,(\Delta; \Delta')\,\sigma'}$$

Note that the identity relation $\bullet$ can be defined as the filter relation $\boxed{\texttt{true}?}$, where, as described in Section 2.2.2, `true` denotes the nullary application of the primitive operation defined by $\texttt{true} \in \mathsf{Val}$. Similarly, the empty relation $\emptyset$ can be defined as the filter relation $\boxed{\texttt{false}?}$. But it is more convenient to have distinguished representations for each of these two special cases.

2.4.3 Examples of transfer relations

As described at the beginning of this chapter, our goal for the sake of generality is to develop a framework for expressing data and operations on data in a language-independent manner. Nevertheless, it is illuminating at this point to look at some examples of transfer relations and consider how they might arise during the execution of a computer program.

Example 12 *The transfer relation*

$$\boxed{\texttt{x} \mapsto 2}; \boxed{\texttt{x} \mapsto \texttt{x} + 1}$$

is equal to (in other words, precisely the same relation as) the transfer relation

$$\boxed{\texttt{x} \mapsto 3}$$

that assigns variable `x` *to be* 3*. In other words, it changes any store by redirecting the edge*

$$\bullet \xrightarrow{\texttt{x}} v$$

to

$$\bullet \xrightarrow{\texttt{x}} 3$$

Here, $+ \in \mathsf{Primop}$*, and the integers are included in* Val *and as nullary operations in* Primop.

Example 13 *The transfer relation*

$$\boxed{(\texttt{x} < 0)?}; \boxed{\texttt{x} \mapsto 0 - \texttt{x}}$$

relates any store in which x is bound to a negative number to a store in which `x` *to be the absolute value of that number and is otherwise equivalent. Furthermore, it relates every store in which* `x` *is not bound to a negative number to no store at all.*

Example 14 *Imagine a use of stores and transfer relations to model a language with heap-allocated data structures. One way to model data allocation is to maintain a convention that the semantic variable* `H` *holds the index of the next available pointer. Then one can use* `ptr` *from Example 7 to generate that pointer. By convention, we write*

$$\mathtt{ptr}(e)$$

as

$$\langle e \rangle$$

Then the transfer relation

$$\boxed{\mathtt{x} \mapsto \langle \mathtt{H} \rangle}\,;\boxed{\mathtt{x.car} \mapsto \mathtt{y}}\,;\boxed{\mathtt{x.cdr} \mapsto \mathtt{z}}\,;\boxed{\mathtt{H} \mapsto \mathtt{H + 1}}$$

allocates a new record that has two fields—`car`*, which is assigned to y's value (which may be* `undef`*), and* `cdr`*, which is assigned to z's value (which may be* `undef`*)—and assigns x to be this record.*

Example 15 *The transfer relation*

$$\boxed{\mathtt{x} \mapsto \mathtt{x.tl}}\,;\boxed{\mathtt{x} \mapsto \mathtt{x.tl}}\,;\boxed{\mathtt{x} \mapsto \mathtt{x.tl}}$$

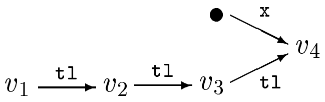
is equal to the transfer relation

$$\boxed{\mathtt{x} \mapsto \mathtt{x.tl.tl.tl}}$$

which in a store that includes the subgraph

$$\bullet \xrightarrow{\mathtt{x}} v_1 \xrightarrow{\mathtt{tl}} v_2 \xrightarrow{\mathtt{tl}} v_3 \xrightarrow{\mathtt{tl}} v_4$$

assigns `x` *to be v_4, producing a store that includes the subgraph*



Although we have written the paths as linear pictures, it is not necessarily the case that v_1, v_2, v_3, and v_4 are distinct. Therefore, the paths shown above may actually include cycles. For instance, if $v_1 = v_3$ then the original subgraph would actually look like

$$\bullet \xrightarrow{\mathtt{x}} v_1 \underset{\mathtt{tl}}{\overset{\mathtt{tl}}{\rightleftarrows}} v_2$$

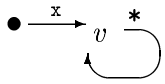
and the transfer relation would modify this to

$$v_1 \underset{\mathtt{tl}}{\overset{\mathtt{tl}}{\rightleftarrows}} v_2 \xleftarrow{\mathtt{x}} \bullet$$

Example 16 *The transfer relation*

$$\boxed{\mathtt{x.*} \mapsto \mathtt{x}}$$

assigns field * *of the value bound to* x *to point to that value itself, thus creating a circular data structure in the store:*



The C statement

```
*x = x;
```

performs a performs a similar operation. Before the statement, x *is bound to a memory address* v*. After the statement,* x *is still bound to the memory address* v*, but now that memory address holds* v *itself. Our graphical representation above directly reflects this memory state.*

Example 17 *The transfer relation*

$$\boxed{\texttt{x.y} \mapsto \texttt{z}}$$

transforms a store σ *as follows. Suppose* x *is bound to value* v*,* y *is bound to value* v'*, and* z *is bound to value* v'' *in* σ*. Then the outgoing edge of* v *labeled with* v' *is redirected to point to* v''*. If* v' *is an integer then this is equivalent to the C statement*

```
x[y] = z;
```

but if v' is not an integer then this transfer relation has no correspondence in C.

Example 18 *The transfer relation*

$$\boxed{\texttt{x.car} \mapsto \texttt{y}};\boxed{\texttt{z.car} \mapsto \texttt{w}}$$

acts as follows. For those stores in which x *and* z *are bound to different values, it assigns field* car *of* x*'s value to be* y*'s value and field* car *of* z*'s value to be* w*'s value. For those stores in which* x *and* z *are bound to the same value, it assigns field* car *of that value to be* w*'s value.*

2.5 The Difficulty of Composition

Above, we defined a basic relation to be either the empty relation $\emptyset$, the identity relation $\bullet$, an assignment relation $\boxed{l \mapsto e}$, or a filter relation $\boxed{e?}$. We defined any transfer relation that is not a basic relation to be a finite composition of basic relations. Note that in Examples 12 and 15, the composition of more than one basic relation is equal to another basic transfer relation, but in those cases the single basic transfer relation such as

$$\boxed{x \mapsto 3}$$

exposes information that is not so clear in the composition itself.

Sometimes, though, the composition of more than one basic relation is *not* a basic relation, as Examples 13, 14, and 18 demonstrate.

It would be convenient if there were a reasonably compact and clear representation scheme for *all* transfer relations. The explicit composition of 100 basic relations is not only cumbersome by any reasonable measure, but also quite likely shed little insight on what exactly the transfer relation does. For instance, the description of the transfer relation

$$\boxed{x.\mathtt{car} \mapsto y} ; \boxed{z.\mathtt{car} \mapsto w}$$

in Example 18 is not at all obvious from that representation itself. The exercise of decoding the net effect of the transfer relation

$$\boxed{x.\mathtt{tl}.\mathtt{tl} \mapsto y.\mathtt{tl}} ; \boxed{y.\mathtt{tl} \mapsto x.\mathtt{tl}} ; \boxed{x.\mathtt{tl} \mapsto y}$$

is much more difficult yet. It may seem as if this last example is designed to destructively insert the first element of a linked-list y into the second position of a linked-list x. However, that behavior only occurs under certain initial aliasing conditions. It is not an easy exercise to determine the possible behaviors of this example under different initial aliasing conditions.

Fortunately, there is a reasonably simple representation scheme that covers all transfer relations in TrRel. This scheme actually computes the effect of any composition, rather than leaving the composition operation explicit as written directly above, and hence reveals quite clearly the effect of any transfer relation. But the four kinds of basic relations in TrRel are not quite sufficient to express these compositions syntactically. Therefore, we have to extend the language.

2.6 The Full Language of Transfer Relations

The language TR of transfer relations is defined inductively as follows.

$$\begin{array}{rcll} \Delta & \in & \mathsf{TR} & ::= \emptyset \mid \delta \mid \boxed{e?\ \Delta \,\big|\, \Delta'} \\ \delta & \in & \mathsf{ATR} & ::= \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n} \end{array}$$

We have already defined $\emptyset$ (the empty relation). Assignment relations are generalized to *parallel assignments* $\mathsf{ATR} \subset \mathsf{TR}$, defined as follows.

$$\frac{l_i \vdash_\sigma w_i \quad e_i \vdash_\sigma v_i \quad i \neq j \Rightarrow w_i \neq w_j}{\sigma \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n} (\sigma[w_1 \mapsto v_1] \ldots [w_n \mapsto v_n])}$$

A crucial fact about assignment relations is that the assignment only takes place if all the l-values to be assigned are actually distinct. One can therefore look at an assignment relation with n assignments and know that whenever it relates an initial store to a final store, it performs exactly n distinct assignments.

Filter relations are generalized to *conditional relations*, defined as follows.

$$\frac{e \vdash_\sigma \mathtt{true} \quad \sigma\, \Delta\, \sigma'}{\sigma \boxed{e?\ \Delta \,\big|\, \Delta'} \sigma'} \qquad \frac{e \vdash_\sigma \mathtt{false} \quad \sigma\, \Delta'\, \sigma'}{\sigma \boxed{e?\ \Delta \,\big|\, \Delta'} \sigma'}$$

We adopt the following syntactic abbreviations.

- The empty parallel assignment (i.e., where $n = 0$) is simply the identity relation $\bullet$ and may be written as such.
- The conditional relation $\boxed{e?\ \Delta \,|\, \emptyset}$ may be abbreviated as $\boxed{e?\ \Delta}$.
- The conditional relation $\boxed{e?\ \emptyset \,|\, \Delta}$ may be abbreviated as $\boxed{e¿\ \Delta}$.

In order to avoid confusion, we introduce different notations for syntactic and semantic equivalence of transfer relations. If Δ and Δ' are both the same syntactic term, or in other words the same element of the language TR, then we write $\Delta = \Delta$ and say that they are *syntactically equivalent.* If Δ and Δ' denote the same relation, then we write $\Delta \equiv \Delta'$ and say that they are *semantically equivalent.* Note that syntactic equivalence obviously implies semantic equivalence, but semantic equivalence does not necessarily imply syntactic equivalence because in this language there may be more than one way to write the same relation. For instance,

$$\boxed{\texttt{true}?\ \Delta \,|\, \Delta'} \equiv \Delta,$$

but

$$\boxed{\texttt{true}?\ \Delta \,|\, \Delta'} \neq \Delta.$$

In this sense, the language TR is not fully abstract [HP79, Mul87]. If it were fully abstract, then we would have a decidable way of testing semantic equality of transfer relations, but this will not be so important for the applications of our analysis framework.

A major property of transfer relations is that if all the primitive operations are deterministic, then all transfer relations are actually partial functions. Formally, we have the following lemma.

Lemma 3 (Deterministic transfer relations) *If* Primop *is deterministic, then for every transfer relation* $\Delta \in \mathsf{TR}$ *and store* $\sigma \in \mathsf{Store}$*, there is at most one store* $\sigma' \in \mathsf{Store}$ *such that* $\sigma\, \Delta\, \sigma'$*.*

Proof: Given σ, we proceed by structural induction on Δ.

- $\emptyset$: By definition, there is no σ' such that $\sigma\, \emptyset\, \sigma'$.
- $\boxed{e?\ \Delta \,|\, \Delta'}$: Because Primop is deterministic, we know from Lemma 2 that there exists exactly one v such that $e \vdash_\sigma v$. There are three cases.
  - $v = \texttt{true}$: Then by the definition of conditional relations, $\sigma \boxed{e?\ \Delta \,|\, \Delta'} \sigma'$ only if $\sigma\, \Delta\, \sigma'$, and by induction there is at most one σ'.
  - $v = \texttt{false}$: Analogous, with Δ'.
  - Otherwise: Then by the definition of conditional relations, there is no σ' such that $\sigma \boxed{e?\ \Delta \,|\, \Delta'} \sigma'$.

- $\boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n}$: Because Primop is deterministic, we know from Lemma 2 that for $i \in \{1, \ldots, n\}$, there exists exactly one w_i such that $l_i \vdash_\sigma w_i$ and exactly one v_i such that $e_i \vdash_\sigma v_i$. Then by the definition of assignment relations, $\sigma \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n} \sigma'$ only if $w_1, \ldots, w_n$ are all distinct and $\sigma' = \sigma[w_1 \mapsto v_1] \ldots [w_n \mapsto v_n]$. There is at most one such σ'.

□ This theorem has the following corollary, which is not useful on its own, but which we will use in some of the proofs in Chapter 3.

Corollary 1 *If* Primop *is deterministic, then for any two transfer relations* $\Delta, \Delta' \in \mathsf{TR}$*, the following two statements are equivalent:*

- $\Delta \equiv \Delta'$
- $(\sigma \, \Delta \, \sigma' \Rightarrow \sigma \, \Delta' \, \sigma') \wedge (\sigma \, \Delta' \, \sigma' \Rightarrow \exists \sigma''. \, \sigma \, \Delta \, \sigma'')$

The main result about this language of transfer functions is that under certain conditions it is closed under composition. In other words, there exists a total *syntactic composition* function

$$\oplus \quad \in \quad \mathsf{TR} \times \mathsf{TR} \to \mathsf{TR}$$

that, given two transfer relations in the language TR, builds a third transfer relation in TR that is semantically equivalent to their composition. In other words,

$$(\Delta \oplus \Delta') \equiv (\Delta; \Delta')$$

for any two transfer relations $\Delta, \Delta' \in \mathsf{TR}$. Under weaker circumstances, $\Delta \oplus \Delta'$ is not guaranteed to be semantically equivalent to $\Delta; \Delta'$, but is guaranteed to be a superset of $\Delta; \Delta'$. But we will see that the conditions for semantic equivalence will be met by any application of transfer relations to model the dynamic semantics of programming language.

In fact, the $\oplus$ function is effectively computable, and so we will call it the *composition algorithm*. If there were a combinator in the language of transfer relations that represented composition, then the composition algorithm would be trivial. In other words, if we extend the language by

$$\Delta \quad ::= \quad \ldots \mid \Delta \oplus \Delta'$$

and define $\Delta \oplus \Delta'$ to be the relation $\Delta; \Delta'$, then the composition algorithm could be simply the $\oplus$ combinator. But, as we explained above, our goal for the practical purpose of program analysis is to avoid a syntactic representation of composition. We present the algorithm in the next chapter.

Chapter 3

Composing Transfer Relations

The composition algorithm for transfer relations is based on a kind of symbolic evaluation. We will present the the algorithm in several stages.

- Symbolic evaluation of primitive operations. This part of course depends on the particular choice of the set Primop of primitive operations and their evaluation semantics. The choice of Primop and design of the symbolic evaluation algorithms for those operations forms the core of any program analysis designed with our framework.
- Symbolic evaluation of expressions and l-expressions. These algorithms are defined relative to Primop and its associated symbolic evaluation algorithms.
- Symbolic evaluation of conditional relations. This part is also a parameter to the composition algorithm.
- Symbolic evaluation of assignment.
- Symbolic evaluation of transfer-relation composition.

3.1 Symbolic Evaluation of Primitive Operations

The first step of any application of our analysis methodology is the choice of the set Val of values and the set Primop of primitive operations, which will largely depend on the language to be analyzed.[1] The second step is the design of an algorithm to symbolically evaluate primitive application expressions. The heart of our methodology is in this symbolic evaluation, and the power of our approach comes from the flexible notion of a primitive operation as potentially any computation that does not modify the store, including both non-deterministic primitive operations and context-sensitive primitive operations. Yet, the fact that primitive operations

[1] Recall that we require Val to include the boolean constants and `undef`, and we require Primop to include the boolean constants and the boolean operations `&`, `=`, and `if`.

are constrained not to modify the store ensures that their symbolic evaluation is never too complicated.

The framework constructed in this section is for us what the notion of the Galois connection and associated fixed-point theorem is for abstract interpretation [CC77]. In abstract interpretation, one first designs a fixed-point-based semantics for the language to be analyzed and then designs an abstraction of the semantic domain. Then, for the most part, the framework of abstract interpretation provides the rest—in particular, a functional for the abstract domain induced by the semantics whose fixed point is guaranteed to satisfy a certain relation with the semantics of the language.

In our approach, one first chooses a set of primitive operations and then designs symbolic evaluation algorithms for them. The various algorithms in this chapter provide much of the remaining work.

In Chapter 2, we gave many examples of useful primitive operations. Some of them were common and familiar, such as the constants and basic operations over booleans and integers. Others were rather distinguished, such as `deref` and `pos`. We will return to many of these in this section.

3.1.1 A first cut: symbolic evaluation of simple primitive operations

A term like "symbolic evaluation" would tend to imply that we need an algorithm

$$\mathsf{P} \quad \in \quad \mathsf{Primop} \to \mathsf{Exp}^* \to \mathsf{Exp}$$

that satisfies the property that

$$(\mathsf{P}\, p\, (e_1, \ldots, e_n)) \vdash_\sigma v \iff p(e_1, \ldots, e_n) \vdash_\sigma v.$$

In other words, P, given an n-ary primitive operation p and n expressions $e_1, \ldots, e_n$, returns an expression that is semantically equivalent to the expression $p(e_1, \ldots, e_n)$. The degenerate function

$$\mathsf{P}\, p\, (e_1, \ldots, e_n) = p(e_1, \ldots, e_n)$$

obviously works, but might not produce optimal results. For instance, that function returns

$$\mathsf{P} + (42, 24) = 42 + 24,$$

but it is obvious that in this case P could have actually performed the addition, thus producing the smaller expression

$$\mathsf{P} + (42, 24) = 66$$

where, again, 66 is technically an application of the nullary primitive operation 66. This is called "constant folding" in the compiler literature [ASU86]. Even beyond this, one could imagine that P might try to use a calculus of arithmetic transformations, perhaps yielding results such as

$$\mathsf{P} + (42, x + 24) = x + 66.$$

In fact, however, this notion of symbolic evaluation makes sense only for primitive operations that are *simple*, as defined in Definition 3. In the next section, we subsume the notion of symbolic evaluation in this section with a more general notion that covers all kinds of primitive operations.

3.1.2 Generalized symbolic evaluation of primitive operations

In this section we present the general notion of symbolic evaluation of primitive operations that works for all kinds of operations, including *non-deterministic* operations, defined in Definition 1, and *context-sensitive* operations, defined in Definition 2.

The symbolic evaluation of a set Primop of primitive operations is computed by a function

$$\mathsf{P} \quad \in \quad \mathsf{Primop} \to \mathsf{Exp}^* \to \mathsf{ATR} \to \mathsf{Exp}$$

that, loosely speaking, given

- an n-ary primitive operation $p \in \mathsf{Primop}$,
- n expressions $e_1, \ldots, e_n \in \mathsf{Exp}$, and
- an assignment relation $\delta \in \mathsf{ATR}$ (recall the definition of ATR from page 38),

produces an expression $e = (\mathsf{P}\, p\, (e_1, \ldots, e_n)\, \delta)$ that satisfies the following property. If $(e_1, \ldots, e_n)$ evaluate to values $(v_1, \ldots, v_n)$ in some store σ, and if p applied to these values evaluates to a value v in a store after the assignment δ is applied to σ, then e must evaluate to v in σ.

The following definition formalizes this correctness condition.

Definition 5 (Symbolic evaluation of primitive operations) *If whenever $\sigma\, \delta\, \sigma'$,*

$$\left(\bigwedge_{i=1}^{n} e_i \vdash_\sigma v_i\right) \Rightarrow (p(v_1, \ldots, v_n) \hookrightarrow_{\sigma'} v \Rightarrow (\mathsf{P}\, p\, (e_1, \ldots, e_n)\, \delta) \vdash_\sigma v),$$

then primitive operation $p \in \mathsf{Primop}$ is said to be symbolically evaluated by P*. If every $p \in \mathsf{Primop}$ is symbolically evaluated by P then P is said to be a* symbolic evaluation*.*

It is worth noting that the second implication in the above proposition is not an iff. This means that if there are nondeterministic primitive operations in Primop then P is allowed to produce a nondeterministic expression (which, recall, is an expression using nondeterministic primitive operations) that may evaluate to “extra” values. The reason we allow this is that the only time that we will need the reverse implication is when there are no nondeterministic primitive operations in the first place, and in that case the reverse implication comes for free. This is expressed by the following lemma.

Lemma 4 (Symbolic evaluation of deterministic operations) *If all primitive operations* $p \in \mathsf{Primop}$ *are deterministic, then the second implication relationship above is strengthened to an iff relationship.*

Proof: Because all primitive operations are deterministic, we know that for any p, $v_1, \ldots, v_n$, $e_1, \ldots, e_n$, δ, σ, and σ':

- There is exactly one v such that $p(v_1, \ldots, v_n) \hookrightarrow_{\sigma'} v$.
- From Lemma 2, there is exactly one v such that $(\mathsf{P}\, p\, (e_1, \ldots, e_n)\, \delta) \vdash_\sigma v$.

Therefore, an implication relationship between them is equivalent to an iff relationship. □

Special case: context-independent primitive operations

Because context-independent operations do not use the store, P can safely ignore its assignment-relation argument δ for any such operations. This is described by the following lemma.

Lemma 5 (Symbolic evaluation of context-independent operations) *If* $p \in \mathsf{Primop}$ *is context-independent and*

$$p(e_1, \ldots, e_n) \vdash_\sigma v \Rightarrow (\mathsf{P}\, p\, (e_1, \ldots, e_n)\, \delta) \vdash_\sigma v$$

then p *is symbolically evaluated by* P.

Proof: Straightforward. □

This is similar to the statement of correctness that we suggested above for the simpler notion of symbolic evaluation. The above lemma suggests the requirement, without any loss of generality, that for any context-independent primitive operation p, the expression $(\mathsf{P}\, p\, (e_1, \ldots, e_n)\, \delta)$ must not depend on the particular value of δ. Because many primitive operations are context-independent, this suggests a simpler notation for their symbolic evaluation in which δ does not appear.

Definition 6 (Notation for context-independent operations) *Given* P*, if* p *is context-independent then*

$$\widetilde{p}(e_1, \ldots, e_n)$$

denotes the unique expression $\mathsf{P}\, p\, (e_1, \ldots, e_n)\, \delta$.

For binary primitive operations, we sometimes abbreviate $\widetilde{p}(e, e')$ with $e\, \widetilde{p}\, e'$.

3.1.3 Examples

In this section, we give examples of some of the primitive operations given in Section 2.2.2.

Context-independent primitive operations

There is no reason for any context-independent nullary primitive operation to symbolically evaluate to anything other than itself. So, for instance, we have:

$$\begin{array}{rcll} \widetilde{v}() & = & v & \text{for } v \in \mathsf{Val} \\ \widetilde{\texttt{pos}}() & = & \texttt{pos} & \\ \widetilde{\texttt{bool}}() & = & \texttt{bool} & \text{(see Example 9)} \end{array}$$

An application of a unary operation such as `ptr` from Example 7 also typically symbolically evaluates to itself. Recall that we write $\langle e \rangle$ for $\texttt{ptr}(e)$. Then:

$$\widetilde{\texttt{ptr}}(e) \quad = \quad \langle e \rangle$$

These are clearly correct symbolic evaluations. In fact, for all context-independent primitive operations, the definition

$$\widetilde{p}(e_1, \ldots, e_n) \quad = \quad p(e_1, \ldots, e_n)$$

is trivially a symbolic evaluation by Lemma 5 because

$$\widetilde{p}(e_1, \ldots, e_n) \vdash_\sigma v \iff p(e_1, \ldots, e_n) \vdash_\sigma v$$

simply by definition.

But there may be some room for simplification. For instance, for binary integer operations (including comparison operations), we could define their symbolic evaluation to perform constant-folding when possible, and otherwise default to the above equation. Here, n and n' denote integers (nullary primitive applications).

$$\begin{array}{rcll} n \mathbin{\widetilde{\texttt{+}}} n' & = & n + n' & \\ n \mathbin{\widetilde{\texttt{-}}} n' & = & n - n' & \\ n \mathbin{\widetilde{\texttt{*}}} n' & = & n \times n' & \\ n \mathbin{\widetilde{\texttt{<}}} n' & = & n < n' & \\ n \mathbin{\widetilde{\texttt{>}}} n' & = & n > n' & \\ e \mathbin{\widetilde{p}} e' & = & e \ p \ e & \text{otherwise (where } p \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{<}, \texttt{>}\}) \end{array}$$

Now, we have to prove that those constant-folding clauses are symbolic evaluations. We prove the case for `+`:

$$\begin{array}{lll} & (n \mathbin{\texttt{+}} n') \vdash_\sigma v & \\ \Rightarrow & v = n + n' & \text{definition of } \vdash \text{, value primitives, and } \texttt{+} \\ \Rightarrow & (n + n') \vdash_\sigma v & \text{definition of value primitives} \\ \Rightarrow & (n \mathbin{\widetilde{\texttt{+}}} n') \vdash_\sigma v & \text{definition of } \widetilde{\texttt{+}} \end{array}$$

The constant-folding rules for the other operations are analogous.

Just like the standard arithmetic operations above, the symbolic evaluation of `&` performs the constant-folding taken straight from its definition.

$$\begin{array}{rcll}
\texttt{true}\,\widetilde{\texttt{\&}}\,e & = & e & \\
e\,\widetilde{\texttt{\&}}\,\texttt{true} & = & e & \\
\texttt{false}\,\widetilde{\texttt{\&}}\,e & = & \texttt{false} & \\
e\,\widetilde{\texttt{\&}}\,\texttt{false} & = & \texttt{false} & \\
e\,\widetilde{\texttt{\&}}\,e' & = & e\,\texttt{\&}\,e' & \text{otherwise}
\end{array}$$

Again, the last line is trivially a symbolic evaluation, and so we need to prove the other four lines. The proofs are similar to the example shown above for `+`.

- $\texttt{true}\,\widetilde{\texttt{\&}}\,e$ ($e\,\widetilde{\texttt{\&}}\,\texttt{true}$ analogous):

$$\begin{array}{rll}
 & (\texttt{true}\,\texttt{\&}\,e) \vdash_\sigma v & \\
\Rightarrow & e \vdash_\sigma v & \text{definition of } \vdash,\ \texttt{true},\ \text{and } \texttt{\&} \\
\Rightarrow & (\texttt{true}\,\widetilde{\texttt{\&}}\,e) \vdash_\sigma v & \text{definition of } \widetilde{\texttt{\&}}
\end{array}$$

- $\texttt{false}\,\widetilde{\texttt{\&}}\,e$ ($e\,\widetilde{\texttt{\&}}\,\texttt{false}$ analogous):

$$\begin{array}{rll}
 & (\texttt{false}\,\texttt{\&}\,e) \vdash_\sigma v & \\
\Rightarrow & v = \texttt{false} & \text{definition of } \vdash,\ \texttt{false},\ \text{and } \texttt{\&} \\
\Rightarrow & (\texttt{false}\,\widetilde{\texttt{\&}}\,e) \vdash_\sigma v & \text{definition of } \widetilde{\texttt{\&}}
\end{array}$$

One can go even further for the symbolic evaluation of `=`. Here is a somewhat subtle definition that depends on whether the argument expressions are deterministic, as defined in Definition 4:

$$\begin{array}{rcll}
e\,\widetilde{\texttt{=}}\,e & = & \texttt{true} & \text{if determ}(e) \\
e\,\widetilde{\texttt{=}}\,e & = & \texttt{bool} & \text{if } \neg\text{determ}(e)\ \text{(see Example 9)} \\
v\,\widetilde{\texttt{=}}\,v' & = & \texttt{false} & \text{if } v \neq v' \\
e\,\widetilde{\texttt{=}}\,e' & = & e\,\texttt{=}\,e' & \text{otherwise}
\end{array}$$

If all primitive operations $p \in \textsf{Primop}$ are deterministic then all expressions are deterministic, and so the definition simplifies to:

$$\begin{array}{rcll}
e\,\widetilde{\texttt{=}}\,e & = & \texttt{true} & \\
v\,\widetilde{\texttt{=}}\,v' & = & \texttt{false} & \text{if } v \neq v' \\
e\,\widetilde{\texttt{=}}\,e' & = & e\,\texttt{=}\,e' & \text{otherwise}
\end{array}$$

But we prove the more general formulation. Again, we need to show that the first three lines yield a symbolic evaluation.

- If determ(e):

$$\begin{array}{rll}
 & (e\,\texttt{=}\,e) \vdash_\sigma v & \\
\Rightarrow & \exists v', v''.\ [e \vdash_\sigma v' \wedge e \vdash_\sigma v'' \wedge \texttt{=}(v', v'') \hookrightarrow v] & \text{definition of } \vdash \\
\Rightarrow & \exists v'.\ [e \vdash_\sigma v' \wedge \texttt{=}(v', v') \hookrightarrow v] & \text{because determ}(e) \\
\Rightarrow & v = \texttt{true} & \text{definition of } \texttt{=} \\
\Rightarrow & \texttt{true} \vdash_\sigma v & \text{definition of } \texttt{true} \\
\Rightarrow & (e\,\widetilde{\texttt{=}}\,e) \vdash_\sigma v & \text{definition of } \widetilde{\texttt{=}}
\end{array}$$

- If $\neg\text{determ}(e)$:

$$\begin{array}{lll} & (e \mathbin{\texttt{=}} e) \vdash_\sigma v & \\ \Rightarrow & v = \texttt{true} \lor v = \texttt{false} & \text{definition of } \vdash \text{ and } \texttt{=} \\ \Rightarrow & \texttt{bool} \vdash_\sigma v & \text{definition of } \texttt{bool} \\ \Rightarrow & (e \mathbin{\widetilde{\texttt{=}}} e) \vdash_\sigma v & \text{definition of } \widetilde{\texttt{=}} \end{array}$$

- If $v \neq v'$:

$$\begin{array}{lll} & (v \mathbin{\texttt{=}} v') \vdash_\sigma v & \\ \Rightarrow & \texttt{=}(v, v') \hookrightarrow v & \text{definition of value primitives and } \vdash \\ \Rightarrow & v = \texttt{false} & \text{definition of } \texttt{=} \\ \Rightarrow & \texttt{false} \vdash_\sigma v & \text{definition of } \texttt{false} \\ \Rightarrow & (v \mathbin{\widetilde{\texttt{=}}} v') \vdash_\sigma v & \text{definition of } \widetilde{\texttt{=}} \end{array}$$

For now, we present a very simple symbolic evaluation of `if`:

$$\begin{array}{rcll} \widetilde{\texttt{if}}(\texttt{true}, e, e') & = & e & \\ \widetilde{\texttt{if}}(\texttt{false}, e, e') & = & e' & \\ \widetilde{\texttt{if}}(e, e', e'') & = & \texttt{if}(e, e', e'') & \quad \text{otherwise} \end{array}$$

The proof is straightforward and similar to the proof shown above for `&`. However, it is often important to do a better job of simplifying conditional expressions, and we will give a more sophisticated algorithm in Chapter 9.

Context-dependent primitive operations

The symbolic evaluation of context-dependent primitive operations is much more complicated than the symbolic evaluation of context-independent operations, such as the ones shown above. As we explained above, for a context-independent operation p one can always fall back on

$$\widetilde{p}(e_1, \ldots, e_n) \quad = \quad p(e_1, \ldots, e_n)$$

which is trivially a symbolic evaluation. But the symbolic evaluation of context-dependent operations needs to compute the effect of an arbitrary parallel assignment on the operation. So far, the only context-dependent operations we have seen are `deref` and `tree`. We introduced `tree` mainly for illustration, but on the other hand `deref` is a crucial operation for modeling and analyzing programing languages because, as we explained in Chapter 2, it is the only way to construct an expression that examines the components of mutable data structures. It has a rather complex symbolic evaluation because of aliasing possibilities. Let $\delta = \boxed{l_1, \ldots, l_n \mapsto e''_1, \ldots, e''_n}$.

Then

$$\begin{array}{l}
\mathsf{P}\,\texttt{deref}\,(e,e')\,\delta = \widetilde{\texttt{if}}((e \mathrel{\tilde{=}} e_1) \mathbin{\tilde{\texttt{\&}}} (e' \mathrel{\tilde{=}} e'_1), \\
\qquad\qquad\qquad\qquad e''_1, \\
\qquad\qquad\qquad\qquad \widetilde{\texttt{if}}((e \mathrel{\tilde{=}} e_2) \mathbin{\tilde{\texttt{\&}}} (e' \mathrel{\tilde{=}} e'_2), \\
\qquad\qquad\qquad\qquad\quad e''_2, \\
\qquad\qquad\qquad\qquad\quad \widetilde{\texttt{if}}(\\
\qquad\qquad\qquad\qquad\qquad\quad \ddots \\
\qquad\qquad\qquad\qquad\qquad\quad \widetilde{\texttt{if}}((e \mathrel{\tilde{=}} e_k) \mathbin{\tilde{\texttt{\&}}} (e' \mathrel{\tilde{=}} e'_k), \\
\qquad\qquad\qquad\qquad\qquad\qquad e''_k, \\
\qquad\qquad\qquad\qquad\qquad\qquad e.e')\ldots)))
\end{array}$$

where the indices in the set

$$\{(e_1.e'_1, e''_1), \ldots, (e_k.e'_k, e''_k)\} = \{(l_i, e_i) \mid i \in \{1, \ldots, n\} \wedge l_i \notin \mathsf{Var}\}.$$

are ordered arbitrarily. Now we show that this is a symbolic evaluation. Suppose that $\sigma\,\delta\,\sigma'$, $e \vdash_\sigma v_1$, and $e' \vdash_\sigma v_2$. By the definition of $\texttt{deref}$, if $\texttt{deref}(v_1, v_2) \hookrightarrow_{\sigma'} v$ then $v = \sigma'(v_1.v_2)$. We must show that

$$(\mathsf{P}\,\texttt{deref}\,(e,e')\,\delta) \vdash_\sigma v.$$

Recall that we overload the phrase $e.e'$ to mean not only the l-expression $e.e'$ but also the expression $\texttt{deref}(e, e')$. First, we prove two results. The first one shows how to move down the $\texttt{true}$ arm of the ith branch in the case that there is a possible alias between $e.e'$ and the l-value $e_i.e'_i$ assigned by δ.

$$\begin{array}{lll}
 & (e_i.e'_i) \vdash_\sigma (v_1.v_2) & \\
\Rightarrow & e_i \vdash_\sigma v_1 \wedge e'_i \vdash_\sigma v_2 & \text{definition of } \vdash \\
\Rightarrow & (e \mathrel{\texttt{=}} e_i) \vdash_\sigma \texttt{true} \wedge (e' \mathrel{\tilde{=}} e'_i) \vdash_\sigma \texttt{true} & \text{definition of } \texttt{=} \\
\Rightarrow & (e \mathrel{\tilde{=}} e_i) \vdash_\sigma \texttt{true} \wedge (e' \mathrel{\tilde{=}} e'_i) \vdash_\sigma \texttt{true} & \tilde{=} \text{ symbolically evaluates } \texttt{=} \\
\Rightarrow & ((e \mathrel{\tilde{=}} e_i) \mathbin{\texttt{\&}} (e' \mathrel{\tilde{=}} e'_i)) \vdash_\sigma \texttt{true} & \text{definition of } \texttt{\&} \\
\Rightarrow & ((e \mathrel{\tilde{=}} e_i) \mathbin{\tilde{\texttt{\&}}} (e' \mathrel{\tilde{=}} e'_i)) \vdash_\sigma \texttt{true} & \tilde{\texttt{\&}} \text{ symbolically evaluates } \texttt{\&} \\
\Rightarrow & e'' \vdash_\sigma v \Rightarrow \texttt{if}((e \mathrel{\tilde{=}} e_i) \mathbin{\tilde{\texttt{\&}}} (e' \mathrel{\tilde{=}} e'_i), e'', e''') \vdash_\sigma v & \text{definition of } \texttt{if} \\
\Rightarrow & e'' \vdash_\sigma v \Rightarrow \widetilde{\texttt{if}}((e \mathrel{\tilde{=}} e_i) \mathbin{\tilde{\texttt{\&}}} (e' \mathrel{\tilde{=}} e'_i), e'', e''') \vdash_\sigma v & \widetilde{\texttt{if}} \text{ symbolically evaluates } \texttt{if}
\end{array}$$

The second result shows how to move down the $\texttt{false}$ arm of the ith branch in the case that there is possibly no alias between $e.e'$ and the l-value $e_i.e'_i$ assigned by δ.

$$\begin{array}{lll}
 & (e_i.e'_i) \vdash_\sigma (v'_1.v'_2) \wedge (v_1 \neq v'_1 \vee v_2 \neq v'_2) & \\
\Rightarrow & e_i \vdash_\sigma v'_1 \wedge e'_i \vdash_\sigma v'_2 \wedge (v_1 \neq v'_1 \vee v_2 \neq v'_2) & \text{definition of } \vdash \\
\Rightarrow & (e \mathrel{\texttt{=}} e_i) \vdash_\sigma \texttt{false} \vee (e' \mathrel{\texttt{=}} e'_i) \vdash_\sigma \texttt{false} & \text{definition of } \texttt{=} \\
\Rightarrow & (e \mathrel{\tilde{=}} e_i) \vdash_\sigma \texttt{false} \vee (e' \mathrel{\tilde{=}} e'_i) \vdash_\sigma \texttt{false} & \tilde{=} \text{ symbolically evaluates } \texttt{=} \\
\Rightarrow & ((e \mathrel{\tilde{=}} e_i) \mathbin{\texttt{\&}} (e' \mathrel{\tilde{=}} e'_i)) \vdash_\sigma \texttt{false} & \text{definition of } \texttt{\&} \\
\Rightarrow & ((e \mathrel{\tilde{=}} e_i) \mathbin{\tilde{\texttt{\&}}} (e' \mathrel{\tilde{=}} e'_i)) \vdash_\sigma \texttt{false} & \tilde{\texttt{\&}} \text{ symbolically evaluates } \texttt{\&} \\
\Rightarrow & e''' \vdash_\sigma v \Rightarrow \texttt{if}((e \mathrel{\tilde{=}} e_i) \mathbin{\tilde{\texttt{\&}}} (e' \mathrel{\tilde{=}} e'_i), e'', e''') \vdash_\sigma v & \text{definition of } \texttt{\&} \\
\Rightarrow & e''' \vdash_\sigma v \Rightarrow \widetilde{\texttt{if}}((e \mathrel{\tilde{=}} e_i) \mathbin{\tilde{\texttt{\&}}} (e' \mathrel{\tilde{=}} e'_i), e'', e''') \vdash_\sigma v & \widetilde{\texttt{if}} \text{ symbolically evaluates } \texttt{if}
\end{array}$$

Now, there are two possibilities.

- No overwrite: $\sigma(v_1.v_2) = v$ and δ did not assign to $v_1.v_2$. Then it must be the case that for $i \in \{1, \ldots, k\}$, $(e_i.e'_i) \vdash_\sigma (v'_1.v'_2)$ where $v_1 \neq v'_1$ or $v_2 \neq v'_2$. Therefore, by induction on the definition of $(\mathsf{P}\,\mathtt{deref})$ using the second result above to move down the k $\mathtt{false}$ branches,

  $$e.e' \vdash_\sigma v \Rightarrow (\mathsf{P}\,\mathtt{deref}\,(e, e')\,\delta) \vdash_\sigma v.$$

  And because in this case $\sigma(v_1.v_2) = v$, by the definition of $\mathtt{deref}$ we indeed have that $e.e' \vdash_\sigma v$.

- Overwrite: δ assigned $v_1.v_2$ to be v. Then it must be the case that for some $i \in \{1, \ldots, k\}$, $(e_i.e'_i) \vdash_\sigma (v_1.v_2)$, and for all $j < i$, $(e_j.e'_j) \vdash_\sigma (v'_1.v'_2)$ where $v_1 \neq v'_1$ or $v_2 \neq v_2$. Therefore, by induction on the definition of $(\mathsf{P}\,\mathtt{deref})$ using the second result above to move down $i - 1$ $\mathtt{false}$ branches and then the first result above to move down the next $\mathtt{true}$ branch,

  $$e''_i \vdash_\sigma v \Rightarrow (\mathsf{P}\,\mathtt{deref}\,(e, e')\,\delta) \vdash_\sigma v.$$

  And because in this case δ assigned $v_1.v_2$ to be v, it must be the case that $e''_i \vdash_\sigma v$.

3.2 Symbolic Evaluation of Expressions and L-expressions

This section describes the following algorithms, which are defined relative to a set Primop of primitive operations with associated symbolic evaluation algorithm P.

$$\begin{array}{lcl} \mathsf{E} & \in & \mathsf{Exp} \rightarrow \mathsf{TR} \rightarrow \mathsf{Exp} \\ \mathsf{L} & \in & \mathsf{Lexp} \rightarrow \mathsf{TR} \rightarrow \mathsf{Lexp} \end{array}$$

Loosely speaking, the E algorithm, given an expression e and a transfer relation Δ, computes an expression e' such that if e' evaluates to a value v in some store σ then e evaluates to v in a store to which Δ transfers from σ (i.e., a store σ' such that $\sigma\,\Delta\,\sigma'$). In other words, e' expresses the combined effects of e and Δ. The L algorithm is similar, but works on l-expressions rather than expressions. Intuitively, given an l-expression l and a transfer relation Δ, L computes an expression l' such that if l' evaluates to an l-value w before Δ then l evaluates to w in a store to which Δ transfers from σ. We distinguish two levels of correctness of E and L, given by the following definition.

Definition 7 (Symbolic evaluation of expressions and l-expressions) *We introduce the following terms to describe correctness properties of* E *and* L*.*

- *If whenever* $\sigma\,\Delta\,\sigma'$*,*

  $$e \vdash_{\sigma'} v \Rightarrow (\mathsf{E}\,e\,\Delta) \vdash_\sigma v \qquad \textit{respectively, } (l \vdash_{\sigma'} w \Rightarrow (\mathsf{L}\,l\,\Delta) \vdash_\sigma w),$$

  then E *(respectively,* L*) is said to be an* upper approximation*.*

- *If whenever* $\sigma\,\Delta\,\sigma'$*,*

  $$e \vdash_{\sigma'} v \iff (\mathsf{E}\,e\,\Delta) \vdash_\sigma v \qquad \textit{respectively, } (l \vdash_{\sigma'} w \iff (\mathsf{L}\,l\,\Delta) \vdash_\sigma w),$$

  then E *(respectively,* L*) is said to be a* translation*.*

3.2.1 The algorithm

The definition of E is inductive on the structure of its arguments, and L is defined in terms of E.

$$\begin{array}{rcl}
\mathsf{E}\, e\, \emptyset & = & \text{any } e' \in \mathsf{Exp} \\[1em]
\mathsf{E}\, e \boxed{e'?\ \Delta} & = & \mathsf{E}\, e\, \Delta \\[1em]
\mathsf{E}\, e \boxed{e'¿\ \Delta} & = & \mathsf{E}\, e\, \Delta \\[1em]
\mathsf{E}\, e \boxed{e'?\ \Delta \,\pmb{|}\, \Delta'} & = & \widetilde{\mathtt{if}}(e', (\mathsf{E}\, e\, \Delta), (\mathsf{E}\, e\, \Delta')) \\[1em]
\mathsf{E}\, x \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n} & = & \left\{ \begin{array}{ll} e_i & \text{if } l_j = x \iff j = i \\ x & \text{otherwise} \end{array} \right. \\
\mathsf{E}\, (p(e_1, \ldots, e_n))\, \delta & = & \mathsf{P}\, p\, (\mathsf{E}\, e_1\, \delta, \ldots, \mathsf{E}\, e_n\, \delta)\, \delta \\[2em]
\mathsf{L}\, x\, \Delta & = & x \\[1em]
\mathsf{L}\, (e.e')\, \Delta & = & (\mathsf{E}\, e\, \Delta).(\mathsf{E}\, e'\, \Delta)
\end{array}$$

Notice that the first line allows the choice of any expression. This is because the transfer relation $\emptyset$ never outputs a store, and so any expression is trivially correct.

Because E is defined by structural induction and L is defined in terms of E, and because the only external algorithm they need is the P algorithm to symbolically evaluate primitive operations, we have that if P always terminates then E and L always terminate.

These algorithms are not only used in the composition algorithm $\oplus$ to come, but they are also useful in their own right, as stand-alone applications of our analysis methodology. Chapter 9 gives an application that is centered around the E algorithm.

The next two lemmas prove the correctness of these algorithms. If all primitive operations $p \in \mathsf{Primop}$ are deterministic then the algorithms are translations, and otherwise we can only show that they are upper approximations.

Theorem 1 (E and L as upper approximations) *If* P *is a symbolic evaluation then* E *and* L *are upper approximations.*

Proof: By the definition of upper approximation in Definition 7, we must prove that whenever $\sigma\, \Delta\, \sigma'$, the following properties hold:

- $e \vdash_{\sigma'} v \Rightarrow (\mathsf{E}\, e\, \Delta) \vdash_{\sigma} v$
- $l \vdash_{\sigma'} w \Rightarrow (\mathsf{L}\, l\, \Delta) \vdash_{\sigma} w$

We prove this by mutual structural induction on the arguments to E and L, following their inductive definitions above. There are six cases for E.

- $\mathsf{E}\,e\,\emptyset$: By definition there is no σ and σ' such that $\sigma\,\emptyset\,\sigma'$, and so the theorem statement is trivially satisfied.
- $\mathsf{E}\,e\,\boxed{e'?\ \Delta}$: By the definition of conditional relations, we know that if $\sigma\,\boxed{e'?\ \Delta}\,\sigma'$ then $\sigma\,\Delta\,\sigma'$.

$$\begin{array}{lll} & e \vdash_{\sigma'} v & \\ \Rightarrow & (\mathsf{E}\,e\,\Delta) \vdash_\sigma v & \text{induction, with above observation}\\ \Rightarrow & (\mathsf{E}\,e\,\boxed{e'?}\Delta) \vdash_\sigma v & \text{definition of } \mathsf{E}\end{array}$$

- $\mathsf{E}\,e\,\boxed{e'\text{¿}\ \Delta}$: Analogous to the previous case.
- $\mathsf{E}\,e\,\boxed{e'?\ \Delta\,\boldsymbol{|}\,\Delta'}$: By the definition of conditional relations, we know that if $\sigma\,\boxed{e'?\ \Delta\,\boldsymbol{|}\,\Delta'}\,st'$ then either $e' \vdash_\sigma \mathtt{true}$ and $\sigma\,\Delta\,\sigma'$, or $e' \vdash_\sigma \mathtt{false}$ and $\sigma\,\Delta'\,\sigma'$.

$$\begin{array}{lll} & e \vdash_{\sigma'} v & \\ \Rightarrow & (e' \vdash_\sigma \mathtt{true} \wedge (\mathsf{E}\,e\,\Delta) \vdash_\sigma v) & \\ & \quad \vee\ (e' \vdash_\sigma \mathtt{false} \wedge (\mathsf{E}\,e\,\Delta') \vdash_\sigma v) & \text{induction, with above observation}\\ \Rightarrow & \mathtt{if}(e', (\mathsf{E}\,e\,\Delta), (\mathsf{E}\,e\,\Delta')) \vdash_\sigma v & \text{definition of } \mathtt{if}\\ \Rightarrow & \widetilde{\mathtt{if}}(e', (\mathsf{E}\,e\,\Delta), (\mathsf{E}\,e\,\Delta')) \vdash_\sigma v & \mathsf{P} \text{ is a symbolic evaluation}\\ \Rightarrow & (\mathsf{E}\,e\,\boxed{e'?\ \Delta\,\boldsymbol{|}\,\Delta'}) \vdash_\sigma v & \text{definition of } \mathsf{E}\end{array}$$

- $\mathsf{E}\,x\,\boxed{l_1,\ldots,l_n \mapsto e_1,\ldots,e_n}$: Note that if $l_i = l_j = x$ then $i = j$, because otherwise l_i and l_j could not evaluate to different l-values and it could not be the case that $\sigma\,\delta\,\sigma'$.

$$\begin{array}{lll} & x \vdash_{\sigma'} v & \\ \Rightarrow & \sigma'\,x = v & \text{definition of } \vdash\\ \Rightarrow & (l_i = x \wedge e_i \vdash_\sigma v) \vee (\sigma\,x = v \wedge \neg\exists i.\ l_i = x) & \text{definition of assignment relations}\\ \Rightarrow & (l_i = x \wedge e_i \vdash_\sigma v) \vee (x \vdash_\sigma v \wedge \neg\exists i.\ l_i = x) & \text{definition of } \vdash\\ \Rightarrow & (\mathsf{E}\,x\,\boxed{l_1,\ldots,l_n \mapsto e_1,\ldots,e_n}) \vdash_\sigma v & \text{definition of } \mathsf{E}\end{array}$$

- $\mathsf{E}\,(p(e_1,\ldots,e_n))\,\delta$:

$$\begin{array}{lll} & (p(e_1,\ldots,e_n)) \vdash_{\sigma'} v & \\ \Rightarrow & \exists v_1,\ldots,v_n.\ [(\bigwedge_{i=1}^n e_i \vdash_{\sigma'} v_i) \wedge p(v_1,\ldots,v_n) \hookrightarrow_{\sigma'} v] & \text{definition of } \vdash\\ \Rightarrow & \exists v_1,\ldots,v_n.\ [(\bigwedge_{i=1}^n (\mathsf{E}\,e_i\,\delta) \vdash_\sigma v_i) \wedge p(v_1,\ldots,v_n) \hookrightarrow_{\sigma'} v] & \text{induction}\\ \Rightarrow & (\mathsf{P}\,p\,(\mathsf{E}\,e_1\,\delta,\ldots,\mathsf{E}\,e_n\,\delta)\,\delta) \vdash_\sigma v & \mathsf{P} \text{ is a symbolic evaluation}\\ \Rightarrow & (\mathsf{E}\,(p(e_1,\ldots,e_n))\,\delta) \vdash_\sigma v & \text{definition of } \mathsf{E}\end{array}$$

There are two cases for L.

- $\mathsf{L}\,x\,\Delta$:

$$\begin{array}{lll} & x \vdash_{\sigma'} w & \\ \Rightarrow & w = x & \text{definition of } \vdash\\ \Rightarrow & x \vdash_\sigma w & \text{definition of } \vdash\\ \Rightarrow & (\mathsf{L}\,x\,\Delta) \vdash_\sigma w & \text{definition of } \mathsf{L}\end{array}$$

- $\mathsf{L}\,(e.e')\,\Delta$:

$$\begin{array}{lll}
 & (e.e') \vdash_{\sigma'} w & \\
\Rightarrow & \exists v, v'.\ [e \vdash_{\sigma'} v \wedge e' \vdash_{\sigma'} v' \wedge w = v.v'] & \text{definition of } \vdash \\
\Rightarrow & \exists v, v'.\ [(\mathsf{E}\,e\,\Delta) \vdash_{\sigma} v \wedge (\mathsf{E}\,e'\,\Delta) \vdash_{\sigma} v' \wedge w = v.v'] & \text{induction} \\
\Rightarrow & ((\mathsf{E}\,e\,\Delta).(\mathsf{E}\,e'\,\Delta)) \vdash_{\sigma} w) & \text{definition of } \vdash \\
\Rightarrow & (\mathsf{L}\,(e.e')\,\Delta) \vdash_{\sigma} w & \text{definition of } \mathsf{L}
\end{array}$$

□

Theorem 2 (E and L as translations) *If all primitive operations $p \in$ Primop are deterministic and P is a symbolic evaluation then both E and L are translations.*

Proof: We prove the statement for E. Because P is a symbolic evaluation, we have from Theorem 1 that E is an upper approximation. Because all primitive operations are deterministic, we know from Lemma 2 that for any e, Δ, σ, and σ':

- There is exactly one v such that $(\mathsf{E}\,e\,\Delta) \vdash_{\sigma} v$.
- There is exactly one v such that $e \vdash_{\sigma'} v$.

Therefore, the implication relationship in the definition of upper approximation of E is equivalent to an iff relationship, and therefore E is a translation. The proof for L is analogous. □

3.2.2 Examples

The E algorithm not only is required for the composition algorithm $\oplus$ that we will present later in this chapter, but is also useful on its own for the analysis of how values relate to each other at different times of program execution. For instance, dependency analysis [ASU86] is concerned with such properties. In Chapter 9 we will see some example applications of E.

Here, we will give some examples of how E works. The L algorithm is just an application of E, so we will not demonstrate it separately. The simplest examples are those in which the expression given to E as input does not contain any context-dependent primitives. Here are some examples, where $x \neq y$ are variables.

$$\begin{array}{llll}
\mathsf{E} & x & \boxed{y \mapsto 3} & = x \\
\mathsf{E} & y & \boxed{y \mapsto 3} & = 3 \\
\mathsf{E} & x + y & \boxed{y \mapsto 3} & = x + 3 \\
\mathsf{E} & y + y & \boxed{y \mapsto 3} & = 6 \\
\mathsf{E} & x + y & \boxed{x, y \mapsto 3, 4} & = 7 \\
\mathsf{E} & x & \boxed{(x = 3)?\ \boxed{x \mapsto 4} \,|\, \boxed{y \mapsto 5}} & = \mathtt{if}(x = 3, 4, x)
\end{array}$$

A more complicated case, however, is when E is given an expression that uses the context-dependent operation **`deref`** to examine the store. Recall that we overload the phrase $e.e'$ to mean not only the l-expression $e.e'$ but also the expression $\mathtt{deref}(e, e')$. In the following examples, $v \neq v'$ are members of Val that are included as constant (nullary) primitive operations. Examples of such values that might occur in a real programming language are record field names, the C `*` token, and integers representing array indices. Also, $x \neq y \neq z$ are variables. In the following examples, some of the equality terms in the symbolic evaluation of `deref` are simplified to `true` or `false` due to the symbolic evaluation of equality on values (v and v' in this case), and thereby simplify the resulting symbolic evaluation of `if`.

$$\begin{array}{llll}
\mathsf{E} & x.v & \boxed{x \mapsto y} & = y.v \\
\mathsf{E} & x.v & \boxed{x.v \mapsto 3} & = 3 \\
\mathsf{E} & x.v & \boxed{y.v \mapsto 3} & = \mathtt{if}(x = y, 3, x.v) \\
\mathsf{E} & x.v & \boxed{y.v' \mapsto 3} & = x.v \\
\mathsf{E} & x.v & \boxed{y.v, z.v \mapsto 3, 4} & = \mathtt{if}(x = y, 3, \mathtt{if}(x = z, 4, x.v)) \\
\mathsf{E} & x.v & \boxed{x.v.v \mapsto 3} & = \mathtt{if}(x = x.v, 3, x.v) \\
\mathsf{E} & x.v & \boxed{y.v'.v \mapsto 3} & = \mathtt{if}(x = y.v', 3, x.v) \\
\mathsf{E} & x.v & \boxed{x.v, y.v \mapsto 3, 4} & = 3 \\
\mathsf{E} & x.v & \boxed{y.v, x.v \mapsto 4, 3} & = \mathtt{if}(x = y, 4, 3)
\end{array}$$

The last two examples may seem strange. Recall that the symbolic evaluation of `deref`, given some assignment relation, chooses an arbitrary order of its assignments and then builds a linear sequence of nested `if` expressions. In the above examples, we choose the left-to-right order to demonstrate that the order does indeed play a practical role in the quality of the output.

The penultimate example first checks x for equality against x, which simplifies to `true` and thus simplifies the entire result to 3. The justification of this general procedure is given in the rather intricate proof of the symbolic evaluation of `deref`. Intuitively, in this case the output expression does not have to check for aliasing because the semantics of assignment relations guarantees that if $\sigma \boxed{x.v, y.v \mapsto 3, 4} \sigma'$ then x and y are bound to different values in σ. Because order of assignment is irrelevant, 3 would have thus been a correct output for the last example, as well. However, E instead outputs $\texttt{if}(x \texttt{ = } y, 4, 3)$. The reason that it tests the equality x with y first, which cannot be simplified. This suggests that the symbolic evaluation of `deref` should instead choose an order that places first any alias test that simplifies to `true`. In this case, the last example would indeed output

$$\mathsf{E} \quad x.v \quad \boxed{y.v, x.v \mapsto 4, 3} \quad = \quad 3$$

If the second argument of a `deref` expression (i.e., the expression to the right of the dot) is not a value, then the symbolic evaluations cannot perform as many simplifications. An example of such a case that might occur in a programming language is an array access where the index is a non-constant expression. Here are some more complicated examples, where $e \neq e'$ are non-value expressions. We first note that nondeterministic primitive operations may produce a more complex output expression. For instance, if determ(e) (i.e., if e contains no nondeterministic primitive operation) then

$$\mathsf{E} \quad x.e \quad \boxed{x.e \mapsto 3} \quad = \quad 3$$

as expected, but if ¬determ(e) then

$$\mathsf{E} \quad x.e \quad \boxed{x.e \mapsto 3} \quad = \quad \texttt{if}(\texttt{bool}, 3, x.e)$$

where `bool` is a nondeterministic operation that evaluates to both `true` and `false`. The reason is justified in the proof of the symbolic evaluation of `=`. Intuitively, because e contains a nondeterministic primitive operation, it may evaluate to two values $v \neq v'$, and in that case the expression $e \texttt{ = } e$ evaluates to both `true` and `false`.

In the remaining examples, we assume determ(e) and determ(e').

$$\mathsf{E} \quad x.e \quad \boxed{x.e' \mapsto 3} \quad = \quad \texttt{if}(e \texttt{ = } e', 3, x.e)$$

$$\mathsf{E} \quad x.e \quad \boxed{y.e \mapsto 3} \quad = \quad \texttt{if}(x \texttt{ = } y, 3, x.e)$$

$$\mathsf{E} \quad x.e \quad \boxed{y.e' \mapsto 3} \quad = \quad \texttt{if}((x \texttt{ = } y) \texttt{ \& } (e \texttt{ = } e'), 3, x.e)$$

3.3 Symbolic Evaluation of Conditional Relations

The remainder of the composition algorithm is parameterized by an algorithm that constructs a conditional transfer relation from a conditional expression and a transfer relation for each of the two branches.

$$\mathsf{C} \quad \in \quad \mathsf{Exp} \to \mathsf{TR} \to \mathsf{TR} \to \mathsf{TR}$$

As for `E` and `L`, we distinguish two different correctness conditions.

Definition 8 (Symbolic evaluation of conditional relations) *We introduce the following terms to describe correctness properties of* C.

- *If*

  $$\sigma \boxed{e?\ \Delta \,\boldsymbol{|}\, \Delta'} \sigma' \Rightarrow \sigma\,(\mathsf{C}\,e\,\Delta\,\Delta')\,\sigma'$$

  then C *is said to be an* upper approximation.

- *If*

  $$\sigma \boxed{e?\ \Delta \,\boldsymbol{|}\, \Delta'} \sigma' \iff \sigma\,(\mathsf{C}\,e\,\Delta\,\Delta')\,\sigma'$$

  then C *is said to be a* translation.

The following lemma makes it easier to prove the stronger property about C in the case that all primitive operations are deterministic.

Lemma 6 *If all primitive operations* $p \in \mathsf{Primop}$ *are deterministic,* C *is an upper approximation, and*

$$\sigma\,(\mathsf{C}\,e\,\Delta\,\Delta')\,\sigma' \Rightarrow \exists\sigma''.\ \sigma \boxed{e?\ \Delta \,\boldsymbol{|}\, \Delta'} \sigma''$$

then C *is a translation.*

Proof: From Corollary 1. □

The most obvious choice for C is simply

$$\mathsf{C}\,e\,\Delta\,\Delta' = \boxed{e?\ \Delta \,\boldsymbol{|}\, \Delta'}.$$

But it is sometimes possible to simplify the resulting transfer relation. For example,

$$\mathsf{C}\,\mathtt{true}\,\Delta\,\Delta' = \Delta.$$

3.4 Engineering Flexibility

The P and C algorithms provide an engineering flexibility for the composition algorithm. There are many correct choices for the syntactic composition $\Delta \oplus \Delta'$, but most of the differences involve how far primitive applications are simplified and how far conditionals are simplified. Different algorithms P and C will allow a tradeoff between the cost of computing a composition and its quality.

3.5 Symbolic Evaluation of Assignment Merging

The most difficult part of the composition algorithm, which we will present in full in Section 3.6, is the composition of two assignment relations δ and δ'. Consider the composition

$$\boxed{\mathtt{x} \mapsto 3} ; \boxed{\mathtt{y} \mapsto \mathtt{x}}.$$

The first issue is that the l-expression y and the expression x that occur in $\boxed{\mathtt{y} \mapsto \mathtt{x}}$ are evaluated in a store *after* the assignment $\boxed{\mathtt{x} \mapsto 3}$ takes place. But to build a transfer relation in TR that is equivalent to this assignment, we must first build a corresponding l-expression and a corresponding expression that are to be evaluated in a store *before* the assignment $\boxed{\mathtt{x} \mapsto 3}$. The L and E algorithms accomplish this task. First of all, we compute

$$\mathsf{L}\,\mathtt{y}\,\boxed{\mathtt{x} \mapsto 3} = \mathtt{y}.$$

This expresses the fact that the l-expression y evaluates to the same l-value (which happens to be y) both before and after $\boxed{\mathtt{x} \mapsto 3}$. Then, we compute

$$\mathsf{E}\,\mathtt{x}\,\boxed{\mathtt{x} \mapsto 3} = 3.$$

This expresses the fact that the expression 3 evaluates before the assignment $\boxed{\mathtt{x} \mapsto 3}$ to the same value to which x evaluates after the assignment.

So, we replace y by y and x by 3, yielding the assignment relation

$$\boxed{\mathtt{y} \mapsto 3}.$$

Now we must *merge* the first assignment, $\boxed{\mathtt{x} \mapsto 3}$, with this new assignment, yielding

$$\boxed{\mathtt{x}, \mathtt{y} \mapsto 3, 3}$$

for the composition.

This "merging" is *not* the same as composition. Simply merging

$$\boxed{\mathtt{x} \mapsto 3}$$

with

$$\boxed{\mathtt{y} \mapsto \mathtt{x}}$$

yields

$$\boxed{\mathtt{x}, \mathtt{y} \mapsto 3, \mathtt{x}}$$

which is *not* semantically equivalent to their composition

$$\boxed{\mathtt{x}, \mathtt{y} \mapsto 3, 3}.$$

The merging operation is simpler than composition, because it does not use E and L generate the "adjusted" expressions and l-expressions. This section presents an algorithm to perform assignment merging, and the composition operation $\oplus$ will use this merging operation as a subroutine, in the manner we described above.

The reason that we need an algorithm to perform assignment merging is that it is not always as trivial as merely concatenating two lists of l-expressions and expressions, as we did for the example above. For example, consider merging

$$\boxed{\mathtt{x}, \mathtt{y} \mapsto 2, 3}$$

with

$$\boxed{\mathtt{x}, \mathtt{z} \mapsto \mathtt{w}, \mathtt{y}}.$$

The literal concatenation of these two is

$$\boxed{\mathtt{x}, \mathtt{y}, \mathtt{x}, \mathtt{z} \mapsto 2, 3, \mathtt{w}, \mathtt{y}}$$

which is semantically equivalent to the empty relation $\emptyset$ because there are two l-expressions `x` that always evaluate to the same l-value `x`. The merging that we are looking for is

$$\boxed{\mathtt{x}, \mathtt{y}, \mathtt{z} \mapsto \mathtt{w}, 3, \mathtt{y}}$$

that *replaces* the assignment to `x` in the first relation with the assignment to `x` in the second relation.

Note once again that this is not semantically equivalent to the composition of the two relations because the composition will perform the assignment to `y` *before* evaluating `y` in the second assignment. In the merging of two assignment relations δ and δ', the l-expressions and expressions in both δ and δ' are considered to be evaluated in the same initial store.

The merging of δ with δ' is written $\delta \otimes \delta'$. (This operator $\otimes$ should not be confused with the composition operator $\oplus$.) It is not symmetric, because, as in the example above, if δ and δ' both assign to the same l-value, the conflict is resolved in favor of δ'. Recall that the semantics of an assignment relation

$$\delta = \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n}$$

requires that the n l-values to which $l_1, \ldots, l_n$ evaluate *must be distinct* in order for the assignment to take place. Given the above assignment relation and a second assignment relation

$$\delta' = \boxed{l'_1, \ldots, l'_m \mapsto e'_1, \ldots, e'_m},$$

the relation $\delta \otimes \delta'$ defined by the following rule:

$$\frac{\begin{array}{c} l_i \vdash_\sigma w_i \quad e_i \vdash_\sigma v_i \quad i \neq j \Rightarrow w_i \neq w_j \\ l'_i \vdash_\sigma w'_i \quad e'_i \vdash_\sigma v'_i \quad i \neq j \Rightarrow w'_i \neq w'_j \\ \sigma' = \sigma[w_1 \mapsto v_1] \ldots [w_n \mapsto v_n][w'_1 \mapsto v'_1] \ldots [w'_m \mapsto v'_m] \end{array}}{\sigma\,(\delta \otimes \delta')\,\sigma'}$$

Again, note that $\delta \otimes \delta'$ is not necessarily semantically equivalent to the concatenation

$$\boxed{l_1, \ldots, l_n, l'_1, \ldots, l'_m \mapsto e_1, \ldots, e_n, e'_1, \ldots, e'_m}$$

because the above rule allows w_i to be equal to w'_j for some i and j. Intuitively, $\delta \otimes \delta'$ cannot merely union the assignments in δ and δ' because an assignment in the latter may *overwrite* an assignment in the former.

In this section, we present an algorithm $\overline{\otimes}$ to compute $\otimes$, which is one of the most difficult parts of the composition algorithm $\oplus$. The algorithm $\overline{\otimes}$ is inductive, and for ease of notation in its correctness proof we generalize $\otimes$ to take an additional parameter J to reflect this induction: a set of indices of the assignments in the right-hand relation. The following rule defines this generalized $\otimes_J$.

$$\frac{\begin{array}{c} l_i \vdash_\sigma w_i \quad e_i \vdash_\sigma v_i \quad i \neq j \Rightarrow w_i \neq w_j \\ l'_i \vdash_\sigma w'_i \quad e'_i \vdash_\sigma v'_i \quad i \neq j \Rightarrow w'_i \neq w'_j \\ j \in J \Rightarrow w'_j \notin \{w_1, \ldots, w_n\} \\ \sigma' = \sigma[w_1 \mapsto v_1] \ldots [w_n \mapsto v_n][w'_1 \mapsto v'_1] \ldots [w'_m \mapsto v'_m] \end{array}}{\sigma \, (\delta \otimes_J \delta') \, \sigma'}$$

In the relation $\delta \otimes_J \delta'$, J is a set of indices into the list of assignments in δ'. If $j \in J$, then the jth l-expression in δ' must not overwrite any assignment in δ. The relation $\delta \otimes_\emptyset \delta'$ is simply $\delta \otimes \delta'$. If the length of δ' is m, then the relation $\delta \otimes_{\{1,\ldots,m\}} \delta'$ is semantically equivalent to the literal concatenation of δ with δ' as we described above.

Before we present $\overline{\otimes}$, we need an auxilliary algorithm

$$\sim \quad \in \quad \mathsf{Lexp} \times \mathsf{Lexp} \to \mathsf{Exp}$$

that, given two l-expressions l and l', generates an expression that tests if the l and l' can evaluate to the same l-value or to different l-values. It is defined to be `false` except for the following cases:

$$\begin{array}{rcl} x \sim x & = & \texttt{true} \\ e_1.e_2 \sim e'_1.e'_2 & = & (e_1 \mathrel{\tilde{\cong}} e'_1) \mathbin{\tilde{\&}} (e_2 \mathrel{\tilde{\cong}} e'_2) \end{array}$$

Formally, we have the following properties of $\sim$.

Lemma 7 *If* P *is a symbolic evaluation, then:*

- *If* $\exists w.\ [l \vdash_\sigma w \wedge l' \vdash_\sigma w]$ *then* $(l \sim l') \vdash_\sigma \texttt{true}$.
- *If* $\exists w, w'.\ [w \neq w' \wedge l \vdash_\sigma w \wedge l' \vdash_\sigma w]$ *then* $(l \sim l') \vdash_\sigma \texttt{false}$.

Proof: Straightforward. □

Now we present the algorithm

$$\overline{\otimes} \quad \in \quad \mathsf{ATR} \times \mathsf{ATR} \times \mathcal{P}_{\mathrm{fin}}(\mathsf{Nat}) \to \mathsf{TR}$$

to compute $\otimes$ as follows:

$$\begin{array}{rcl} \bullet\,\overline{\otimes}_J\,\delta' & = & \delta' \\ \boxed{l_1,\ldots,l_n \mapsto e_1,\ldots,e_n}\,\overline{\otimes}_J\,\delta' & = & \Delta_k \end{array}$$

where

$$\begin{array}{rcll} \delta & = & \boxed{l_2,\ldots,l_n \mapsto e_2,\ldots,e_n} & \\ \delta' & = & \boxed{l'_1,\ldots,l'_m \mapsto e'_1,\ldots,e'_m} & \\ \{j_1,\ldots,j_k\} & = & \{1,\ldots,m\} - J & \text{ordered arbitrarily} \\ \Delta_0 & = & \delta\,\overline{\otimes}_{J\cup\{m+1\}}\,\boxed{l'_1,\ldots,l'_m,l_1 \mapsto e'_1,\ldots,e'_m,e_1} & \\ \Delta_i & = & \mathsf{C}\,(l_1 \sim l'_{j_i})\,(\delta\,\overline{\otimes}_{J\cup\{j_i\}}\,\delta')\,\Delta_{i-1} & \end{array}$$

The order of $j_1,\ldots,j_k$ is arbitrary; the correctness proof will make no assumption as to their order. There may be engineering advantages to choosing a particular order dynamically, because a particular choice of C might produce different results with different orderings. This is similar to the situation with the symbolic evaluation of `deref` that we illustrated with the examples in Section 3.2.

Intuitively, $\delta\,\overline{\otimes}\,\delta'$ examines each assignment of δ in turn, to see which ones might be overwritten by δ' and thus should be eliminated, and which ones might not be overwritten by δ' and thus should remain. The assignments in δ are so processed from left to right. The l-expression of each one is tested in turn, via $\sim$, against the l-expressions of δ' not already in the set J. Whenever an l-expression in δ might be equal to some l-expression l'_j in δ', that l-expression never needs to be tested for equivalence again, and so j is added to J. It is this handling of J that is rather subtle, but the correctness proof explains this in detail.

Because $\overline{\otimes}$ is defined by structural induction, and because the only external algorithms it needs are the P algorithm to symbolically evaluate the primitive operations in $\sim$ and the C algorithm to symbolically evaluate conditional relations, we have that if P and C always terminate then $\overline{\otimes}$ always terminates.

Now we may proceed with the proof of $\overline{\otimes}$. First we show that $\overline{\otimes}$ computes a relation that includes $\otimes$.

Lemma 8 *If* P *is a symbolic evaluation,* C *is an upper approximation, and* $\sigma\,(\delta \otimes_J \delta')\,\sigma'$*, then* $\sigma\,(\delta\,\overline{\otimes}_J\,\delta')\,\sigma'$*.*

Proof: By induction on the size of δ. If $\delta = \bullet$ then the result is immediate. Otherwise, without loss of generality, let

$$\delta = \boxed{l_1,\ldots,l_n \mapsto e_1,\ldots,e_n} \qquad \delta' = \boxed{l'_1,\ldots,l'_m \mapsto e'_1,\ldots,e'_m}$$

and let $\{j_1, \ldots, j_k\} = \{1, \ldots, m\} - J$, where the order of $j_1, \ldots, j_k$ is arbitrary. Let $w_1, \ldots, w_n$, $v_1, \ldots, v_n$, $w'_1, \ldots, w'_m$, and $v'_1, \ldots, v'_m$ be as given by the definition of $\otimes_J$. By that definition, we know that

- $w_1 \notin \{w_2, \ldots, w_n\}$, and
- $w_1 \notin \{w'_j \mid j \in J\}$.

There are two cases.

- Case 1: There is some $j \notin J$ such that $w_1 = w'_j$. Because P is a symbolic evaluation we have by Lemma 7 that
$$(l_1 \sim l_j) \vdash_\sigma \texttt{true}.$$
In this case, the update to w_1 in store σ is overwritten by the later update to w'_j, and so in this case it may be removed from the definition of σ' in the rule that defines $\otimes_J$. Furthermore, because $w_1 = w'_j$ we have that $w'_j \notin \{w_2, \ldots, w_n\}$, and so j may be added to J in the rule that defines $\otimes_J$. Therefore,
$$\sigma\,(\boxed{l_2, \ldots, l_n \mapsto e_2, \ldots, e_n} \otimes_{J \cup \{j\}} \delta')\,\sigma'.$$
By induction, we have that
$$\sigma\,(\boxed{l_2, \ldots, l_n \mapsto e_2, \ldots, e_n} \overline{\otimes}_{J \cup \{j\}} \delta')\,\sigma'.$$

- Case 2: There is no $j \notin J$ such that $w_1 = w'_j$, and so
$$\bigwedge_{j \notin J} (l_1 \sim l'_j) \vdash_\sigma \texttt{false}.$$
In this case, $w_1 \notin \{w_2, \ldots, w_n, w'_1, \ldots, w'_m\}$. Hence, the update of w_1 to v_1 in store σ is not overwritten and thus may be moved to the end of the list of updates in the definition of $\otimes_J$. Therefore,
$$\sigma\,(\boxed{l_2, \ldots, l_n \mapsto e_2, \ldots, e_n} \otimes_{J \cup \{m+1\}} \boxed{l'_1, \ldots, l'_m, l_1 \mapsto e'_1, \ldots, e'_m, e_1})\,\sigma'.$$
By induction, we have that
$$\sigma\,(\boxed{l_2, \ldots, l_n \mapsto e_2, \ldots, e_n} \overline{\otimes}_{J \cup \{m+1\}} \boxed{l'_1, \ldots, l'_m, l_1 \mapsto e'_1, \ldots, e'_m, e_1})\,\sigma'.$$

Therefore, because C is an upper approximation, either all of the branches in $\delta \,\overline{\otimes}_J\, \delta'$ will evaluate to $\texttt{false}$ in σ, in which case $\sigma\,(\delta \,\overline{\otimes}_J\, \delta')\,\sigma'$ by Case 2, or at least one the branches will evaluate to $\texttt{true}$ in σ, in which case $\sigma\,(\delta \,\overline{\otimes}_J\, \delta')\,\sigma'$ by Case 1. □

Now we show that if all primitive operations are deterministic, $\overline{\otimes}$ computes a relation that is precisely $\otimes$.

Lemma 9 *If all primitive operations $p \in \mathsf{Primop}$ are deterministic, P is a symbolic evaluation, C is a translation, and $\sigma\,(\delta \,\overline{\otimes}_J\, \delta')\,\sigma'$, then $\sigma\,(\delta \otimes_J \delta')\,\sigma'$.*

Proof: By induction on the size of δ. If $\delta = \bullet$ then the result is immediate. Otherwise, without loss of generality, let

$$\delta = \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n} \qquad \delta' = \boxed{l'_1, \ldots, l'_m \mapsto e'_1, \ldots, e'_m}$$

and let $\{j_1, \ldots, j_k\} = \{1, \ldots, m\} - J$, where the order of $j_1, \ldots, j_k$ is arbitrary. Because all primitive operations are deterministic, we know from Lemma 2 that each l-expression (expression) evaluates to a unique l-value (value). Let $w_1, \ldots, w_n$, $v_1, \ldots, v_n$, $w'_1, \ldots, w'_m$, and $v'_1, \ldots, v'_m$, correspond to δ and δ' as shown above. There are two cases.

- Case 1: There is some $j \notin J$ such that

  $$(l_1 \sim l'_j) \vdash_\sigma \texttt{true}$$

  and

  $$\sigma\,(\boxed{l_2, \ldots, l_n \mapsto e_2, \ldots, e_n}\,\overline{\otimes}_{J \cup \{j\}}\, \delta')\,\sigma'.$$

  By induction,

  $$\sigma\,(\boxed{l_2, \ldots, l_n \mapsto e_2, \ldots, e_n} \otimes_{J \cup \{j\}} \delta')\,\sigma'.$$

  Hence, by the definition of $\otimes_{J \cup \{j\}}$ we have that

  - $w_2, \ldots, w_n$ are distinct,
  - $w'_1, \ldots, w'_m$ are distinct,
  - $k \in J \Rightarrow w'_k \notin \{w_2, \ldots, w_n\}$, and
  - $w'_j \notin \{w_2, \ldots, w_n\}$.

  But because P is a symbolic evaluation, we have by Lemma 7 that $w_1 = w'_j$. Hence,

  - $w_1 \notin \{w_2, \ldots, w_n\}$,
  - $k \in J \Rightarrow w'_k \neq w_1$, and
  - an assignment to w'_j overwrites an earlier assignment to w_1.

  Therefore, by the definition of $\otimes_J$,

  $$\sigma\,(\delta \otimes_J \delta')\,\sigma'.$$

- Case 2:

  $$\sigma\,(\boxed{l_2, \ldots, l_n \mapsto e_2, \ldots, e_n}\,\overline{\otimes}_{J \cup \{m+1\}}\,\boxed{l'_1, \ldots, l'_m, l_1 \mapsto e'_1, \ldots, e'_m, e_1})\,\sigma'.$$

  By induction,

  $$\sigma\,(\boxed{l_2, \ldots, l_n \mapsto e_2, \ldots, e_n} \otimes_{J \cup \{m+1\}} \boxed{l'_1, \ldots, l'_m, l_1 \mapsto e'_1, \ldots, e'_m, e_1})\,\sigma'.$$

  Hence, by the definition of $\otimes_{J \cup \{m+1\}}$ we have that

- $w_2, \ldots, w_n$ are distinct,
- $w'_1, \ldots, w'_m, w_1$ are distinct,
- $k \in J \Rightarrow w'_k \notin \{w_2, \ldots, w_n\}$, and
- $w_1 \notin \{w_2, \ldots, w_n\}$.

Therefore, because the assignment to w_1 does not overwrite any preceding assignment, it may be moved to the front, and hence by the definition of $\otimes_J$,

$$\sigma\,(\delta \otimes_J \delta')\,\sigma'.$$

□

3.6 The Composition Operation

Finally, we are ready to present the syntactic composition operation

$$\oplus \quad \in \quad \mathsf{TR} \times \mathsf{TR} \to \mathsf{TR}$$

Definition 9 (Syntactic composition of transfer relations) *We introduce the following terms to describe correctness properties of* $\oplus$.

- *If*
$$\sigma\,(\Delta; \Delta')\,\sigma' \Rightarrow \sigma\,(\Delta \oplus \Delta')\,\sigma',$$
then $\oplus$ *is said to be an* upper approximation.
- *If*
$$\sigma\,(\Delta; \Delta')\,\sigma' \iff \sigma\,(\Delta \oplus \Delta')\,\sigma',$$
then $\oplus$ *is said to be a* translation.

The definition of the $\oplus$ algorithm is as follows.

$$\begin{array}{rcl@{\qquad}c@{\quad}l}
\emptyset & \oplus & \Delta & = & \emptyset \\
\Delta & \oplus & \emptyset & = & \emptyset \\
\boxed{e?\ \Delta \,|\, \Delta'} & \oplus & \Delta'' & = & \mathsf{C}\,e\,(\Delta \oplus \Delta'')\,(\Delta' \oplus \Delta'') \\
\delta & \oplus & \boxed{e?\ \Delta \,|\, \Delta'} & = & \mathsf{C}\,(\mathsf{E}\,e\,\delta)\,(\delta \oplus \Delta)\,(\delta \oplus \Delta') \\
\delta & \oplus & \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n} & = & \delta\,\overline{\otimes}_\emptyset \boxed{\mathsf{L}\,l_1\,\delta, \ldots, \mathsf{L}\,l_n\,\delta \mapsto \mathsf{E}\,e_1\,\delta, \ldots, \mathsf{E}\,e_n\,\delta}
\end{array}$$

We have shown that if the P and C algorithms terminate then E, L, and $\overline{\otimes}$ algorithms terminate. So because $\oplus$ is defined by structural induction, it thus always terminates.

We will give some examples of the composition algorithm later when we use transfer relations to model the semantics of programming languages. For the remainder of this chapter we give the correctness proofs of $\oplus$.

Theorem 3 ($\oplus$ as an upper approximation) *If P is a symbolic evaluation and C is an upper approximation then $\oplus$ is an upper approximation.*

Proof: Because P is a symbolic evaluation, we have from Theorem 1 that E and L are upper approximations. We proceed by structural induction. There are five cases.

- $(\emptyset; \Delta) \equiv \emptyset = (\emptyset \oplus \Delta)$
- $(\Delta; \emptyset) \equiv \emptyset = (\Delta \oplus \emptyset)$
- $(\boxed{e?\ \Delta \mid \Delta'}; \Delta'')$:

$$\begin{array}{lll}
 & \sigma\,(\boxed{e?\ \Delta \mid \Delta'}; \Delta'')\,\sigma'' & \\
\Rightarrow & \exists\sigma'.\,[\sigma \boxed{e?\ \Delta \mid \Delta'} \sigma' \wedge \sigma'\,\Delta''\,\sigma''] & \text{relation composition} \\
\Rightarrow & \exists\sigma'.[((e \vdash_\sigma \texttt{true} \wedge \sigma\,\Delta\,\sigma') & \\
 & \qquad \vee\ (e \vdash_\sigma \texttt{false} \wedge \sigma\,\Delta'\,\sigma')) \wedge \sigma'\,\Delta''\,\sigma''] & \text{definition of conditional relations} \\
\Rightarrow & (e \vdash_\sigma \texttt{true} \wedge \sigma\,(\Delta; \Delta'')\,\sigma'') & \\
 & \quad \vee\ (e \vdash_\sigma \texttt{false} \wedge \sigma\,(\Delta'; \Delta'')\,\sigma'') & \text{relation composition} \\
\Rightarrow & (e \vdash_\sigma \texttt{true} \wedge \sigma\,(\Delta \oplus \Delta'')\,\sigma'') & \\
 & \quad \vee\ (e \vdash_\sigma \texttt{false} \wedge \sigma\,(\Delta' \oplus \Delta'')\,\sigma'') & \text{induction} \\
\Rightarrow & \sigma \boxed{e?\ (\Delta \oplus \Delta'') \mid (\Delta' \oplus \Delta'')} \sigma'' & \text{definition of conditional relations} \\
\Rightarrow & \sigma\,(\mathsf{C}\,e\,(\Delta \oplus \Delta'')\,(\Delta' \oplus \Delta''))\,\sigma'' & \text{assumption about } \mathsf{C} \\
\Rightarrow & \sigma\,(\boxed{e?\ \Delta \mid \Delta'} \oplus \Delta'')\,\sigma'' & \text{definition of } \oplus
\end{array}$$

- $(\delta; \boxed{e?\ \Delta \mid \Delta'})$: Let $e' = (\mathsf{E}\,e\,\delta)$.

$$\begin{array}{lll}
 & \sigma\,(\delta; \boxed{e?\ \Delta \mid \Delta'})\,\sigma'' & \\
\Rightarrow & \exists\sigma'.\,[\sigma\,\delta\,\sigma' \wedge \sigma' \boxed{e?\ \Delta \mid \Delta'} \sigma''] & \text{relation composition} \\
\Rightarrow & \exists\sigma'.\,[\sigma\,\delta\,\sigma' \wedge ((e \vdash_{\sigma'} \texttt{true} \wedge \sigma'\,\Delta\,\sigma'') & \\
 & \qquad\qquad \vee\ (e \vdash_{\sigma'} \texttt{false} \wedge \sigma'\,\Delta'\,\sigma''))] & \text{definition of conditional relations} \\
\Rightarrow & \exists\sigma'.\,[\sigma\,\delta\,\sigma' \wedge ((e' \vdash_\sigma \texttt{true} \wedge \sigma'\,\Delta\,\sigma'') & \\
 & \qquad\qquad \vee\ (e' \vdash_\sigma \texttt{false} \wedge \sigma'\,\Delta'\,\sigma''))] & \mathsf{E} \text{ is an upper approximation} \\
\Rightarrow & ((e' \vdash_\sigma \texttt{true} \wedge \sigma\,(\delta; \Delta)\,\sigma'') & \\
 & \quad \vee\ (e' \vdash_\sigma \texttt{false} \wedge \sigma\,(\delta; \Delta')\,\sigma'')) & \text{relation composition} \\
\Rightarrow & ((e' \vdash_\sigma \texttt{true} \wedge \sigma\,(\delta \oplus \Delta)\,\sigma'') & \\
 & \quad \vee\ (e' \vdash_\sigma \texttt{false} \wedge \sigma\,(\delta \oplus \Delta')\,\sigma'')) & \text{induction} \\
\Rightarrow & \sigma \boxed{e'?\ (\delta \oplus \Delta) \mid (\delta \oplus \Delta')} \sigma'' & \text{definition of conditional relations} \\
\Rightarrow & \sigma\,(\mathsf{C}\,e'\,(\delta \oplus \Delta)\,(\delta \oplus \Delta'))\,\sigma'' & \text{assumption about } \mathsf{C} \\
\Rightarrow & \sigma\,(\delta \oplus \boxed{e?\ \Delta \mid \Delta'})\,\sigma'' & \text{definition of } \oplus
\end{array}$$

- $(\delta; \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n})$: For $i \in \{1, \ldots, n\}$ let $l'_i = (\mathsf{L}\, l_i\, \delta)$ and $e'_i = (\mathsf{E}\, e_i\, \delta)$.

$$
\begin{array}{lll}
 & \sigma\, (\delta; \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n})\, \sigma'' & \\
\Rightarrow & \exists \sigma'.\, [\sigma\, \delta\, \sigma' \wedge \sigma' \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n}\, \sigma''] & \text{relation composition} \\
\Rightarrow & \exists \sigma'.\, [\sigma\, \delta\, \sigma' \wedge \exists w_1, \ldots, w_n, v_1, \ldots, v_n.\, [& \\
 & \qquad\qquad (i \neq j \Rightarrow w_i \neq w_j) & \\
 & \qquad\qquad\quad \wedge\, (\bigwedge_{i=1}^{n} l_i \vdash_{\sigma'} w_i \wedge e_i \vdash_{\sigma'} v_i) & \\
 & \qquad\qquad\quad \wedge\, \sigma'' = \sigma'[w_1 \mapsto v_1] \ldots [w_n \mapsto v_n]]] & \text{definition of assignment relations} \\
\Rightarrow & \exists \sigma'.\, [\sigma\, \delta\, \sigma' \wedge \exists w_1, \ldots, w_n, v_1, \ldots, v_n.\, [& \\
 & \qquad\qquad (i \neq j \Rightarrow w_i \neq w_j) & \\
 & \qquad\qquad\quad \wedge\, (\bigwedge_{i=1}^{n} l'_i \vdash_{\sigma} w_i \wedge e'_i \vdash_{\sigma} v_i) & \\
 & \qquad\qquad\quad \wedge\, \sigma'' = \sigma'[w_1 \mapsto v_1] \ldots [w_n \mapsto v_n]]] & \mathsf{E} \text{ and } \mathsf{L} \text{ are upper approximations} \\
\Rightarrow & \sigma\, (\delta \otimes \boxed{l'_1, \ldots, l'_n \mapsto e'_1, \ldots, e'_n})\, \sigma'' & \text{definition of } \otimes \\
\Rightarrow & \sigma\, (\delta\, \overline{\otimes}_{\emptyset} \boxed{l'_1, \ldots, l'_n \mapsto e'_1, \ldots, e'_n})\, \sigma'' & \text{Lemma 8} \\
\Rightarrow & \sigma\, (\delta \oplus \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n})\, \sigma'' & \text{definition of } \oplus
\end{array}
$$

□

Theorem 4 ($\oplus$ as a translation) *If all primitive operations $p \in$ Primop are deterministic, P is a symbolic evaluation, and C is a translation, then $\oplus$ is a translation.*

Proof: We know from Theorem 3 that $\oplus$ is an upper approximation. Therefore, from Corollary 1, we need only show that

$$\sigma\, (\Delta \,\oplus\, \Delta')\, \sigma' \Rightarrow \exists \sigma''.\, \sigma\, (\Delta; \Delta')\, \sigma''$$

to establish that $\oplus$ is a translation. From Theorem 2 we have that E and L are translations. We proceed by structural induction. There are five cases.

- $(\emptyset \,\oplus\, \Delta) = \emptyset$, and so $\sigma\, (\emptyset \,\oplus\, \Delta)\, \sigma'$ must be false.

- $(\Delta \,\oplus\, \emptyset) = \emptyset$, and so $\sigma\, (\Delta \,\oplus\, \emptyset)\, \sigma'$ must be false.

- $(\boxed{e?\ \Delta\,|\,\Delta'} \oplus \Delta'')$:

$$\begin{array}{lll}
 & \sigma\,(\boxed{e?\ \Delta\,|\,\Delta'} \oplus \Delta'')\,\sigma' & \\
\Rightarrow & \sigma\,(\mathsf{C}\,e\,(\Delta \oplus \Delta'')\,(\Delta' \oplus \Delta''))\,\sigma' & \text{definition of } \oplus \\
\Rightarrow & \sigma\,\boxed{e?\ (\Delta \oplus \Delta'')\,|\,(\Delta' \oplus \Delta'')}\,\sigma' & \mathsf{C} \text{ is a translation} \\
\Rightarrow & (e \vdash_\sigma \texttt{true} \wedge \sigma\,(\Delta \oplus \Delta'')\,\sigma') & \\
 & \quad \vee (e \vdash_\sigma \texttt{false} \wedge \sigma\,(\Delta' \oplus \Delta'')\,\sigma') & \text{defn. of conditional relations} \\
\Rightarrow & (e \vdash_\sigma \texttt{true} \wedge \exists\sigma''.\ \sigma\,(\Delta;\Delta'')\,\sigma'') & \\
 & \quad \vee (e \vdash_\sigma \texttt{false} \wedge \exists\sigma''.\ \sigma\,(\Delta';\Delta'')\,\sigma'' & \text{induction} \\
\Rightarrow & (e \vdash_\sigma \texttt{true} \wedge \exists\sigma'.\ \sigma\,\Delta\,\sigma' \wedge \exists\sigma''\ \sigma'\,\Delta''\,\sigma'') & \\
 & \quad \vee (e \vdash_\sigma \texttt{false} \wedge \exists\sigma'.\ \sigma\,\Delta'\,\sigma' \wedge \exists\sigma''\ \sigma'\,\Delta''\,\sigma'') & \text{relation composition} \\
\Rightarrow & \exists\sigma',\sigma''.\ [((e \vdash_\sigma \texttt{true} \wedge \sigma\,\Delta\,\sigma') & \\
 & \qquad\qquad \vee (e \vdash_\sigma \texttt{false} \wedge \sigma\,\Delta'\,\sigma')) \wedge \sigma'\,\Delta''\,\sigma''] & \text{distributivity} \\
\Rightarrow & \exists\sigma',\sigma''.\ [\sigma\,\boxed{e?\ \Delta\,|\,\Delta'}\,\sigma' \wedge \sigma'\,\Delta''\,\sigma''] & \text{defn. of conditional relations} \\
\Rightarrow & \exists\sigma''.\ \sigma\,(\boxed{e?\ \Delta\,|\,\Delta'}\,;\Delta'')\,\sigma'' & \text{relation composition}
\end{array}$$

- $(\delta \oplus \boxed{e?\ \Delta\,|\,\Delta'})$: Let $e' = (\mathsf{E}\,e\,\delta)$.

$$\begin{array}{lll}
 & \sigma\,(\delta \oplus \boxed{e?\ \Delta\,|\,\Delta'})\,\sigma' & \\
\Rightarrow & \sigma\,(\mathsf{C}\,e'\,(\delta \oplus \Delta)\,(\delta \oplus \Delta'))\,\sigma' & \text{definition of } \oplus \\
\Rightarrow & \sigma\,\boxed{e'?\ (\delta \oplus \Delta)\,|\,(\delta \oplus \Delta')}\,\sigma' & \mathsf{C} \text{ is a translation} \\
\Rightarrow & (e' \vdash_\sigma \texttt{true} \wedge \sigma\,(\delta \oplus \Delta)\,\sigma') & \\
 & \quad \vee (e' \vdash_\sigma \texttt{false} \wedge \sigma\,(\delta \oplus \Delta')\,\sigma') & \text{defn. of conditional relations} \\
\Rightarrow & (e' \vdash_\sigma \texttt{true} \wedge \exists\sigma''.\ \sigma\,(\delta;\Delta)\,\sigma'') & \\
 & \quad \vee (e' \vdash_\sigma \texttt{false} \wedge \exists\sigma''.\ \sigma\,(\delta;\Delta')\,\sigma'') & \text{induction} \\
\Rightarrow & \exists\sigma',\sigma''.\ [\sigma\,\delta\,\sigma' \wedge ((e' \vdash_\sigma \texttt{true} \wedge \sigma'\,\Delta\,\sigma'') & \\
 & \qquad\qquad \vee (e' \vdash_\sigma \texttt{false} \wedge \sigma'\,\Delta'\,\sigma''))] & \text{relation composition} \\
\Rightarrow & \exists\sigma'.\sigma''.\ [\sigma\,\delta\,\sigma' \wedge ((e \vdash_{\sigma'} \texttt{true} \wedge \sigma'\,\Delta\,\sigma'') & \\
 & \qquad\qquad \vee (e \vdash_{\sigma'} \texttt{false} \wedge \sigma'\,\Delta'\,\sigma''))] & \mathsf{E} \text{ is a translation} \\
\Rightarrow & \exists\sigma'.\sigma''.\ [\sigma\,\delta\,\sigma' \wedge \sigma'\,\boxed{e?\ \Delta\,|\,\Delta'}\,\sigma''] & \text{defn. of conditional relations} \\
\Rightarrow & \exists\sigma''.\ \sigma\,(\delta;\boxed{e?\ \Delta\,|\,\Delta'})\,\sigma'' & \text{relation composition}
\end{array}$$

- $(\delta \oplus \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n})$: For $i \in \{1, \ldots, n\}$ let $l'_i = (\mathsf{L}\, l_i\, \delta)$ and $e'_i = (\mathsf{E}\, e_i\, \delta)$.

$$\begin{array}{lll}
 & \sigma\,(\delta \oplus \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n})\,\sigma' & \\
\Rightarrow & \sigma\,(\delta\,\overline{\otimes}_{\emptyset} \boxed{l'_1, \ldots, l'_n \mapsto e'_1, \ldots, e'_n})\,\sigma' & \text{definition of } \oplus \\
\Rightarrow & \sigma\,(\delta \otimes_{\emptyset} \boxed{l'_1, \ldots, l'_n \mapsto e'_1, \ldots, e'_n})\,\sigma' & \text{Lemma 9} \\
\Rightarrow & \exists \sigma', w_1, \ldots, w_n. & \\
 & \qquad [\sigma\,\delta\,\sigma' \wedge (i \neq j \Rightarrow w_i \neq w_j) \wedge \bigwedge_{i=1}^{n} l'_i \vdash_{\sigma} w_i] & \text{definition of } \otimes \\
\Rightarrow & \exists \sigma', w_1, \ldots, w_n. & \\
 & \qquad [\sigma\,\delta\,\sigma' \wedge (i \neq j \Rightarrow w_i \neq w_j) \wedge \bigwedge_{i=1}^{n} l_i \vdash_{\sigma'} w_i] & \mathsf{L} \text{ is a translation} \\
\Rightarrow & \exists \sigma', \sigma''. \, [\sigma\,\delta\,\sigma' \wedge \sigma' \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n}\,\sigma''] & \text{definition of assignment relations} \\
\Rightarrow & \exists \sigma''. \, \sigma\,(\delta; \boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n})\,\sigma'' & \text{relation composition}
\end{array}$$

□

Chapter 4

Semantics via Transfer Relations

In Chapter 2, we introduced the store as an object for modeling the state of memory during a point of program execution. We also introduced the notion of relating a store at one point in an execution to some later point in the execution; these relations are called transfer relations. For the sake of automatic program analysis, we developed in Chapter 3 a computer representation for the kinds of transfer relations that might naturally correspond to such execution segments, and we gave an algorithm for composing these representations, to build bigger execution segments out of smaller ones.

However, those chapters presented these concepts in an abstract manner, apart from any particular programming language. Although we gave examples designed to spark intuition about actual programming languages, we never described how these transfer relations correspond to any kind of a semantics of a programming language. In this chapter, we describe a semantic methodology of programming languages that is founded upon transfer relations, and as such is particularly useful as a basis for program analysis.

4.1 Denotational and Operational Semantics

The semantics of programming languages is a topic both broad and deep, and we can only touch on some of the overarching issues here, in order to put our work in a larger perspective. A denotational semantics [Sto77] uses structural induction to assign each term in the source language an object in some abstract model. The spirit of denotational semantics is to model function terms in the source language with actual functions. This turns out to be difficult; Dana Scott solved the underlying problems [Sco70, Sco76, Sco82]. On the other hand, Jean-Yves Girard in [GLT89] makes the following philosophical observation about the α, β, and η equations of λ-calculus [Bar84]:

> In fact, these equations may be read in two different ways, which re-iterate the dichotomy [in logic] between sense and denotation:

> - as the *equations* which define the equality of terms, in other words the equality of denotations (the *static* viewpoint).
> - as *rewrite* rules which allows us to calculate terms by reduction to a normal form. That is an operational, *dynamic* viewpoint, the only truly fruitful view for this aspect of logic.
>
> Of course the second viewpoint is under-developed by comparison with the first one, as was the case in Logic! For example *denotational* semantics of programs (Scott's semantics, for example) abound: for this kind of semantics, nothing changes throughout the execution of a program. On the other hand, there is hardly any civilised *operational* semantics of programs (we exclude the *ad hoc* semantics which crudely paraphrase the steps toward normalisation). The establishment of a truly operational semantics of algorithms is perhaps the most important problem in computer science.

The dichotomy between sense and denotation in logic to which Girard refers is the comparison between Tarski's classical view, in which for instance the meaning of $A \wedge B$ is its truth value and is given by a truth table on the meanings of A and B, and Heyting's intuitionistic view, in which the meaning of $A \wedge B$ is a proof and is given by a proof of A coupled with a proof of B. This leads to the study of proof theory, of which the most famous result is the Curry-Howard isomorphism between natural deduction and the simply-typed λ-calculus in which types correspond to sentences, terms correspond to proofs (meanings, in the Heyting view), and reduction of terms corresponds to rewriting of proofs. Girard points out that "the fundamental idea of denotational semantics is to interpret reduction (a dynamic notion) by equality (a static notion)".

This discussion provides some insight into the role of semantics in program analysis. Strictly speaking, a program analysis does not answer questions about a program, it answers questions about a program's semantics. This is a rather specialized and practical application of semantics. As Girard points out, denotational semantics is concerned with the equality of programs—a notion that is undecidable for most languages. One of the practical benefits of a well-designed denotational semantics is to shed light upon or otherwise aid in the reasoning about program equivalence. It stands to reason, then, that the purpose of a program analysis based on a denotational semantics must be to provide some automatic support for reasoning about program equivalence.

Almost as soon as abstract interpretation arrived on the scene, to make a connection between program analyses and the semantics of programming languages, a great amount of effort was spent in adapting it to denotational semantics. For some examples, see [Myc81], [Nie84], [Nie86], and [AH87]. As one would expect, this body of work offers some of the most esthetically pleasing formulations of program analyses, but it also has found little use beyond a narrow range of applications such as strictness analysis [BHA86].

In general, however, one would like a program analysis to produce some information about a *dynamic* interpretation of a program rather than this static denotation. This is why most program analyses are based on an operational semantics that describes how a program *reduces* during execution.

This is also perhaps why the field of program analysis continues to struggle for acceptance in programming-language theory. As Girard says, operational semantics tend to be "uncivilised", and despite the frameworks of structural operational semantics [Plo81] (generalized to infinite behaviors in [CC92b]) and natural semantics [Kah87], operational semantics does not have nearly the developed and refined theory of denotational semantics. Even worse, despite an effort by Schmidt in [Sch95] to begin to develop a sub-framework of abstract interpretation for natural semantics, most program analyses use what Girard calls the "*ad hoc* semantics that crudely paraphrase the steps toward normalisation". The reason is that a program analysis is usually designed to answer questions about "the run-time behavior" of a program, which requires this crude notion of operational semantics: ad hoc because it is modeling the execution of a program on some kind of machine, and paraphrasing normalization steps because they are precisely the steps of execution on this machine.

Therefore, whether they are presented in this manner or not, most useful program analyses are founded upon semantics based on *transition systems.* These semantics mimic the execution of a program on a particular abstract machine that reflects the properties of interest. In our work, the store is the heart of such an abstract machine. We designed the store with this application in mind.

4.2 Modeling a Program as a Transition System

Typically, a dynamic semantics of a programming language must model two components of execution: data and control. By *data*, we mean the state of memory. In our framework, the data is modeled by a store. By *control*, we mean the state of the code itself. For instance, the control might be modeled by a label describing the position in the code that is scheduled to be executed next.

In some operational semantics, the control state and the data state are intertwined. For instance, in *context semantics* [FF86], a state of execution is simply a syntactic term; the control state is encoded in as the next redex to be reduced, and the data state is modeled with syntactic constructs (such as substitution or the *heap variables* in [MFH95]) and folded into the term itself. But much of program analysis is concerned with analyzing the patterns of data access during execution, and so we wish to keep control and data explicitly separated.

This inspires a semantic methodology in which a program is modeled by a *transition system.* As described in Chapter 2, given a set Var of variables and a set Val of values, one can define the set Store of stores that model the instantaneous states of data. We must introduce a new set $\mathsf{CtrlPoint}$ of control points, such as labels, that model the instantaneous syntactic position of execution. A transition system is then a tuple

$$\langle \mathsf{CtrlPoint}, \mathsf{Var}, \mathsf{Val}, \longmapsto \rangle$$

where

$$\longmapsto \subseteq (\mathsf{CtrlPoint} \times \mathsf{Store}) \times (\mathsf{CtrlPoint} \times \mathsf{Store})$$

is a single-step binary transition relation between adjacent control-store pairs in an execution, where Store is defined from Var and Val as in Chapter 2:

$$\begin{array}{lcll} \mathsf{Store} & = & \mathsf{Lval} \rightarrow \mathsf{Val} & \\ \mathsf{Lval} & = & \mathsf{Var} \cup (\mathsf{Val} \times \mathsf{Val}) & \quad \text{l-values} \end{array}$$

In other words,

$$(C, \sigma) \longmapsto (C', \sigma')$$

if execution can proceed in one step from a state at control point $C \in \mathsf{CtrlPoint}$ and store $\sigma \in \mathsf{Store}$ to a state at control point C' and store σ'.

As we described in the Chapter 2, stores have rich structure for analysis. A control point is typically much simpler. For instance, if the subterms (e.g., commands, expressions) in a program are uniquely labeled, then a control point often can be simply the label of the next subterm to be executed. Or the control point might be the unlabeled subterm itself. As another example, the control point of a machine-language program is the value of the program counter, while the rest of the registers and memory are modeled by the store.

There are many examples of other kinds of transition systems as models of programming languages. Most of these systems do not use our precise notions of control and store, but they all have some notion of a control state and a data state. These systems are also called *abstract machines*. Our notion of a store is expressive enough to encompass all of these, with the appropriate choice of the set Val of values.

The heart of a transition system is the definition of the transition relation $\longmapsto$. Almost always, $\longmapsto$ is defined by a set of *meta-rules* that define how each occurrence of a certain kind of syntactic term in the program induces a family or families of transitions. We give a concrete example of this in Section 5.8.2, and the rules in that section are indeed quite standard. But for the rest of this chapter we instead describe a different approach. This novel approach replaces the traditional meta-rules with our computer-representable transfer relations, thus opening the door to a wide range of program-analysis possibilities.

4.3 Modeling a Program as a Table of Transfer Relations

In the previous section, we suggested modeling the semantics of a program with a transition system. We explained that a transition system is a tuple

$$\langle \mathsf{CtrlPoint}, \mathsf{Var}, \mathsf{Val}, \longmapsto \rangle$$

where

$$\longmapsto \subseteq (\mathsf{CtrlPoint} \times \mathsf{Store}) \times (\mathsf{CtrlPoint} \times \mathsf{Store})$$

is a single-step binary transition relation between adjacent control-store pairs in an execution, where Store is defined from Var and Val as in Chapter 2.

Observe, however, the following isomorphism:

$$\mathcal{P}((\mathsf{CtrlPoint} \times \mathsf{Store}) \times (\mathsf{CtrlPoint} \times \mathsf{Store})) \simeq \mathsf{CtrlPoint} \times \mathsf{CtrlPoint} \to \mathcal{P}(\mathsf{Store} \times \mathsf{Store})$$

Therefore, given any transition relation

$$\longmapsto \subseteq (\mathsf{CtrlPoint} \times \mathsf{Store}) \times (\mathsf{CtrlPoint} \times \mathsf{Store})$$

one can view $\longmapsto$ as a table of binary relations on stores, indexed by pairs of program points. If we write the (C, C') entry of this table as $\overset{C,C'}{\longmapsto}$ then the correspondence is as follows.

$$(C, \sigma) \longmapsto (C', \sigma') \quad \text{iff} \quad \sigma \overset{C,C'}{\longmapsto} \sigma'.$$

Now, recall that transfer relations are binary relations on stores:

$$\mathsf{TR} \subseteq \mathcal{P}(\mathsf{Store} \times \mathsf{Store})$$

But not all binary relations on stores are transfer relations. In Section 2.4.1, we argued that some binary relations on stores are not "natural", in that they will not occur in any reasonable programming language. This notion of naturalness motivated the design of our language TR of transfer relations, and our claim is that TR is indeed rich enough to model programming languages.

The implications of that claim now become manifest. We now claim not only that a transition relation $\longmapsto$ can be replaced by a table in

$$\mathsf{CtrlPoint} \times \mathsf{CtrlPoint} \to \mathcal{P}(\mathsf{Store} \times \mathsf{Store})$$

of binary relations on stores, but that it can indeed be replaced by a table in

$$\mathsf{CtrlPoint} \times \mathsf{CtrlPoint} \to \mathsf{TR}$$

of terms in our language of transfer relations, again indexed by the control points before and after the transition. Ultimately, this is more of a philosophical claim than a provable statement. The claim is that the language TR of transfer relations is expressive enough to model all possible store changes that may arise as single execution steps of any reasonable programming language. To support this claim, we will demonstrate in future chapters that TR is indeed expressive enough to model a wide variety of programming-language constructs.

Given this claim, one may replace any transition system

$$\langle \mathsf{CtrlPoint}, \mathsf{Var}, \mathsf{Val}, \longmapsto \rangle$$

by a tuple

$$\langle \mathsf{CtrlPoint}, \mathsf{Var}, \mathsf{Val}, \mathsf{Primop}, \vec{\Delta} \rangle$$

where

$$\vec{\Delta} \in \mathsf{CtrlPoint} \times \mathsf{CtrlPoint} \to \mathsf{TR}$$

and the set TR of transfer relations is defined as in Chapter 2 from the sets Var, Val, and Primop. Now, the heart of a semantic definition of a programming language is not a set of meta-rules defining a transition relation $\longmapsto$, but is instead a description of how to map a program in the language to a table $\vec{\Delta}$ of transfer relations, one transfer relation for each pair of control points in the program, describing the single steps of program execution. By convention, we write $\vec{\Delta}(C, C')$ as

$$\Delta_{C,C'}$$

and call these transfer relations the *single-step transfer relations* of the program.

There are $|\mathsf{CtrlPoint}|^2$ single-step transfer relations. As described above, $\mathsf{CtrlPoint}$ is typically a set of pointers into the text of a program P, and so $|\mathsf{CtrlPoint}|$ will usually be linear with the size of P. If that size is n, then this means that there are $O(n^2)$ single-step transfer relations in the semantics of P. However, the vast majority of these will be the empty relation, because transitions between most pairs of control points is impossible. For instance, in straight-line code, the only possible transitions are between adjacent control points, and so there are only $O(n)$ non-empty single-step transfer relations, as one would expect.

Note that transfer relations thus encode control-flow information about the program. If $\Delta_{C,C'} = \emptyset$ then it is not possible for the program in question to take a single step from control path C to control path C'. Similarly, if $\Delta_{C,C'} = \boxed{e?\ \Delta}$ then a single step from C to C' is possible only from stores at C in which e evaluates to `true`.

It is interesting to note that *each* non-empty single-step transfer relation replaces an *infinite* family of transitions. This is demonstrated with the following simple example.

Example 19 *Suppose the transfer-relation semantics of program P is the tuple*

$$\langle \mathsf{CtrlPoint}, \mathsf{Var}, \mathsf{Val}, \mathsf{Primop}, \vec{\Delta} \rangle$$

and that $\Delta_{C,C'} = \boxed{\mathtt{x} \mapsto \mathtt{y + 1}}$ *for some* $C, C' \in \mathsf{CtrlPoint}$. *Then the transition-system semantics*

$$\langle \mathsf{CtrlPoint}, \mathsf{Var}, \mathsf{Val}, \longmapsto \rangle$$

of P includes the family of transitions

$$(C, \sigma) \longmapsto (C', (\sigma[\mathtt{x} \mapsto (\sigma\, \mathtt{y}) + 1]))$$

where $\sigma \in \mathsf{Store}$ *is any store.*

4.4 Composing Single-Step Transfer Relations

Given the single-step transfer relations $\vec{\Delta}$ for a program, one can use the transfer-relation composition algorithm $\oplus$ from Chapter 3 to compute the transfer relation for any finite control path in a program.

4.4.1 Two-step transition sequences

Suppose the transfer-relation semantics of program P is the tuple

$$\langle \mathsf{CtrlPoint}, \mathsf{Var}, \mathsf{Val}, \mathsf{Primop}, \vec{\Delta} \rangle$$

and, following our convention of notation, we write $\vec{\Delta}(C, C')$ as $\Delta_{C,C'}$.

The single-step transfer relation $\Delta_{C,C'}$ defines all of the transitions from control point C to control point C'. The single-step transfer relation $\Delta_{C',C''}$ defines all of the transitions from control point C' to control point C''. Therefore, the relation

$$\Delta_{C,C'} ; \Delta_{C',C''}$$

defines exactly the two-step transition sequences from C through C' to C''.

Suppose that all primitive operations $p \in \mathsf{Primop}$ are deterministic, as will be the case in most programming languages, and that we are given the symbolic evaluation

$$\mathsf{P} \quad \in \quad \mathsf{Primop} \to \mathsf{Exp}^* \to \mathsf{ATR} \to \mathsf{Exp}$$

as defined by Definition 5 and translation

$$\mathsf{C} \quad \in \quad \mathsf{Exp} \to \mathsf{TR} \to \mathsf{TR} \to \mathsf{TR}$$

as defined by Definition 8. Then we know from Theorem 4 that

$$(\Delta_{C,C'} \oplus \Delta_{C',C''}) \in \mathsf{TR}$$

is equivalent to $\Delta_{C,C'} ; \Delta_{C',C''}$ and thus defines exactly the two-step transition sequences from C through C' to C''. But the profound advantage of $\Delta_{C,C'} \oplus \Delta_{C',C''}$ over $\Delta_{C,C'} ; \Delta_{C',C''}$ is that, *syntactically*, the former is a (computer-representable) term in our language TR of transfer relations, while the latter is not. The advantage of this is illustrated in the following example, which also serves to remind why we disallowed explicit *syntactic* composition of transfer relations in the language TR.

Example 20 *Suppose*

$$\begin{aligned} \Delta_{C,C'} &= \boxed{\mathtt{x} \mapsto \mathtt{y + 1}} \\ \Delta_{C',C''} &= \boxed{\mathtt{x} \mapsto \mathtt{x - 1}} \end{aligned}$$

Suppose also that all primitive operations $p \in \mathsf{Primop}$ are deterministic, and $\{\mathtt{+}, \mathtt{-}\} \subseteq \mathsf{Primop}$. Assuming an arbitrary C (because it will not be used here) and the trivial P that is the identity function on context-independent operations (such as `+` and `-`), we have that

$$\boxed{\mathtt{x} \mapsto \mathtt{y + 1}} \oplus \boxed{\mathtt{x} \mapsto \mathtt{x - 1}} = \boxed{\mathtt{x} \mapsto \mathtt{(y + 1) - 1}}$$

describes exactly the possible net effects of the two-step fragment of execution from control point C through C' to C''. If P *instead performs some better symbolic evaluation,* $\oplus$ *may yield a better result such as*

$$\boxed{\texttt{x} \mapsto \texttt{y}}$$

Either way, it is more enlightening as to the behavior of this execution fragment than the term

$$\boxed{\texttt{x} \mapsto \texttt{y + 1}} ; \boxed{\texttt{x} \mapsto \texttt{x - 1}}$$

By convention we write $\Delta_{C,C'} \oplus \Delta_{C',C''}$ as $\Delta_{C,C',C''}$

Suppose that there are nondeterministic primitive operations in Primop. Then from Theorem 3 we have that $\Delta_{C,C'} \oplus \Delta_{C',C''}$ is a superset of $\Delta_{C,C'} ; \Delta_{C',C''}$ and thus defines *at least* all of the two-step transition sequences from C through C' to C''. It might, however, relate some initial stores (i.e., at C) to some final stores (i.e., at C'') that cannot be achieved by such a two-step transition sequence.

We now generalize the above to arbitrary-length sequences of transitions.

4.4.2 Arbitrary-length transition sequences

Single transitions involve two control points, an initial and a final. Two-step transition sequences involve three control points, an initial, a middle, and a final. We generalize this concept with the following definition.

Definition 10 *Given a set* CtrlPoint *of control points, a* control path *is a sequence of one or more control points. The set of control paths is written* $\mathsf{CtrlPoint}^+$*. If* $\Gamma \in \mathsf{CtrlPoint}^+$ *and* $\Gamma' \in \mathsf{CtrlPoint}^+$ *then* $\Gamma, \Gamma' \in \mathsf{CtrlPoint}^+$ *is their concatenation. For any* $\Gamma \in \mathsf{CtrlPoint}^+$*,* $\Gamma' \in \mathsf{CtrlPoint}^+$*, and* $C \in \mathsf{CtrlPoint}$*,* $(\Gamma, C); (C, \Gamma') = \Gamma, C, \Gamma'$*.*

A transfer-relation semantics provides the single-step transfer relations of a program that define all of the program's valid transitions. Transitions are merely execution sequences through control paths of length two. Concatenating adjacent transitions, or equivalently, composing adjacent transfer relations, produces the execution sequences through control paths of length three. Another composition covers the control paths of length four, and so forth.

Therefore we define from the finite collection of single-step transfer relations $\Delta_{C,C'}$ the infinite collection of transfer relations Δ_Γ for all control paths Γ of length at least 2:

$$\Delta_{\Gamma;\Gamma'} = \Delta_\Gamma \oplus \Delta_{\Gamma'}$$

Note that this definition is nondeterministic, because there are $n-2$ ways to split up a length-n control path Γ'' into Γ and Γ' such that $\Gamma'' = \Gamma; \Gamma'$ because any control point in the path Γ'' other than the end points can act as the "pivot point". In fact, the choice of pivot point will in general produce different syntactic transfer relations. But if $\oplus$ is a translation, which by Theorem 4 will be the case if all primitive operations $p \in \mathsf{Primop}$ are deterministic, then by associativity of relation composition, all of these transfer relations are semantically equivalent. If on the other hand $\oplus$ is merely an upper approximation, then they may not all be semantically equivalent, but they are all supersets of the true composition.

4.5 Treatment of Errors

There are three ways in which an implementation of a programming language treats an error.

1. The error may be caught at compile time. For instance, most languages with static typing, such as ML, will prevent at compile time all attempts to add an integer and a boolean value.

2. An error not caught at compile time may be caught at run time. For instance, ML's static type system cannot detect out-of-bound array references. Instead, all array references perform a test a run time to check if the index is within bounds of the array, and raise an exception if the test fails.

3. Finally, an error may not be caught at all. In this case, the semantics of the error is unspecified, and the execution is allowed to "run wild" after the error. In ML, no errors reach this stage; they are all caught either at compile time or at run time. But in C, for instance, it is possible to extract a value from an uninitialized local variable, but the definition of C does not specify this value. Also, one may cast any integer into a pointer and attempt to write into that address in memory. Depending on the implementation, this can produce a wide range of errors that are impossible to catch at run time. For instance, the address of a pointer may happen to correspond to a local variable on the stack, and so any write into a pointer changes the value of the variable. Similarly, writing past the boundary of a struct or array may interfere with other data structures. An even more dramatic example of bad behavior resulting from uncaught errors is the overwriting of code by bad pointers or out-of-bound array references. The semantics of C is unspecified for such programs; once such an error occurs, the execution may run wild. Depending on the implementation, the execution may proceed in an unpredictable manner or may violate the run-time system, causing a segmentation fault or bus error.

We discuss the way in which our methodology addresses these three kinds of errors.

1. Because our methodology is appropriate only for a dynamic semantics of a language (in other words, the run-time behavior) and not for a static semantics (for instance, the static type system), we do not address errors caught at compile time. We assume that the static semantics has already caught these errors and provided the dynamic semantics with a program that is free of these errors. A dynamic semantics may, incidentally, provide a model for programs that contain compile-time errors, but it does not matter what this model is. For example, in Chapters 5 and 6, we will use our methodology to model languages with records. These semantics will not model the type of a record (in other words, the names and types of its fields), and thus allow type-unsafe uses of the record (for instance, an attempt to read a non-existent field). These semantics do provide a model for such type-unsafe operations, but it is expected that a static semantics will ensure that they will never occur.

2. Because our methodology is for the design of a dynamic semantics, modeling the run-time behavior of programs, it should be able to provide a treatment of errors caught at run-time. Typically, when a run-time error is detected, control proceeds to an error handler. For instance, a run-time error in ML raises an exception which is caught by either a user-defined handler or the run-time system's top-level handler. Nothing prevents our methodology from dealing with run-time errors in a similar fashion. Suppose that (C, σ) is a state in which a run-time error occurs. For instance, in an ML program, control point C may reference code to take the head of a list object in store σ, and that object is `nil`. Then there will be a transition

   $$(C, \sigma) \longmapsto (C', \sigma')$$

   where C' is the entry of some error handler, which in the case of ML will be an exception handler.

   However, in this dissertation we will not give any examples of such run-time errors. In other words, the languages in Chapters 5, 6, and 7 do not perform any run-time checks.

3. We assume that if an error is not caught at compile time or run time, then the run-time behavior of the program after the error occurs is unspecified. Therefore, our methodology treats these errors in the same way as it treats compile-time errors: we assign a behavior to a program that exhibits such an error, but the particular behavior is unimportant because conceptually the semantics is unspecified in that case.

Generally, the value `undef` may come about as the result of errors that are allowed to "run wild" and thus have unspecified run-time behavior—in other words, the the first and third categories above. For instance, in Chapters 5, 6, and 7 we will give a semantic model in which an attempt to lookup the value of an unbound variable or field of a data structure results in `undef`, and primitive operations are defined on `undef`. For instance, (1 `+ undef`) evaluates to `undef`, and (`undef = undef`) evaluates to `true`. The latter may seem odd, but is perfectly reasonable because, once again, the run-time behavior of errors that produce undefined is unspecified. Essentially, we need only model the run-time behavior of programs that do not exhibit any errors in the first and third categories above.

It is worth commenting on a phenomenon with transfer relations that should not be confused with the treatment of errors. Suppose control path Γ begins with control point C. Given the transfer relation Δ_Γ corresponding to control path Γ, and given a store σ, if there is no σ' such that $\sigma \, \Delta_\Gamma \, \sigma'$, then it means that execution from the state (C, σ) cannot progress through control path Γ. For instance, C may be a branching point, with path Γ proceeding down the branch for when $x > 0$, but the value of x in σ is not greater than 0. This is *not* intended to be a way of modeling errors that may have occurred during control path Γ.

Part III

Programming Languages

Chapter 5

A Case Study: The Language Mini-C

In this chapter, we present Mini-C, a simple imperative language with while loops, assignment, mutable records, and immutable tuples, but without procedures or arrays. We also give a semantics for Mini-C in terms of a transition system. As suggested in Chapter 4, we will demonstrate two different techniques for defining that transition system—the traditional approach of meta-rules and our new approach using single-step transfer relations.

The main purpose of this chapter is to develop a relatively straightforward case study of our approach to program analysis.

5.1 Syntax

A Mini-C program is a list of zero or more *statements*. A statement is either an assignment, an allocation of a new record with n named fields, a conditional with a statement list for each branch, or a while loop with a statement list for a body.

$$
\begin{array}{rcll}
S & ::= & \{s_1, \ldots, s_n\} & \text{(ordered) statement list } (n \geq 0) \\[1ex]
s & ::= & L := E & \text{assignment statement} \\
 & | & L := \{f_1 = E_1, \ldots, f_n = E_n\} & \text{mutable-record allocation} \\
 & | & \textbf{if } E \textbf{ then } S \textbf{ else } S' & \text{conditional statement} \\
 & | & \textbf{while } E \textbf{ do } S & \text{while loop} \\[1ex]
f & \in & \mathsf{Field} & \text{mutable-record field names}
\end{array}
$$

We define the *source expressions* and *source l-expressions* slightly differently from the expressions and l-expressions in Chapter 2.

$$\begin{array}{lcll}
E & ::= & L & \text{location lookup} \\
 & | & P(E_1, \ldots, E_n) & \text{primitive application} \\[1em]
L & ::= & x & \text{variable location} \\
 & | & E.f & \text{data subcomponent location} \\[1em]
x & \in & \mathsf{Var} & \text{variables}
\end{array}$$

Finally, the *source primitive operations* are

$$\begin{array}{lcll}
P & ::= & c & \text{constants (nullary)} \\
 & | & \texttt{+} \mid \texttt{-} \mid \texttt{*} & \text{integer operations (binary and unary } \texttt{-}\text{)} \\
 & | & \texttt{<} \mid \texttt{>} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{\&} & \text{boolean operations (binary)} \\
 & | & \texttt{if} & \text{conditional expression (ternary)} \\
 & | & \texttt{tuple} & \text{immutable-tuple construction (}n\text{-ary)} \\
 & | & \pi_i & \text{immutable-tuple component selection (unary)}
\end{array}$$

where the set Constant of constants is

$$\begin{array}{lcll}
c & ::= & n & \text{integers } (\mathsf{Int}) \\
 & | & \texttt{true} \mid \texttt{false} & \text{booleans}
\end{array}$$

All of the source primitives are simple (i.e., deterministic and context-independent). We leave the set Field of field names of mutable data structures open for the moment. Note that constants are nullary primitive operations, as suggested in Chapter 2. We will sometimes use `nil` as another name for `false`. (One could just as easily add `nil` as another constant.)

We adopt the following syntactic conventions.

- The statement list $\{s\}$ may be written as s. item The expression $c()$ may be written as c (our usual convention).
- The expression $P(E, E')$ may be written in infix as $E\ P\ E'$.
- The expression $\texttt{tuple}(E_1, \ldots, E_n)$ may be written as $(E_1, \ldots, E_n)$.

5.2 Discussion

Mini-C does not have procedures, but its data features and imperative features of are similar to C. Allocating a Mini-C record with n fields corresponds to calling the C `malloc` operation to allocate a size-n block of memory on the heap and then immediately filling the block with n values. Thus, Mini-C field names correspond to structure field names in C as well as the `*` token for pointers.

For simplicity, Mini-C does not have arrays, which add an extra element of complexity in two ways. One, they may be of statically unknown size. Two, the index (which would correspond to the field name of a record) of an array dereference may also be statically unknown. In Chapter 7 we will develop a source language with arrays.

One feature of C that is not present in Mini-C is low-level control over data layout. C's `&` operator is not in Mini-C because there is no distinction in Mini-C between an updatable data structure and a pointer to that structure. Intuitively, all updatable data structures in Mini-C are pointers, in much the same way that all arrays in C are pointers. In contrast, C provides a mechanism for distinguishing a struct itself from a pointer to that struct; this is useful for programmer control of data layout—for instance, the inline allocation of a struct as a field in another struct. Furthermore, there is no pointer arithmetic in Mini-C, and nor is there a notion of casting one updatable data structure to another. All of those features of C exist to give the programmer low-level control of data layout. In this dissertation, we will not cover such issues, and so Mini-C does not include those language features.

However, it is in fact possible to model a functionality similar to the C `&` operator, as well as pointer arithmetic for an extension of Mini-C with arrays. We discuss this further in Section 5.9.

5.3 Simplification of Syntax

Above we presented the syntax of Mini-C in a form that is intended to be used by a programmer. But it will be much more convenient to describe the semantics of Mini-C programs if we first recast the syntax in a form that more closely fits our development of transfer relations in the previous chapters.

Our first task is to recast source expressions and source l-expressions respectively into the expressions and l-expressions of the transfer-relation language. We recall their definitions here.

$$\begin{array}{lcll} e & \in & \mathsf{Exp} & ::= \; x \mid p(e_1, \ldots, e_n) \\ l & \in & \mathsf{Lexp} & ::= \; x \mid e.e' \\ \\ p & \in & \mathsf{Primop} & \\ x & \in & \mathsf{Var} & \end{array}$$

First of all, we include all of the source primitive operations P in the set Primop. For the purpose of modeling the semantics of Mini-C with transfer relations, we will have to add some operations to Primop that are not available to the user at source level. First of all, we need to add the field names Field to Primop as nullary primitive operations; then the source l-expression $E.f$ is a member of Lexp.

Secondly, we need to add the context-dependent binary primitive operation `deref` of Chapter 2 to Primop in order to consider the source expression $E.f$ to be the expression $\mathtt{deref}(E, f) \in$

Exp. Now all source expressions E are in Exp, and all source l-expressions L are in Lexp; henceforth we will thus call them expressions and l-expressions and use the metavariables e and l, respectively.

Our second transformation is to "compile" a statement allocating a record with n fields into a statement that allocates new memory followed by n statements that fill the n fields with their values. For this purpose, we need the following, as suggested in Chapter 2:

- for each natural number m, a pointer value $\langle m \rangle \in \mathsf{Val}$
- the unary primitive operation `ptr` that casts an integer m to the pointer $\langle m \rangle$ (formally, $\mathtt{ptr}(m) \hookrightarrow \langle m \rangle$), where we write $\langle e \rangle$ for $\mathtt{ptr}(e)$
- a distinguished variable `H`, initialized to 1 at the beginning of execution, to hold the integer of the next free pointer on the heap

We now perform the following transformation of a MINI-C program S. We assume without loss of generality that `H` does not appear in S. We first add the assignment statement

$$\mathtt{H} := 1$$

to the beginning of statement list S. Then we rewrite every allocation statement

$$l := \{f_1 = e_1, \ldots, f_n = e_n\}$$

as the sequence

$$\begin{array}{l} l := \langle \mathtt{H} \rangle; \\ \langle \mathtt{H} \rangle.f_1 := e_1; \\ \quad \vdots \\ \langle \mathtt{H} \rangle.f_n := e_n; \\ \mathtt{H} := \mathtt{H} + 1 \end{array}$$

Our resulting program is now in the simplified language

$$\begin{array}{rcll} S & ::= & \{s_1, \ldots, s_n\} & \text{(ordered) statement list } (n \geq 0) \\ \\ s & ::= & l := e & \text{assignment statement} \\ & | & \textbf{if } e \textbf{ then } S \textbf{ else } S' & \text{conditional statement} \\ & | & \textbf{while } e \textbf{ do } S & \text{while loop} \end{array}$$

where $l \in \mathsf{Lexp}$, $e \in \mathsf{Exp}$, and Primop includes the source primitive operations as well as `deref`, `ptr`, and f for all $f \in \mathsf{Field}$.

We now wish to design a transition system to describe the semantics of MINI-C programs expressed in this simplified language. Recall that a transition system is a tuple

$$\langle \mathsf{CtrlPoint}, \mathsf{Var}, \mathsf{Val}, \longmapsto \rangle.$$

We already have the set Var of variables; it is one of the syntactic domains of MINI-C. All we have mentioned about the other three components is that Val includes the collection of pointers $\langle n \rangle$. We now describe each of these remaining three components in turn.

5.4 Control Points

In order to define a transition system describing the semantics of Mini-C programs, we must design a way to refer to the control points in a program. The control points are merely the statements in the program, and execution proceeds from control point to control point as it processes the statements in order.

A control point of a program is an "index" into the syntax tree of the program. Formally, a control point is a finite sequence of natural numbers.

$$C \quad \in \quad \mathsf{CtrlPoint} = \mathsf{Nat}^*$$

The empty sequence is written ϵ. If $C \in \mathsf{CtrlPoint}$, then $C.i \in \mathsf{CtrlPoint}$ and $i.C \in \mathsf{CtrlPoint}$ respectively represent the extensions of the sequence C on the right and on the left by $i \in \mathsf{Nat}$. Intuitively, the numbers in a control point describe, from left to right, how to descend into the syntax tree of a program. Formally, this is given below, where $S[C]$ returns the statement within statement list S at control point C.

$$\begin{array}{rcll}
\{s_1, \ldots, s_n\}[i] & = & s_i & \\
\{s_1, \ldots, s_n\}[i.j.C] & = & S_j[C] & \text{if } s_i = \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \\
\{s_1, \ldots, s_n\}[i.C] & = & S[C] & \text{if } s_i = \textbf{while } e \textbf{ do } S
\end{array}$$

Example 21 *The following* Mini-C *program is annotated with its control points.*

```
               {
1                if n < 0 then
1.1.1              n := -(n)
                 else
1.2.1              n := n * 2;
2                r := 1;
3                while n > 1 do
                   {
3.1                  r := r * n;
3.2                  n := n - 1
                   };
4                if r > 60 then
4.1.1              r := r - 1;
5                x := r
               }
```

Recall that a one-armed conditional **if** e **then** S *is an abbreviation for* **if** e **then** S **else** $\{\}$.

Note that a traversal into a conditional statement extends the control point by two numbers—an index identifying one of the two branches and an index into the statement list in that branch—while a traversal into a while loop extends the control point by only a single number—an index into the loop body.

5.5 Values

As described above, the set Var of variables is already given as one of the syntactic domains of Mini-C. So the next stage of the design of the transition system is the set Val of values. Recall that the states in the system are pairs

$$\mathsf{CtrlPoint} \times \mathsf{Store}$$

where stores are defined as in Chapter 2:

$$\begin{array}{lcll} \mathsf{Store} & = & \mathsf{Lval} \rightarrow \mathsf{Val} & \\ \mathsf{Lval} & = & \mathsf{Var} \cup (\mathsf{Val} \times \mathsf{Val}) & \text{l-values} \end{array}$$

We thus need to design an appropriate set of values for this store.

5.5.1 Constants, field names, and pointers

Recall the set $\mathsf{Constant}$ of Mini-C constants:

$$\begin{array}{lcll} c & ::= & n & \text{integers } (\mathsf{Int}) \\ & | & \mathtt{true} \mid \mathtt{false} & \text{booleans} \end{array}$$

All constants are values.

In Section 5.3 we performed a syntactic transformation where field names were considered as nullary primitive operations. Therefore, we include the set Field in the set of values.

In the same section, we suggested the approach of using pointer values to model the roots of mutable records. As we described, there is a pointer $\langle m \rangle$ for every natural number m. The set of all pointers is denoted $\mathsf{Pointer}$.

5.5.2 Immutable ordered tuples

Recall that Mini-C includes the n-ary primitive operation `tuple` for tuple construction and the operations π_i for tuple-component selection. Therefore, we would like the set of values to include all ordered tuples of values. The ordered tuple of the n values $v_1, \ldots, v_n$ is written $(v_1, \ldots, v_n)$.

5.5.3 The undefined value `undef`

As we described in Chapter 2, we demand that the set Val of values include the distinguished token `undef` representing the "undefined value". As we explained, this requirement comes from the fact that stores are total functions from l-values to values, and thus require such an explicit representation. For instance, at any given point in a Mini-C program, it is reasonable for only a small set of variables to be defined, and the store at that point would map all other variables to `undef`.

5.5.4 The set of values

Above, we described the different kinds of values in Mini-C. The set Val is their disjoint union (in order to distinguish pointers from natural-number constants and to distinguish a value from its unary tuple). It is defined inductively as the smallest set satisfying the following equation.

$$v \in \mathsf{Val} = \mathsf{Constant} + \mathsf{Field} + \mathsf{Pointer} + \mathsf{Val}^* + \{\mathtt{undef}\}$$

5.6 Semantics of Primitive Operations

The semantics of primitive operations follows the methodology described in Chapter 2. To review, the phrase $p(v_1, \ldots, v_n) \hookrightarrow_\sigma v$ means that the n-ary primitive operation p applied to values $(v_1, \ldots, v_n)$ in store σ evaluates to value v. However, the only primitive operation whose evaluation depends on σ (in other words, the only context-dependent operation, as defined in Definition 2), is `deref`, so we omit the σ parameter for all other operations.

Recall from Condition 1 that for any n-ary primitive operation $p \in \mathsf{Primop}$, for any n values $v_1, \ldots, v_n \in \mathsf{Val}$, and for any store $\sigma \in \mathsf{Store}$, there is at least one value $v \in \mathsf{Val}$ such that $p(v_1, \ldots, v_n) \hookrightarrow_\sigma v$. In other words, all primitive operations must be defined everywhere. All primitive operations in Mini-C are also *deterministic*; as defined in Definition 1, this means that they evaluate to only one value. Therefore, all Mini-C primitive operations are total functions.

We define the primitive operations in Mini-C below. We already gave most of these definitions in Chapter 2. A primitive operation evaluates to `undef` unless otherwise defined below. First, we have the constants and field names.

$$\begin{array}{rcl} c() & \hookrightarrow & c \\ f() & \hookrightarrow & f \end{array}$$

Now the integer operations

$$\begin{array}{rcl} \mathtt{+}(n, n') & \hookrightarrow & (n + n') \\ \mathtt{-}(n, n') & \hookrightarrow & (n - n') \\ \mathtt{-}(n) & \hookrightarrow & -n \\ \mathtt{*}(n, n') & \hookrightarrow & (n \times n') \end{array}$$

Next, the boolean operations and `if`.

$$\begin{array}{rcl}
\texttt{\&}(\texttt{true}, v) & \hookrightarrow & v \\
\texttt{\&}(v, \texttt{true}) & \hookrightarrow & v \\
\texttt{\&}(\texttt{false}, v) & \hookrightarrow & \texttt{false} \\
\texttt{\&}(v, \texttt{false}) & \hookrightarrow & \texttt{false} \\
\\
\texttt{<}(n, n') & \hookrightarrow & (n < n') \\
\texttt{>}(n, n') & \hookrightarrow & (n > n') \\
\texttt{=}(v, v') & \hookrightarrow & (v = v') \\
\texttt{<>}(v, v') & \hookrightarrow & (v \neq v') \\
\\
\texttt{if}(\texttt{true}, v, v') & \hookrightarrow & v \\
\texttt{if}(\texttt{false}, v, v') & \hookrightarrow & v'
\end{array}$$

Next, the operations for immutable tuples.

$$\begin{array}{rcl}
\texttt{tuple}(v, \ldots, v') & \hookrightarrow & (v_1, \ldots, v_n) \\
\pi_i((v_1, \ldots, v_n)) & \hookrightarrow & v_i
\end{array}$$

Finally, we have the operations to support mutable records.

$$\begin{array}{rcll}
\texttt{ptr}(n) & \hookrightarrow & \langle n \rangle & \\
\texttt{deref}(v, v') & \hookrightarrow_\sigma & \sigma(v.v') & \qquad \text{(context-dependent)}
\end{array}$$

5.7 Semantics of Expressions and L-expressions

The semantics of expressions and l-expressions are precisely the same as in Chapter 2. We review that definition here. Formally, the interpretations of expressions and l-expressions are given by the following relations.

- The phrase $l \vdash_\sigma w$ means that the l-expression l evaluates in store σ to l-value w.
- The phrase $e \vdash_\sigma v$ means that the expression e evaluates in store σ to value v.

We recall the following rules, which inductively define these relations.

$$x \vdash_\sigma x \qquad \frac{e \vdash_\sigma v \quad e' \vdash_\sigma v'}{(e.e') \vdash_\sigma (v.v')}$$

$$x \vdash_\sigma (\sigma\, x) \qquad \frac{e_i \vdash_\sigma v_i \quad p(v_1, \ldots, v_n) \hookrightarrow_\sigma v}{p(e_1, \ldots, e_n) \vdash_\sigma v}$$

We recall Lemma 1, that all expressions (respectively, l-expressions) evaluate to at least one value (respectively, l-value). In addition, we know from Lemma 2 that because Primop is deterministic that this value (respectively, l-value) is unique.

5.8 Transition-system Semantics

In this section we present a transition system that models the executions of Mini-C programs. We consider two ways to define such a system.

- The usual approach is to give a meta-rule for each kind of syntactic form in the language. There are one or more meta-rules for each syntactic form in the language (for instance, conditional, assignment, and so forth). These meta-rules describe how any occurrence of that syntactic form in the source program induces single-step transitions. The single-step transitions for the entire program is the collection (union) of all of these transitions.
- The second approach is technically equivalent, but uses the framework that we developed in Chapters 2, 3, and 4, thus opening up all the possibilities of our program-analysis methodology for the language. The idea is that each rule introduced by a meta-rule in the above approach is equivalent to a single transfer relation that describes all of the possible transitions induced by that rule. Hence, the approach is to give a rule for each kind of syntactic form in the language that describes how any occurrence of that form induces a transfer relation describing all of the possible single-step transitions for that occurrence.

We will illustrate each of these approaches in turn for Mini-C. But first, we need a helper function to manage control points.

5.8.1 The next function

In most languages, much of the control flow is syntactically apparent. Conceptually, the *dynamic semantics* of a language should not have to be concerned with syntactically apparent information. Of course, the program's flow of control must be part of the program's semantics, or else the semantics would not adequately model the program's execution. But for expository purposes, it is pleasing to factor out information that is a trivial property of the syntax, so that the rules of the semantics themselves succinctly capture exactly the dynamic properties of execution.

To this end, we will need a helper function next to manage the syntactically apparent control flow in a program. Given the control point C of a statement in program S, if C is in the middle of a statement list then next merely returns the control point of the next statement in the list. Note that if C points to a conditional or a while loop then the next statement is not necessarily the next control point in the execution.

$$\mathsf{next}_S(C.i) \quad = \quad C.(i+1) \quad \text{if } S[C.(i+1)] \text{ defined}$$

If C points to a statement that is the last in a statement list, then $\mathsf{next}_S(C)$ will not be defined by the above equation. There are two cases for when this might happen. The first case is that the statement to which C points is the last statement in the outermost statement list in S

(i.e., the statement list S itself). In this case next returns the empty control point ϵ to signal program completion.

$$\mathsf{next}_S(i) \quad = \quad \epsilon$$

The second case is that the statement to which C points is not in the outermost statement list. In that case, we want $\mathsf{next}_S(C)$ to return the control point of the next statement to be executed, which in this language is simply in a lexically-enclosed statement list and is always syntactically apparent. There are two cases. If C points to the last statement in an arm of a conditional statement s, then $\mathsf{next}_S(C)$ should be the next statement after s.

$$\mathsf{next}_S(C.j.i) \quad = \quad \mathsf{next}_S(C) \quad \text{if } S[C] = \textbf{if} \ldots \textbf{then} \ldots \textbf{else} \ldots$$

The second case is that C points to the last statement in a while loop s, in which case $\mathsf{next}_S(C)$ should be s itself, as the loop might need to be executed again.

$$\mathsf{next}_S(C.i) \quad = \quad C \quad \text{if } S[C] = \textbf{while} \ldots \textbf{do} \ldots$$

The following example demonstrates all of the concepts of the next function.

Example 22 *Consider the* MINI-C *program S in Example 21, shown below on the right. The complete definition of* next *is:*

$$\begin{aligned}
\mathsf{next}_S(0) &= 1\\
\mathsf{next}_S(1) &= 2\\
\mathsf{next}_S(1.1.0) &= 1.1.1\\
\mathsf{next}_S(1.1.1) &= 2\\
\mathsf{next}_S(1.2.0) &= 1.2.1\\
\mathsf{next}_S(1.2.1) &= 2\\
\mathsf{next}_S(2) &= 3\\
\mathsf{next}_S(3) &= 4\\
\mathsf{next}_S(3.0) &= 3.1\\
\mathsf{next}_S(3.1) &= 3.2\\
\mathsf{next}_S(3.2) &= 3\\
\mathsf{next}_S(4) &= 5\\
\mathsf{next}_S(4.1.0) &= 4.1.1\\
\mathsf{next}_S(4.1.1) &= 5\\
\mathsf{next}_S(4.2.0) &= 5\\
\mathsf{next}_S(5) &= \epsilon
\end{aligned}$$

```
            {
1             if n < 0 then
1.1.1           n := -(n)
              else
1.2.1           n := n * 2;
2             r := 1;
3             while n > 1 do
                {
3.1               r := r * n;
3.2               n := n - 1
                };
4             if r > 60 then
4.1.1           r := r - 1;
5             x := r
            }
```

Note from this example that for every statement list, there is an element in the domain of next ending with 0 and thus not a real control point in the program. The only reason for this is because we allow empty statement lists in MINI-C programs. So, for instance, if control point C points to a conditional expression, then $\mathsf{next}_S(C.2.0)$ always returns the next statement in an execution that takes the **else** arm. So, $\mathsf{next}_S(1.2.0) = 1.2.1$, which is the first (and only) statement in the **else** arm of the first conditional in the example program, but $\mathsf{next}_S(4.2.0) = 5$, because the **else** arm of the second conditional is empty, and thus execution immediately proceeds to the statement after the conditional.

5.8.2 Transition system via meta-rules

Now we can define the meta-rules that define the transition relation $\longmapsto$ for a Mini-C program S. There are five kinds of transitions. Each statement in S of the form

$$\textbf{if } e \textbf{ then } S \textbf{ else } S'$$

induces a two families of transitions: the transitions from a successful test of e into S, and the transitions from a failed test of e into the S'.

$$\frac{S[C] = \textbf{if } e \textbf{ then} \ldots \textbf{else} \ldots \quad C' = \mathsf{next}_S(C.1.0) \quad e \vdash_\sigma \texttt{true}}{(C, \sigma) \ \longmapsto \ (C', \sigma)}$$

$$\frac{S[C] = \textbf{if } e \textbf{ then} \ldots \textbf{else} \ldots \quad C' = \mathsf{next}_S(C.2.0) \quad e \vdash_\sigma \texttt{false}}{(C, \sigma) \ \longmapsto \ (C', \sigma)}$$

Each statement in S of the form

$$\textbf{while } e \textbf{ do } S'$$

induces two families of transitions: the transitions from a successful test of e into S', and the transitions from a failed test of e to the rest of the code after the loop.

$$\frac{S[C] = \textbf{while } e \textbf{ do} \ldots \quad C' = \mathsf{next}_S(C.0) \quad e \vdash_\sigma \texttt{true}}{(C, \sigma) \ \longmapsto \ (C', \sigma)}$$

$$\frac{S[C] = \textbf{while } e \textbf{ do} \ldots \quad C' = \mathsf{next}_S(C) \quad e \vdash_\sigma \texttt{false}}{(C, \sigma) \ \longmapsto \ (C', \sigma)}$$

Finally, each statement in S of the form

$$l := e$$

induces a family of transitions that perform the assignment in the store.

$$\frac{S[C] = (l := e) \quad C' = \mathsf{next}_S(C) \quad l \vdash_\sigma w \quad e \vdash_\sigma v}{(C, \sigma) \ \longmapsto \ (C', \sigma[w \mapsto v])}$$

Below is an example that illustrates how this transition system models program execution.

Example 23 *Recall the* Mini-C *program in Example 21. The transition system defines the following execution from a state at the beginning of the program and with an initial store in which* `n` *is bound to* -4 *and all other l-values are bound to* `undef`*. (The only store mappings* *shown are the mappings to non-*`undef` *values.)*

$$\begin{array}{ll}
 & (1\quad,\{\mathtt{n}\mapsto -4\}\qquad\qquad)\\
\longmapsto & (1.1.1,\{\mathtt{n}\mapsto -4\}\qquad\qquad)\\
\longmapsto & (2\quad,\{\mathtt{n}\mapsto 4\}\qquad\qquad)\\
\longmapsto & (3\quad,\{\mathtt{n}\mapsto 4,\mathtt{r}\mapsto 1\}\qquad)\\
\longmapsto & (3.1\quad,\{\mathtt{n}\mapsto 4,\mathtt{r}\mapsto 1\}\qquad)\\
\longmapsto & (3.2\quad,\{\mathtt{n}\mapsto 4,\mathtt{r}\mapsto 4\}\qquad)\\
\longmapsto & (3\quad,\{\mathtt{n}\mapsto 3,\mathtt{r}\mapsto 4\}\qquad)\\
\longmapsto & (3.1\quad,\{\mathtt{n}\mapsto 3,\mathtt{r}\mapsto 4\}\qquad)\\
\longmapsto & (3.2\quad,\{\mathtt{n}\mapsto 3,\mathtt{r}\mapsto 12\}\qquad)\\
\longmapsto & (3\quad,\{\mathtt{n}\mapsto 2,\mathtt{r}\mapsto 12\}\qquad)\\
\longmapsto & (3.1\quad,\{\mathtt{n}\mapsto 2,\mathtt{r}\mapsto 12\}\qquad)\\
\longmapsto & (3.2\quad,\{\mathtt{n}\mapsto 2,\mathtt{r}\mapsto 24\}\qquad)\\
\longmapsto & (3\quad,\{\mathtt{n}\mapsto 1,\mathtt{r}\mapsto 24\}\qquad)\\
\longmapsto & (4\quad,\{\mathtt{n}\mapsto 1,\mathtt{r}\mapsto 24\}\qquad)\\
\longmapsto & (5\quad,\{\mathtt{n}\mapsto 1,\mathtt{r}\mapsto 24\}\qquad)\\
\longmapsto & (\epsilon\quad,\{\mathtt{n}\mapsto 1,\mathtt{r}\mapsto 24,\mathtt{x}\mapsto 24\})
\end{array}$$

```
            {
1             if n < 0 then
1.1.1           n := -(n)
              else
1.2.1           n := n * 2;
2             r := 1;
3             while n > 1 do
                {
3.1               r := r * n;
3.2               n := n - 1
                };
4             if r > 60 then
4.1.1           r := r - 1;
5             x := r
            }
```

5.8.3 Transition system via transfer relations

The key idea of using transfer relations to replace meta-rules is that a single transfer relation can capture the commonalities inherent in each family of transitions defined by the meta-rules. For instance, in the meta-rule approach, every **if** e **then** S **else** S' statement induces an infinite family of transitions, one for each store σ in which e evaluates to `true`, from that statement into S. Each transition in this infinite family does exactly the same thing: simply test that e is true in the store on the left-hand side of the transition before proceeding to S.

This inspires the idea of defining a transfer relation $\Delta_{C,C'}$ for every pair $C, C' \in \mathsf{CtrlPoint}$ of control points in a Mini-C program S. Each one will specify exactly the transitions, as defined by the meta-rules above, from C to C. In other words,

$$(C,\sigma) \longmapsto (C',\sigma') \quad \text{iff} \quad \sigma\,\Delta_{C,C'}\,\sigma'.$$

Of course, the vast majority of these transfer relations will be the empty relation $\emptyset$, because single-step transitions between most pairs of control points are impossible.

The semantics of a Mini-C program S is thus defined as a finite table

$$\vec{\Delta} = \{\Delta_{C,C'} \mid C, C' \in \mathsf{CtrlPoint}\}$$

of transfer relations, one for each pair of control points in S, such that $\Delta_{C,C'}$ describes all of the transitions between control point C and the control point C'. This table is defined as follows.

$$\Delta_{C,C'} = \begin{cases} \boxed{e?\ \bullet} & \text{if} \quad S[C] = (\textbf{if } e \textbf{ then} \ldots \textbf{ else} \ldots) \quad \text{and} \quad C' = \mathsf{next}_S(C.1.0) \\ \boxed{e¿\ \bullet} & \text{if} \quad S[C] = (\textbf{if } e \textbf{ then} \ldots \textbf{ else} \ldots) \quad \text{and} \quad C' = \mathsf{next}_S(C.2.0) \\ \boxed{e?\ \bullet} & \text{if} \quad S[C] = (\textbf{while } e \textbf{ do} \ldots) \quad \text{and} \quad C' = \mathsf{next}_S(C.0) \\ \boxed{e¿\ \bullet} & \text{if} \quad S[C] = (\textbf{while } e \textbf{ do} \ldots) \quad \text{and} \quad C' = \mathsf{next}_S(C) \\ \boxed{l \mapsto e} & \text{if} \quad S[C] = (l := e) \quad \text{and} \quad C' = \mathsf{next}_S(C) \\ \emptyset & \text{otherwise} \end{cases}$$

We recall that $\boxed{e?\ \Delta}$ is short for $\boxed{e?\ \Delta \,|\, \emptyset}$ and $\boxed{e¿\ \Delta}$ is short for $\boxed{e?\ \emptyset \,|\, \Delta}$, where $\emptyset$ is the empty relation and $\bullet$ is the identity relation (i.e., empty parallel assignment).

Example 24 *The semantics defines 13 non-empty transfer relations for the* Mini-C *program in Example 21.*

$$\begin{array}{lcl@{\qquad\qquad}lcl} \Delta_{1,(1.1.1)} & = & \boxed{\mathtt{n} < 0?\ \bullet} & \Delta_{(3.2),3} & = & \boxed{\mathtt{n} \mapsto \mathtt{n} - 1} \\ \Delta_{(1.1.1),2} & = & \boxed{\mathtt{n} \mapsto \mathtt{-(n)}} & \Delta_{3,4} & = & \boxed{\mathtt{n} > 1¿\ \bullet} \\ \Delta_{1,(1.2.1)} & = & \boxed{\mathtt{n} < 0¿\ \bullet} & \Delta_{4,(4.1.1)} & = & \boxed{\mathtt{r} > 60?\ \bullet} \\ \Delta_{(1.2.1),2} & = & \boxed{\mathtt{n} \mapsto \mathtt{n} * 2} & \Delta_{(4.1.1),5} & = & \boxed{\mathtt{r} \mapsto \mathtt{r} - 1} \\ \Delta_{2,3} & = & \boxed{\mathtt{r} \mapsto 1} & \Delta_{4,5} & = & \boxed{\mathtt{r} > 60¿\ \bullet} \\ \Delta_{3,(3.1)} & = & \boxed{\mathtt{n} > 1?\ \bullet} & \Delta_{5,\epsilon} & = & \boxed{\mathtt{x} \mapsto \mathtt{r}} \\ \Delta_{(3.1),(3.2)} & = & \boxed{\mathtt{r} \mapsto \mathtt{r} * \mathtt{n}} & & & \end{array}$$

Note that this semantics does not need any of the mechanism developed in Chapter 3 for composing and manipulating transfer relations. Indeed, the transfer relations that it yields as the model of program execution are quite simple. But the intent is that the output of the semantics is merely a first step in an application of our program-analysis methodology. Once the single-step transfer relations of a Mini-C program are in hand, one can compose these transfer relations to yield a single compound relation that expresses the behavior of any finite segment of execution. The following example illustrates that composing single-step transfer relations is analogous to stringing together transitions defined by the meta-rules in the previous section.

Example 25 *The execution shown in Example 23 has control path*

$$1, (1.1.1), 2, 3, (3.1), (3.2), 3, (3.1), (3.2), 3, (3.1), (3.2), 3, 4, 5, \epsilon$$

and so its compound transfer relation is

$$\Delta_{1,(1.1.1)}; \Delta_{(1.1.1),2}; \Delta_{2,3}; \cdots ; \Delta_{3,4}; \Delta_{4,5}; \Delta_{5,\epsilon}$$

which relates the input store

$$\{\mathtt{n} \mapsto -4\}$$

to the output store

$$\{\mathtt{n} \mapsto 1, \mathtt{r} \mapsto 24, \mathtt{x} \mapsto 24\}.$$

Now, one can use the composition algorithm $\oplus$ of Chapter 3 to perform effectively these compositions, thereby facilitating the analysis of the program. Recall the convention of Chapter 4 of writing Δ_Γ to represent the transfer relation of control path Γ, which is a list of control points. Recall that

$$\Delta_{\Gamma,C,\Gamma'} = \Delta_{\Gamma,C} \oplus \Delta_{C,\Gamma'}.$$

Example 26 *A transfer relation expressing the above execution is computed as*

$$\Delta_{1,(1.1.1)} \oplus \Delta_{(1.1.1),2} \oplus \Delta_{2,3} \oplus \cdots \oplus \Delta_{3,4} \oplus \Delta_{4,5} \oplus \Delta_{5,\epsilon}$$

which, if the symbolic evaluation P *for primitive operations and* C *for conditional relations are both simply the identity function, is*

$$\boxed{(\mathtt{n} < 0)? \ \boxed{e_1? \ \boxed{e_2? \ \boxed{e_3? \ \boxed{e_4\text{¿} \ \boxed{(e' > 60)? \ \boxed{\mathtt{n},\mathtt{r},\mathtt{x} \mapsto e, e', e'}}}}}}}$$

where

$$\begin{array}{rcl}
e_1 & = & \mathtt{-(n)} > 1 \\
e_2 & = & (\mathtt{-(n)} - 1) > 1 \\
e_3 & = & ((\mathtt{-(n)} - 1) - 1) > 1 \\
e_4 & = & e > 1 \\
e & = & ((\mathtt{-(n)} - 1) - 1) - 1 \\
e' & = & ((1 * \mathtt{-(n)}) * (\mathtt{-(n)} - 1)) * ((\mathtt{-(n)} - 1) - 1)
\end{array}$$

Adding some logic to P *to simplify arithmetic operations could (in principle) yield a result as simple as*

$$\begin{array}{rcl}
e_1 & = & \mathtt{n} < -1 \\
e_2 & = & \mathtt{n} < -2 \\
e_3 & = & \mathtt{n} < -3 \\
e_4 & = & \mathtt{n} < -4 \\
e & = & \mathtt{-(n)} - 3 \\
e' & = & (\mathtt{-(n)} * (\mathtt{-(n)} - 1)) * (\mathtt{-(n)} - 2)
\end{array}$$

Note how the conditional relation expresses the control-flow constraints on this particular control path. Note also that the conjunction of the first five conditions in the above transfer relation can *only evaluate to* `true` *when* `n` *is* -4*. A* C *algorithm could (in principle) determine this property automatically, dispense with* e_1 *through* e_4*, pass this value of* `n` *onto a simple constant-folding symbolic evaluation of* e *and* e'*, and achieve the extremely simple transfer relation*

$$\boxed{\mathtt{n} = -4?\ \boxed{\mathtt{n}, \mathtt{r}, \mathtt{x} \mapsto 1, 24, 24}}.$$

Such sophisticated symbolic reasoning about integer arithmetic and conjoined comparison tests may be difficult to achieve in general. For the most part, we leave this topic as an open issue and offer no general algorithms. However, even simple symbolic evaluations can go far whenever any initial bindings are known. The following example illustrates how this works.

Example 27 *Suppose that* C *is the identity function, performing no simplification of conditional expressions, and* P *merely performs constant folding. Then*

$$\boxed{\mathtt{n} \mapsto -4} \oplus \Delta_{1,(1.1.1)} \oplus \Delta_{(1.1.1),2} \oplus \Delta_{2,3} \oplus \cdots \oplus \Delta_{3,4} \oplus \Delta_{4,5} \oplus \Delta_{5,\epsilon}$$

is the transfer relation

$$\boxed{\mathtt{n}, \mathtt{r}, \mathtt{x} \mapsto 1, 24, 24}.$$

So far, we have introduced only a single example Mini-C program. This example uses only integer data, and in particular does not allocate or use records. However, much of the sophistication of our methodology lies in its treatment of heap-allocated mutable data structures. The following example demonstrates record allocation.

Example 28 *Consider the* Mini-C *program*

```
1    while a <> nil do
     {
       new := {car = a.car, cdr = b};
       a := a.cdr;
       b := new
     }
```

that constructs a reverse of list a*. Let* Γ *be the control path that starts at control point* 1*, progresses through one iteration of the loop, and ends back at control point* 1*. Then,*

$$\Delta_\Gamma \quad = \quad \boxed{(\mathsf{a} \mathrel{\mathtt{<>}} \mathtt{nil})?\ \boxed{\begin{array}{lcl} \mathsf{a}, \mathsf{b}, \mathtt{new}, \mathtt{H}, & \mapsto & \mathsf{a.cdr}, \langle\mathtt{H}\rangle, \langle\mathtt{H}\rangle, \mathtt{H} + 1, \\ \langle\mathtt{H}\rangle.\mathsf{car}, \langle\mathtt{H}\rangle.\mathsf{cdr} & & \mathsf{a.car}, \mathsf{b.cdr} \end{array}}}$$

represents the transfer relation of one iteration and

$$\Delta_{\Gamma;\Gamma} \quad = \quad \boxed{(\mathsf{a} \mathrel{\mathtt{<>}} \mathtt{nil})?\ \boxed{(\mathsf{a.cdr} \mathrel{\mathtt{<>}} \mathtt{nil})?\ \Delta}}$$

where

$$\Delta \;=\; \boxed{\begin{array}{lll} \texttt{a},\texttt{b},\texttt{new},\texttt{H}, & & \texttt{a.cdr.cdr},\langle\texttt{H + 1}\rangle,\langle\texttt{H + 1}\rangle,\texttt{H + 2}, \\ \langle\texttt{H}\rangle\texttt{.car},\langle\texttt{H}\rangle\texttt{.cdr}, & \mapsto & \texttt{a.car},\texttt{b.cdr}, \\ \langle\texttt{H + 1}\rangle\texttt{.car},\langle\texttt{H + 1}\rangle\texttt{.cdr} & & \texttt{a.cdr.car},\langle\texttt{H}\rangle \end{array}}$$

represents the transfer relation of two adjacent iterations

Note how an assignment relation can represent multiple record allocations in parallel via the expressions $\langle\texttt{H}\rangle$, $\langle\texttt{H + 1}\rangle$, $\langle\texttt{H + 2}\rangle$, and so forth. These expressions evaluate to sequential free pointers. The assignment to `H` reflects the total number of pointers allocated by the execution segment that the transfer relation models.

Also, note again that the conditional relations encode the conditions under which a particular control path is taken.

The following example demonstrates the subtlety of unknown initial aliasing.

Example 29 *Consider the* Mini-C *program*

```
1    while (a <> nil) do
     {
        temp := a;
        a := a.cdr;
        temp.cdr := b;
        b := temp
     }
```

that destructively appends the reverse of list `a` *onto list* `b`*. If* Γ *is the control path that begins at control point* 1*, progresses through one iteration of the loop, and ends back at control point* 1*, then*

$$\Delta_\Gamma \;=\; \boxed{(\texttt{a <> nil})?\ \boxed{\texttt{a},\texttt{b},\texttt{temp},\texttt{a.cdr} \mapsto \texttt{a.cdr},\texttt{a},\texttt{a.cdr},\texttt{b}}}$$

is the transfer relation of one loop iteration, and

$$\Delta_{\Gamma;\Gamma} \;=\; \boxed{(\texttt{a <> nil})?\ \boxed{(\texttt{a.cdr <> nil})?\ \boxed{(\texttt{a = a.cdr})?\ \Delta \,\big|\, \Delta'}}}$$

where

$$\Delta \;=\; \boxed{\texttt{a},\texttt{b},\texttt{temp},\texttt{a.cdr} \mapsto \texttt{if}(\texttt{a = a.cdr},\texttt{b},\texttt{a.cdr.cdr}),\texttt{a.cdr},\texttt{a.cdr},\texttt{a}}$$

$$\Delta' \;=\; \boxed{\texttt{a},\texttt{b},\texttt{temp},\texttt{a.cdr},\texttt{a.cdr.cdr} \mapsto \texttt{if}(\texttt{a = a.cdr},\texttt{b},\texttt{a.cdr.cdr}),\texttt{a.cdr},\texttt{a.cdr},\texttt{b},\texttt{a}}$$

is the transfer relation of two adjacent loop iterations. If `C` *were defined to propagate the first test of* `a = a.cdr` *into its two branches, then the composition algorithm could simplify the* `if` *expressions and achieve*

$$\Delta \;=\; \boxed{\texttt{a},\texttt{b},\texttt{temp},\texttt{a.cdr} \mapsto \texttt{b},\texttt{a.cdr},\texttt{a.cdr},\texttt{a}}$$

$$\Delta' \;=\; \boxed{\texttt{a},\texttt{b},\texttt{temp},\texttt{a.cdr},\texttt{a.cdr.cdr} \mapsto \texttt{a.cdr.cdr},\texttt{a.cdr},\texttt{a.cdr},\texttt{b},\texttt{a}}$$

This example is worth some study. It expresses that the net effect of executing two adjacent iterations of the loop in some context (store) depends on whether a was aliased to a.cdr in that context (in other words, if a is a circular list of length one). If not, then the execution has the net effect of Δ', which directly expresses the expected net behavior of two iterations of a reverse-append routine. On the other hand, if a is initially aliased to a.cdr, then some examination of Δ reveals that the net effect of two iterations reduces to swapping a and b.

This example thus demonstrates that the effect of aliasing on data-structure dereference and destructive assignment is rather subtle and unpredictable, but the composition operation $\oplus$ on transfer relations reveals this subtlety. Furthermore, it suggests that it is well worth the effort to design the C algorithm to look for and simplify syntactically redundant conditional expressions. In this dissertation, we do not describe such an advanced C algorithm, and so this is left for future work.

5.9 Modeling & and Pointer Arithmetic

The only reason that we did not include arrays in Mini-C was for simplicity. In Chapter 7 we will show how to model arrays in a functional language with our methodology, and it is straightforward to extend Mini-C in the same manner. In this section, we give a discussion of how to add some of the features of C's pointers that are not present in Mini-C. For generality, this section will assume that Mini-C includes arrays as described in Chapter 7.

C includes the following expressions.

&x	the address of variable x
&(s.f)	the address of field f of struct s
&(a[i])	the address of element i of array a

Unlike C, Mini-C has no & operator. Related to this is our choice to treat pointers as records with the single field *.

Alternatively, we could have treated a pointer as a pair value (v, v') representing the l-value $v.v'$. In this way, we can treat the latter two of the three cases above via the syntactic translation

$$\texttt{\&}(e.e') \quad \Rightarrow \quad (e, e')$$

where $e[e']$ is represented as $e.e'$, as we describe in Chapter 7. Then, we would treat an occurrence of $\texttt{*}e$ not as a reference (as an l-expression) or dereference (as an expression) of the field named * of the record e, but rather as an extraction of the l-value represented by the pair e. This would be accomplished by the following syntactic transformation.

$$\texttt{*}e \quad \Rightarrow \quad (\pi_1(e)).(\pi_2(e))$$

Recall that an occurrence of $e.e'$ as an expression (as opposed to an l-expression) is short for $\texttt{deref}(e, e')$.

In this manner, we can treat most of the functionality of C's & and * operators. What we cannot do is to take the address of a variable, and there is a good reason why this is the case. Intuitively, & coerces an l-value into a value (so that it may be manipulated as data and so forth), and * coerces a value back into an l-value. Above, we coerce a reference l-value $v.v'$ as the pair value (v, v'), which may then be coerced back into the l-value $v.v'$ (and dereferenced, if treated as an expression). We could attempt a similar approach with variables—for instance, coercing the variable x into the token 'x'. In our current formulation, the only l-expression that evaluates to x is x itself, and so there is no way to translate an arbitrary expression e into an l-expression that will evaluate to x if e evaluates to 'x'. But this is just due to our particular language of expressions and l-expressions and our choice to model Mini-C variables with l-expression variables. In principle, there is no difficulty to extend the notion of & for variables.

With the above model of pointers, it is possible to model C's pointer arithmetic for array indices. A pointer to an array element is (v, v'), where v is the array and v' is the index of the element (as we explain in Chapter 7). Suppose the expression

$$e \uparrow e'$$

represents the increment of pointer e by e' (which would be written as e + e' in C). We would translate this into the Mini-C expression

$$(\pi_1(e), \pi_2(e) + e').$$

Chapter 6

First-Class Functions: The Language Pure

In Chapter 5 we presented the imperative while-loop language Mini-C, the primary purpose of which was to introduce the methodology of defining a transition-system semantics of a programming language with computer-representable transfer relations representing the single steps, and then using the $\oplus$ algorithm to build multiple steps corresponding to particular control paths in the program. But Mini-C is a rather simple language, and so in this chapter we consider more advanced language features. Our purpose is to demonstrate that our methodology of semantics-based program analysis is reasonably general.

The only control constructs in Mini-C are conditionals and while loops. One can get by without any other control constructs, but it would be quite inconvenient for most programming tasks. Real programming languages have some mechanism for defining functions. A function accepts some input data (parameters) from its caller and returns a result value to the caller. In some languages, such as Haskell [H$^+$92], functions have the same input-output behavior in any context. This is sometimes known as *referential transparency* [SS90]. We call this kind of function "pure". The vast majority of programming languages, however, provide impure functions. In this chapter we model a programming language with pure functions, and in the next chapter we will extend this language with imperative features and impure functions.

In some languages, the functions are said to be *first class*. This means that the functions are semantic values, and as such can be manipulated by a program like any other value. For instance, they may be assigned to variables, placed in data structures, and passed to other functions. The functions in most advanced languages, such as Scheme [ReC86] or Standard ML (SML) [MTH90], are first class. In contrast, the functions in C [KR78], Fortran [Knu71], and Pascal [Bar81] are not first class.

In some ways, our methodology must be rather stretched to handle first-class functions. We will see this below and in Chapter 7, and we will give a summary at the end of Chapter 7.

6.1 Substitution vs. Closures

Consider the syntax of λ-calculus terms [Bar84]:

$$e ::= x \mid \lambda x.\, e \mid e\, e'$$

Now consider the following term:

$$(\lambda x.\, \lambda y.\, x)\, (\lambda z.\, z)$$

Via the reduction rules of the λ-calculus, this term reduces in a single step to the (unique up to renaming) normal form:

$$\lambda y.\, \lambda z.\, z$$

This term represents a function that, given any argument, yields the identity function.

Much of programming-language theory and practice is based on the notion that reduction of λ-calculus terms is a kind of *computation.* Even further, the Curry-Howard isomorphism [How80] introduces the connection between proof theory and computation, and consequently between logical systems and programming languages. (We refer the curious reader to [GLT89].) Let us consider how the above λ-calculus term might correspond to a SML program (chosen rather arbitrarily, simply as an example of a "real" language), and how its reduction might correspond to the execution of the program. The SML program

```
(fn x => (fn y => x)) (fn z => z)
```

corresponds to the λ-calculus term above; indeed, the syntax trees of the two terms are isomorphic. Now, consider the execution of this SML program. Everyone who has written SML programs imagines that the execution of this program will proceed something like this:

1. Evaluate `(fn x => (fn y => x))`.
   (a) Create a closure f in the heap for `(fn x => (fn y => x))`.
   (b) Return f as the result of evaluation.
2. Evaluate `(fn z => z)`.
   (a) Create a closure g in the heap for `(fn x => (fn y => x))`.
   (b) Return g as the result of evaluation.
3. Apply f to g.
   (a) Bind `x` to g.
   (b) Create a closure h in the heap for `(fn y => x)` with this binding of `x`.
   (c) Return g as the result of evaluation.
4. Return g as the result of evaluation.

There seems to be much more going on in the execution of the SML program than in the single step reduction from

$$\lambda x.\, \lambda y.\, x\, (\lambda z.\, z)$$

to

$$\lambda y.\, \lambda z.\, z.$$

The overarching reason is that, as we have suggested, reduction of λ-terms is an abstract notion of computation, while SML programs execute on real computers. The salient point that this example illustrates is that each step of a λ-term reduction builds a whole new term, but it is infeasible to build literally an entire SML program over and over during execution. More specifically, a reduction of a λ-term *substitutes* the argument of a function for every occurrence of the parameter of the function in the body of the function. In the above example, the reduction substitutes the literal term $\lambda z.\, z$ for the single occurrence of x in $\lambda y.\, x$ to build the final normal-form term. Theoretically, an implementation of a real programming language could be based on a similar idea. But in practice, it is usually more efficient to build *closures* instead of performing substitution.

The difference between closures and substitution lies in the treatment of variables. A programmer is accustomed to thinking of variables as identifiers that are bound to values when the program runs. This deeply ingrained notion that a variable "has a value" is partially an artifact of this implementation issue, and is supported by the standard denotational model of the λ-calculus which models a term $\lambda x.\, e$ as a continuous function [Sto77]. In the reduction of λ-terms, one must consider variables in a different light; they are placeholders that during reduction (execution) are replaced with terms and disappear entirely.

Because the overarching goal of program analysis is to determine information about the run-time behavior of programs, one must begin by modeling the programming language in a way that reflects or abstracts this run-time behavior. Therefore, because real implementations typically use closures to model first-class functions, we will model functions with closures in our semantics.

6.2 Syntax

We now present the purely functional language called Pure. A program is a member of the set Term of terms, written in a brand of continuation-passing style [LD93].

$$\begin{array}{rcll}
t & ::= & \mathbf{let}\ x = e\ \mathbf{in}\ t & \text{local binding} \\
 & | & \mathbf{rec}\ g\ \mathbf{in}\ t & \text{recursive function binding} \\
 & | & e(\vec{e}) & \text{function application} \\
 & | & \mathbf{if}\ e\ \mathbf{then}\ t\ \mathbf{else}\ t' & \text{conditional} \\
 & | & e & \text{simple term } (\mathsf{Exp}) \\
 & & & \\
g & ::= & x(\vec{y}) = t' & n\text{-ary function definitions}
\end{array}$$

A simple term is an expression as defined in Chapter 2.

$$\begin{array}{rcll} e & ::= & x & \text{variable lookup} \\ & \mid & p(e_1, \ldots, e_n) & \text{primitive application} \\ \\ x, y, z & \in & \mathsf{Var} & \text{variables} \end{array}$$

Usually, we use x for a function name or let binding, y for a function parameter, and z to refer to a free variable of a term t, the set of which is denoted by $\mathbf{FV}(t)$ and defined in the usual fashion. Expressions that appear in Pure terms may use the following primitive operations, which are members of Primop.

$$\begin{array}{rcll} p & ::= & c & \text{constants (nullary)} \\ & \mid & \texttt{+} \mid \texttt{-} \mid \texttt{*} & \text{integer operations (binary)} \\ & \mid & \texttt{<} \mid \texttt{>} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{\&} & \text{boolean operations (binary)} \\ & \mid & \texttt{tuple} & \text{ordered-tuple construction (n-ary)} \\ & \mid & \pi_i & \text{ordered-tuple component selection (unary)} \end{array}$$

Constants are the integers and booleans, as in Mini-C.

6.3 Discussion

Notice that

$$e(e')(e'')$$

is not a valid program in Pure. For instance, consider a program that defines and then calls a curried addition function.

$$\begin{array}{l} \textbf{rec}\ \texttt{f}(\texttt{x}) = (\textbf{rec}\ \texttt{g}(\texttt{y}) = \texttt{x + y}\ \textbf{in}\ \texttt{g}) \\ \textbf{in}\ \texttt{f}(24)(42) \end{array}$$

This is not a term. One would have to write this by using a *continuation* function as an interface between the application $\texttt{f}(1)$ and the application $v(2)$ where v is the result of the former application.

$$\begin{array}{l} \textbf{rec}\ \texttt{f}(\texttt{x},\texttt{k}) = (\textbf{rec}\ \texttt{g}(\texttt{y}) = \texttt{x + y}\ \textbf{in}\ \texttt{k}(\texttt{g})) \\ \textbf{in}\ (\textbf{rec}\ \texttt{k}(\texttt{v}) = \texttt{v}(42)\ \textbf{in}\ \texttt{f}(24, \texttt{k})) \end{array}$$

We are moving toward a form of continuation-passing style (CPS). CPS was studied early as the subset of λ-calculus terms for which call-by-name and call-by-value reduction strategies are equivalent [Plo75]. The first major practical use of CPS was in the Scheme Rabbit compiler [Ste78], which translated source programs into a restricted syntax much like ours. This was later done in the Orbit Scheme compiler [Kra88] and then in the SML/NJ compiler [App92]. All translations are based on a universal calling convention in which all source functions take a continuation argument and all source applications must thus pass a continuation function describing the remainder of the computation. So, for instance, an automatic CPS converter might produce

$$\begin{array}{l}\textbf{rec}\ \texttt{f}(\texttt{x},\texttt{k}) = (\textbf{rec}\ \texttt{g}(\texttt{y},\texttt{k}) = \texttt{k}(\texttt{x + y})\ \textbf{in}\ \texttt{k}(\texttt{g}))\\ \textbf{in}\ (\textbf{rec}\ \texttt{k}(\texttt{v}) = \texttt{v}(42,\texttt{top})\ \textbf{in}\ \texttt{f}(24,\texttt{k}))\end{array}$$

for the program above.

These translations are well studied, both in theory and practice [Plo75] (see also other references cited above), and so we will not go any further into CPS here. Suffice it to say that our syntactic restriction does not limit the expressivity of the language.

6.4 Semantics

In Chapter 5 we modeled a Mini-C program by a transition system in which a state is a pair of a control point, representing the current syntactic position of execution, and a store, representing the state of the memory/data. We gave alternate definitions of the single-step transitions induced by a program, one in terms of meta-rules (the standard practice) and one in terms of transfer relations. It is the latter formulation that provides a basis for program analysis with our methodology. In this section, we discuss semantics of Pure in a similar fashion.

6.4.1 Control, data, and execution states

Actually, both the notion of control point and the notion of store are simpler in Pure than in Mini-C.

We designed a whole notation for the control points of a Mini-C program, but it turns out that one can simply use the subterms of a Pure program to function as control points.[1] This is intuitively pleasing because the control point itself has meaning: if an execution is at control point t, it means that the rest of the execution is the evaluation of t. (This works because of Pure's CPS-like syntax.) In contrast, a sequence of integers that functions as a control point of a Mini-C program has no meaning alone; it is only an index into the text of the program.

As for stores, because there are no assignable data structures in Pure, much of the complexity of stores is not needed to model the data. Recall that a store is a map from l-values, which are either variables or references $v.v'$, to values. In Pure there is no need for the references, and so all that is needed is a map from variables to values. This is the familiar notion of an *environment*:

$$\rho \ \in \ \mathsf{Env} \ = \ \mathsf{Var} \to \mathsf{Val} \qquad \text{environments}$$

Recall that there were five different kinds of values (members of Val) in Mini-C:

- constants (members of $\mathsf{Constant}$)

[1] Actually, in Section 6.4.3 we will need to refer to a function definition $x(\vec{y}) = t$ as a control point. In this case, that control point is identified with the control point t. We will discuss this later.

- field names (members of Field)
- pointers to assignable data structures (members of $\mathsf{Pointer}$)
- immutable tuples (members of Val^*)
- the undefined value `undef`

Again because there are no assignable data structures in Pure, pointers are not needed, and neither are field names. However, Pure has first-class functions, and so there must be values that model these functions. As we explained in Section 6.1, it is best for many applications of program analysis to model a function as a *closure*. A closure has two parts:

- a function g (a phrase of the form $x(\vec{y}) = t$)
- an environment, providing the values for all free variables of g

The set of closures is thus defined as follows:

$$\langle g, \rho \rangle \quad \in \quad \mathsf{Closure}$$

Finally, the set of values is given by the following equation:

$$v \quad \in \quad \mathsf{Val} \quad = \quad \mathsf{Constant} + \mathsf{Closure} + \mathsf{Val}^* + \{\texttt{undef}\}$$

Note there is a circularity in the equation for Val, and there is another circularity in the equations for Env, $\mathsf{Closure}$, and Val. The actual sets are defined by mutual induction, as the least solution to these three equations.

As an aside, an infamous difficulty in designing analyses of languages with either first-class functions or data structures lies in how to deal with these circularities in an analysis algorithm that is guaranteed to terminate. The circularity in the equation for Val arises from immutable tuples, and indeed most static analyses of even immutable structured values—not to mention mutable data structures—are quite crude (e.g., [Wad87], [Hei92]). One of the few satisfactory analyses of structured values is [Deu92], but it is still somewhat ad hoc and also quite complicated. The other circularity arises from first-class functions, and again it is no coincidence that analysis designers have traditionally encountered trouble with first-class functions. The usual ad hoc approaches are to be found in the work on denotational-based abstract interpretations, usually applied to strictness analysis [BHA86], the work on finite approximations of closures [Shi91], or the work on augmenting higher-order type systems with "effects" [TJ92]. None of this work seems satisfactory. What lies at the root of these problems is the ubiquitous analysis methodology that begins with the design of a (hopefully clever) approximation of an infinite domain. In contrast, because our methodology is centered around the analysis of the *changes* to a store (or environment in the case of Pure) rather than the store (or environment) itself, complexities (induction, recursion, infinite sets, etc.) in the structure of values themselves do not cause any *a priori* difficulty; rather, the focus is on the complexity of the *transitions between* states.

As we explained above, a state of execution comprises the state of control, which is a term, and the state of data, which is an environment.

$$\mathsf{State} = \mathsf{Term} \times \mathsf{Env}$$

What remains is to define the transitions induced by a PURE term. These model the single steps of execution of the term.

6.4.2 Transitions via meta-rules

Here we describe how each kind of term induces transitions. No transition is possible from an expression term; if an execution reaches the state

$$(e, \rho)$$

then the execution halts, and the result of the execution is the value v such that

$$e \vdash_\rho v$$

which is guaranteed to be unique because PURE is deterministic.

Each of the four other kinds of terms take transitions.

- **let** $x = e$ **in** t. In this case, x is bound to a value to which e evaluates in the current environment, and execution proceeds to t.

  $$\frac{e \vdash_\rho v}{(\mathbf{let}\ x = e\ \mathbf{in}\ t, \rho) \ \longmapsto\ (t, \rho[x \mapsto v])}$$

  Note that an environment is just a store in which all reference l-values $(v.v')$ are bound to `undef`, and so for convenience we use the same definition of $\vdash$ for the evaluation of an expression in an environment. Similarly, the definition of $\rho[x \mapsto v]$ is a special case of store extension, defined on page 2.4.2.

  $$(\rho[x \mapsto v])\, y = \begin{cases} v & \text{if } x = y \\ \sigma\, y & \text{otherwise} \end{cases}$$

  This notion of variable binding may seem strange. Why isn't it necessary to rename the variable x in order to avoid conflicts with other occurrences of x in ρ that might be needed later in the computation? It turns out that these other occurrences of x will always be captured in closures and so will not interfere with the update of *env*. We will discuss this further in Section 6.5.

- **rec** g **in** t where $g = (x(\ldots) = \ldots)$ In this case, x is bound to a closure whose function component is g and whose environment component is the current environment, and execution proceeds to t.

  $$\frac{g = (x(\ldots) = \ldots)}{(\mathbf{rec}\ g\ \mathbf{in}\ t, \rho) \ \longmapsto\ (t, \rho[x \mapsto \langle g, \rho\rangle])}$$

- $e(\vec{e})$. For a transition to be possible from this term, e must evaluate to some closure $\langle x(\vec{y}) = t, \rho' \rangle$ in the current environment. In that case, the new environment is ρ' extended with the following additional bindings: a binding from x to the closure itself and a binding from the variables $\vec{y}$ to the corresponding values to which the expressions $\vec{e}$ evaluate in the current environment. Then execution proceeds to t.

  $$\frac{e \vdash_\rho v \quad (\vec{e})_i \vdash_\rho (\vec{v})_i \quad v = \langle x(\vec{y}) = t, \rho' \rangle}{(e(\vec{e}), \rho) \longmapsto (t, \rho'[x \mapsto v][\vec{y} \mapsto \vec{v}])}$$

  Note that ρ is simply "thrown away" in this transition. This is because PURE terms are in continuation-passing style, and so an evaluation returns only when the execution of the entire program is complete. All parts of ρ that will be needed in the future computation must be passed through via the arguments $\vec{e}$, typically in the closure of a continuation function.

  It may at first seem unnecessary to have closures in the first place. If we appropriately renamed variables during execution, we could ensure that the bindings of the free variables of a function g are never overwritten later in the execution. In this way, there would be no need to save and restore ρ'; instead, ρ may simply be threaded through on function application. We discuss this choice further in Section 6.5.

- **if** e **then** t **else** t'. For a transition to be possible from this term, e must evaluate to either `true`, in which case evaluation proceeds to t, or `false`, in which case evaluation proceeds to t'.

  $$\frac{e \vdash_\rho \texttt{true}}{(\textbf{if } e \textbf{ then } t \textbf{ else } t', \rho) \longmapsto (t, \rho)}$$

  $$\frac{e \vdash_\rho \texttt{false}}{(\textbf{if } e \textbf{ then } t \textbf{ else } t', \rho) \longmapsto (t', \rho)}$$

6.4.3 Transitions via transfer relations

Instead of using meta-rules to define the transitions, it is possible to represent them directly in a computer as transfer relations. The methodology here is exactly the same as for PURE. The transfer relation

$$\Delta_{t,t'}$$

represents all and only the valid single-step transitions from term t to term t'; it is a binary relation between the environment at t and the environment at t'.

Functions as control points

In MINI-C, the definition of the single-step transfer relations precisely corresponded to the meta-rules that defined the transitions. However, there is a subtle issue concerning first-class functions. It turns out that to define the single-step transfer relations of a Pure term, we need to have a slightly special treatment for the control points of functions.

In particular, we need to use a phrase $x(\vec{y}) = t$ as a control point. In this case, that control point is identified with t, but in a rather subtle way. For instance, consider two functions with the same body t_2, but different names and parameters:

$$\begin{array}{rcl} g & = & x(\vec{y}) = t_2 \\ g' & = & x'(\vec{y}') = t_2 \end{array}$$

The transfer relation

$$\Delta_{t_1,g}$$

describes the single steps from term t_1 into the function g, and the transfer relation

$$\Delta_{t_1,g'}$$

describes the single steps from term t_1 into the function g'. Both of these may be composed with the transfer relation

$$\Delta_{t_2,t_3}$$

that describes both the first step of function g and the first step of function g'.

So, in other words, g and g' are both identified with t_2 for the purpose of relating the transfer-relation formulation of the semantics with the meta-rule formulation, and thus for composing a transfer relation that ends with one control point (in this case, g or g') with a transfer relation that begins with the same control point (in which case, t_2). However, one may give separate definitions for both $\Delta_{t_1,g}$ and $\Delta_{t_1,g'}$.

Primitive operations to support closures

It is necessary to add three new families of primitive operations to Primop in order to build and examine closures. All of these operations are simple.[2]

- There is an n-ary simple primitive operation $\texttt{closure}_{g,(z_1,\ldots,z_n)}$ for every function g and variables $z_1, \ldots, z_n$ that creates a closure whose function is g and whose environment binds the variables $z_1, \ldots, z_n$. It is defined as follows:

$$\texttt{closure}_{g,(z_1,\ldots,z_n)}(v_1, \ldots, v_n) \hookrightarrow \langle g, \rho \rangle$$

where:

$$\begin{array}{rcll} \rho\, z_i & = & v_i & \\ \rho\, z & = & \texttt{undef} & \text{if } \forall 1 \leq i \leq n.\ z \neq z_i \end{array}$$

[2] Note that in general it makes sense only to have simple primitive operations because Pure is a deterministic pure language.

- There is a unary simple primitive operation $\texttt{code}_g$ for every function g that tests whether the function of a closure is equal to g. It is defined as follows:

$$\begin{array}{ll}\texttt{code}_g(\langle g,\rho\rangle) \hookrightarrow \texttt{true} & \\ \texttt{code}_g(v) \hookrightarrow \texttt{false} & \text{otherwise}\end{array}$$

- There is a unary simple primitive operation $\texttt{env}_z$ for every variable z that returns the value of the variable z in the environment of a closure. It is defined as follows:

$$\begin{array}{ll}\texttt{env}_z(\langle g,\rho\rangle) \hookrightarrow (\rho\, z) & \\ \texttt{env}_z(v) \hookrightarrow \texttt{undef} & \text{if } v \notin \mathsf{Closure}\end{array}$$

Free variables and bisimulation

For technical reasons that we will explain below, it is necessary to introduce a kind of equivalence relation on states.

One can also easily show that the only variables whose values might be needed in the execution of term t are the free variables of t. The following bisimulation expresses this precisely.

Definition 11 (Similar values and states) *We define the similar relation $\sim$ on values and states as follows, where $\mathbf{FV}(t)$ denotes the free variables of t, and similarly for $\mathbf{FV}(g)$.*

- *Two values v and v' are said to be similar (written $v \sim v'$) if either $v = v'$ or $v = \langle g,\rho\rangle$, $v' = \langle g,\rho'\rangle$, and*
$$x \in \mathbf{FV}(g) \Rightarrow (\rho\, x) \sim (\rho'\, x).$$

- *Two states (t,ρ) and (t',ρ') are said to be similar (written $(t,\rho) \sim (t',\rho')$) if $t = t'$ and*
$$x \in \mathbf{FV}(t) \Rightarrow (\rho\, x) \sim (\rho'\, x)$$

Proposition 1 (Bisimulation) *Let $\longmapsto^*$ be the transitive closure of $\longmapsto$. If $\psi_1 \sim \psi_1'$ and $\psi_2 \sim \psi_2'$ then*
$$\psi_1 \;\longmapsto^*\; \psi_2 \iff \psi_1' \;\longmapsto^*\; \psi_2'.$$

The single-step transfer relations

Now we can define a single-step transfer relation $\Delta_{t,t'}$ for every two terms t and t'; this relation specifies how the environment changes in a transition from t to t'. We also define $\Delta_{t,g}$ for the transitions into a functions g, as described above. In Mini-C, these transfer relations are indexed by control points instead of terms; because there are only a finite number of control points in a Mini-C program, the number of single-step transfer relations for a Mini-C program is also finite. The situation is not quite analogous for Pure. A Pure term t does indeed have only a finite number of subterms. However, if t is meant to be executed in an environment that initially contains some (non-`undef`) values—which would typically be the case for partial programs, or in other words for terms t that have free variables—then some of those values might be closures that contain terms not in t.

In Mini-C the single-step transfer relations precisely mirrored the semantic meta-rules. This is almost the case here, but there is some difference due to functions. Above, we gave the meta-rules for each of the four kinds of Pure terms. Here, we do the same for the single-step transfer relations, all of which are $\emptyset$ unless defined otherwise below.

- **let** $x = e$ **in** t. This case is just like the meta-rule; there is a single-step transfer relation from this term to t that describes the new binding to the environment:

$$\Delta_{(\textbf{let}\ x=e\ \textbf{in}\ t),t} = \boxed{x \mapsto e}$$

- **rec** g **in** t. This case is very much like the meta-rule; there is a single-step transfer relation from this term to t that describes the binding of the new closure. But there is a subtle difference. The meta-rule builds a closure that contains the entire current environment, while the transfer relation builds an environment that keeps the bindings only of the free variables of the function g. The bisimulation proposition above justifies this change. This relation uses a primitive operation to create this closure:

$$\Delta_{(\textbf{rec}\ g\ \textbf{in}\ t),t} = \boxed{x \mapsto \texttt{closure}_{g,(z_1,\ldots,z_n)}(z_1,\ldots,z_n)}$$

  where $\{z_1,\ldots,z_n\} = \mathbf{FV}(g)$ (ordered arbitrarily).

  This difference between the meta-rule and the transfer relation is not conceptually deep. We could very well have defined the meta-rule to restrict the environment of the closure to the free variables of g, as well, but that choice is unnecessarily cumbersome, not to mention non-standard. On the other hand, there are two reasons why we define the transfer relation as we do.

  - We have less flexibility in the design of the transfer relation. Because environments are not members of Val (in which case we might have imagined a nullary context-sensitive primitive operation that evaluates in ρ to ρ itself), it is necessary to build the environment explicitly, as we do with $\texttt{closure}_{g,(z_1,\ldots,z_n)}$. Therefore, we must know the set of variables to be bound, and it is both more convenient and more flexible to examine locally g to see what variables it might need than to examine the lexical context of g within the larger program to see what variables are merely allowed to be free in g.
  - Transfer relations are actual computer-representable structures, and so for practical reasons these structures should be as small as possible. Restricting the environment to the free variables of g is a simple way to reduce potentially the textual size of the closure.

- $e(\vec{e})$. This case is quite different from the meta-rule. Execution from this term will transition to the function g of the closure to which e evaluates. Thus, the control part of the state after the transition not only depends on the environment part of the state before the transition, as is the case with conditionals, but is actually *taken from* the environment. This relationship is no problem for the meta-rule formulation of a transition system, because it is just another example of how one state in an execution depends in some fashion on the previous in the execution. But such transitions are difficult to express as a single-step transfer relation because the transfer relation itself is already parameterized over the two control points, in this case terms. In other words, when defining $\Delta_{t,g}$, specifying all transitions from a state at t into the function g (recall that the control point g is identified with g's body), one cannot express how the control point g depends on the environment at the beginning of the transition, because g is fixed.

In all transitions in Mini-C and all other transitions in Pure, the only dependency of the *control* part of the latter state on the *store* part of the former state (or environment part, in the case of Pure) is for conditionals, in which the store (or environment) in the former state determines which one of two possible control points is in the latter state. In contrast, function application is fundamentally more difficult. The reason is that the control point (or term, in Pure) to which execution proceeds is part of the store (or environment) itself. Thus, given an application term $t = e(\vec{e})$, one cannot extract the function g from e as in the meta-rule; rather, one must define $\Delta_{t,g}$ to implement the appropriate condition that e will indeed evaluate to a closure whose function is g.

This suggests a definition of the form

$$\Delta_{(e(\vec{e})),g} = \boxed{\texttt{code}_g(e)?\ \Delta}$$

for some Δ. The choice of Δ brings up the second difficulty with functions, and this time not limited only to first-class functions. Namely, the transition from function application to function body is the only time in which the environment is changed wholesale. This, too, is somewhat at odds with our particular language of transfer relations. We provided parallel assignment in the language of transfer relations to express a store modification, but not to replace an entire store with a new one. It is not as bad as it seems, however, because Pure uses environments, which contain only variable bindings. Furthermore, one can easily show that the only variables bound (to a non-`undef` value) when execution is at a term t are the variables *in the lexical scope of* t, a well-known concept that we do not define formally here.

Therefore, one solution would be to define a new nullary primitive operation `undef` that evaluates to value `undef` and then define Δ to bind all variables in the scope of $e(\vec{e})$ to `undef` and all variables in the scope of g to their appropriate value, with the latter taking precedence over the former for any variables in both scopes.

However, we choose a different solution, largely for practical reasons. The bisimulation above tells us that in any transition to g it is sufficient to ensure only that all free variables of g are bound correctly. The resulting execution may not be identical to the one given by the meta-rules, but will be equivalent modulo the bisimulation relation $\sim$. It is also easy to see that this does not affect the final result of the program. So we have the following definition.

$$\Delta_{(e(\vec{e})),g} = \boxed{\texttt{code}_g(e)?\ \boxed{z_1, \ldots, z_n \mapsto e_1, \ldots, e_n}}$$

if $g = x(\vec{y}) = t$, $\{z_1, \ldots, z_n\} = \mathbf{FV}(t)$, and

$$e_i = \begin{cases} e & \text{if } z_i = x \\ (\vec{e})_j & \text{if } z_i = (\vec{y})_j \\ \texttt{env}_{z_i}(e) & \text{otherwise} \end{cases}$$

Thus, the assignment relation, instead of replacing the environment wholesale, as done in the meta-rule, simply ensures that all of the free variables of the function are bound correctly.

- **if** e **then** t **else** t'. This case is just like the meta-rules; there are two single-step transfer relations, one from this term to t filtering the `true` condition, and the other from this term to t' filtering the `false` condition:

  $$\Delta_{(\textbf{if}\ e\ \textbf{then}\ t\ \textbf{else}\ t'),t} = \boxed{e?\ \bullet}$$

  $$\Delta_{(\textbf{if}\ e\ \textbf{then}\ t\ \textbf{else}\ t'),t'} = \boxed{e¿\ \bullet}$$

  Recall that $\boxed{e?\ \Delta}$ is an abbreviation for $\boxed{e?\ \Delta \,|\, \emptyset}$, and $\boxed{e¿\ \Delta}$ is an abbreviation for $\boxed{e?\ \emptyset \,|\, \Delta}$.

Symbolic evaluation of $\texttt{code}_g$ and $\texttt{env}_z$

Whenever one adds a new primitive operation to Primop, one needs to define its symbolic evaluation. Almost all primitives are context-independent, and it is safe to use the identity function for their symbolic evaluation. This is the case with $\texttt{closure}_{g,\vec{z}}$, $\texttt{code}_g$, and $\texttt{env}_z$, but in the case of the latter two it is important to perform some simple but very useful simplifications. We define their symbolic evaluations as follows.

$$\begin{aligned} \widetilde{\texttt{env}}_{z_i}(\texttt{closure}_{g,(z_1,\ldots,z_n)}(e_1, \ldots, e_n)) &= e_i \\ \widetilde{\texttt{code}}_g(\texttt{closure}_{g',\vec{z}}(e)) &= \begin{cases} \texttt{true} & \text{if } g = g' \\ \texttt{false} & \text{otherwise} \end{cases} \end{aligned}$$

This is similar to the symbolic evaluation of π_i that selects the ith component of a tuple.

$$\tilde{\pi}_i((e_1, \ldots, e_n)) = e_i$$

We will see why these simplifications are important in the following example.

A small example

Consider the PURE program

$$\textbf{rec}\ g\ \textbf{in}\ \texttt{f}(1)$$

where

$$g = (\texttt{f}(\texttt{x}) = \texttt{x} + \texttt{y}).$$

By the meta-rule formulation of the transition-system semantics, the execution of this program in an environment in which y is bound to n proceeds as follows.

$$\begin{array}{lll} & (\textbf{rec}\ g\ \textbf{in}\ \texttt{f}(1), \{\texttt{y} \mapsto n\} &) \\ \longmapsto & (\texttt{f}(1) \quad ,\{\texttt{f} \mapsto \langle g, \{\texttt{y} \mapsto n\}\rangle, \texttt{y} \mapsto n\} &) \\ \longmapsto & (\texttt{x} + \texttt{y} \quad ,\{\texttt{f} \mapsto \langle g, \{\texttt{y} \mapsto n\}\rangle, \texttt{x} \mapsto 1, \texttt{y} \mapsto n\}) & \end{array}$$

There are two transitions in this execution. The first one is described by the transfer relation

$$\Delta_{(\textbf{rec}\ g\ \textbf{in}\ \texttt{f}(1)),(\texttt{f}(1))} = \boxed{\texttt{f} \mapsto \texttt{closure}_{g,(\texttt{y})}(\texttt{y})}$$

and the second one is described by the transfer relation

$$\Delta_{(\texttt{f}(1)),g} = \boxed{\texttt{code}_g(\texttt{f})?\ \boxed{\texttt{x} \mapsto 1}}.$$

The composition of the two transitions is described by the transfer relation

$$\Delta_{(\textbf{rec}\ g\ \textbf{in}\ \texttt{f}(1)),(\texttt{f}(1)),g} = \Delta_{(\textbf{rec}\ g\ \textbf{in}\ \texttt{f}(1)),(\texttt{f}(1))} \oplus \Delta_{(\texttt{f}(1)),g}$$

If the symbolic evaluations of $\texttt{code}_g$ and $\texttt{env}_z$ performed no simplifications, then the $\oplus$ would return

$$\boxed{\texttt{code}_g(\texttt{closure}_{g,(\texttt{y})}(\texttt{y}))?\ \boxed{\texttt{f}, \texttt{x}, \texttt{y} \mapsto \texttt{closure}_{g,(\texttt{y})}(\texttt{y}), 1, \texttt{env}_{\texttt{y}}(\texttt{closure}_{g,(\texttt{y})}(\texttt{y}))}}$$

as this composition, which is correct but extremely cumbersome. However, with the symbolic evaluations we defined above, $\oplus$ returns

$$\boxed{\texttt{f}, \texttt{x}, \texttt{y} \mapsto \texttt{closure}_{g,(\texttt{y})}(\texttt{y}), 1, \texttt{y}}$$

which exploits the fact that the called function is known in order to both eliminate the dynamic condition on the control flow and to propagate statically the value of y through the closure.

A subtle point that is unrelated to these symbolic simplifications concerns the final binding of f. Note that:

- In the execution trace of 3 states shown above, the final environment contains a binding for f.

- The transfer relation shown immediately above describes that the net effect of this length-3 control path includes an assignment to `f`.
- The value bound to `f` is the same in both cases.

This may seem exactly as expected. After all, we did describe the meta-rules and the single-step transfer relations as alternate formulations of the same transition-system semantics. But as it turns out, the fact that the net effect of both formulations on `f` are equivalent is an accident in this case. The explanation lies in the bisimulation relation we defined earlier. For this case:

- In the meta-rule formulation, the second transition does a whole-scale replacement of the caller's environment with the closure of the callee and then extends this environment with both `f` and `x`, representing the passing of those two values to the callee.
- In the transfer-relation formulation, the second transition binds the free variables of the function body, which is the set $\{\mathtt{x}, \mathtt{y}\}$, but does not remove the binding of `f` that was present in the caller's environment.

In this case, the two bindings of `f` happen to be the same, but this will not generally be the case. However, the bisimulation relation tells us that in a state at term t we may simply "filter out" all bindings of variables not in $\mathbf{FV}(t)$, and then the correspondence between the meta-rule formulation and the transfer-relation formulation will be exact.

In this case, we could thus view the transition trace as

$$\begin{array}{lll} & (\mathbf{rec}\ g\ \mathbf{in}\ \mathtt{f}(1), & \{\mathtt{y} \mapsto n\} \qquad\qquad) \\ \longmapsto & (\mathtt{f}(1) & ,\{\mathtt{f} \mapsto \langle g, \{\mathtt{y} \mapsto n\}\rangle\}) \\ \longmapsto & (\mathtt{x} + \mathtt{y} & ,\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto n\} \qquad) \end{array}$$

and the composed transfer relation as

$$\boxed{\mathtt{x}, \mathtt{y} \mapsto 1, \mathtt{y}}.$$

As a final note, we make a note about the final state of execution. In general, the final state of an execution is a state

$$(e, \rho)$$

and the resulting value of the execution is a value v such that

$$e \vdash_\rho v.$$

In the transfer-relation formulation, we may use `E` to obtain an expression that represents the value of a term in terms of the free variables of the term. In the example above, this corresponds to

$$\mathsf{E}\,(\mathtt{x} + \mathtt{y})\,\boxed{\mathtt{f}, \mathtt{x}, \mathtt{y} \mapsto \mathtt{closure}_{g,(\mathtt{y})}(\mathtt{y}), 1, \mathtt{y}}$$

which returns

$$1 + \mathtt{y}.$$

6.5 Variable Renaming vs. Closures

The semantics that we have given for PURE does not involve variable renaming. For instance, the term

$$\textbf{let}\ \mathtt{x} = 1\ \textbf{in}\ \mathtt{x + z}$$

and the term

$$\textbf{let}\ \mathtt{y} = 1\ \textbf{in}\ \mathtt{y + z}$$

are distinguished apart in PURE, although they differ merely by the choice of variable name. In fact, these two distinct terms induce two distinct families of transitions. The first term induces the family of transitions

$$(\textbf{let}\ \mathtt{x} = 1\ \textbf{in}\ \mathtt{x + z}, \rho) \longmapsto (\mathtt{x + z}, \rho[\mathtt{x} \mapsto 1])$$

ranging over environments ρ, while the second term induces the family of transitions

$$(\textbf{let}\ \mathtt{y} = 1\ \textbf{in}\ \mathtt{y + z}, \rho) \longmapsto (\mathtt{y + z}, \rho[\mathtt{y} \mapsto 1]).$$

But this may seem strange. If ρ already has a binding for the variable in question (`x` for the first case and `y` for the second case) then what assurance do we have that that binding is no longer needed and may be discarded by the environment update? One may expect instead a meta-rule for let-binding transitions that looks something like

$$\frac{e \vdash_\rho v \quad \rho\, x' = \mathtt{undef}}{(\textbf{let}\ x = e\ \textbf{in}\ t, \rho) \longmapsto (t[x'/x], \rho[x' \mapsto v])}$$

where $t[x'/x]$ substitutes the variable x' for all free occurrences of the variable x. Note that the rule does not have the syntactic non-interference condition $x' \notin \mathbf{FV}(t)$ because it is covered by the semantic non-interference condition that $\rho\, x' = \mathtt{undef}$. The notion of variable renaming is based on *α-conversion* of the λ-calculus [Bar84].

We can get away without variable renaming, however. First we describe how we achieve this and compare this choice with a semantics based upon variable renaming, and then we explain why it is desirable for our purposes of program analysis to avoid the need for variable renaming.

In this section, we will need a notion of how to examine a transition system to determine that it is reasonable. We will start by defining a notion of well formed states, and then we apply the following test to the transition system.

Definition 12 (Preservation of well-formedness) *Given a notion of well-formedness on states, a transition system is said to* preserve well-formedness *if, for all well formed states ψ, $\psi \longmapsto \psi'$ implies that ψ' is well formed.*

Our semantics for PURE uses the following notion of well formed states.

Definition 13 (Well formed states (#1)) *A state (t, ρ) is well formed iff ρ contains the correct bindings for all free variables of t (in other words, all $x \in \mathbf{FV}(t)$).*

When we say that a binding is "correct" we mean that it is the binding that one would intuitively expect from an execution of the program in question.

Now, it is easy to see that the semantics that we have given for PURE preserves well-formedness. For instance, consider the rule for let-binding transitions.

$$\frac{e \vdash_\rho v}{(\mathbf{let}\ x = e\ \mathbf{in}\ t, \rho) \ \longmapsto\ (t, \rho[x \mapsto v])}$$

Note that $x \notin \mathbf{FV}(\mathbf{let}\ x = e\ \mathbf{in}\ t)$ and $\mathbf{FV}(t) \subseteq (\mathbf{FV}(\mathbf{let}\ x = e\ \mathbf{in}\ t) \cup \{x\})$. Therefore, if ρ contains the correct bindings for each $y \in \mathbf{FV}(\mathbf{let}\ x = e\ \mathbf{in}\ t)$, then $\rho[x \mapsto v]$ will contain

- the correct bindings for each $y \in \mathbf{FV}(\mathbf{let}\ x = e\ \mathbf{in}\ t)$ and
- the correct binding for x,

and thus will contain the correct bindings for each $y \in \mathbf{FV}(t)$.

The rule for **rec** g **in** t is analogous. It is easy to see that the rule for function application works.

$$\frac{e \vdash_\rho v \quad (\vec{e})_i \vdash_\rho (\vec{v})_i \quad v = \langle x(\vec{y}) = t, \rho' \rangle}{(e(\vec{e}), \rho) \ \longmapsto\ (t, \rho'[x \mapsto v][\vec{y} \mapsto \vec{v}])}$$

Note that ρ is discarded in the transition. The well-formedness of the state $(e(\vec{e}), \rho)$ thus merely ensures that function e and the arguments $\vec{e}$ evaluate to correct values. To show that the state $(t, \rho'[x \mapsto v][\vec{y} \mapsto \vec{v}])$ is well formed, we must reason that, because ρ' came from a previous well formed state $(\mathbf{rec}\ x(\vec{y}) = t\ \mathbf{in}\ t', \rho')$, that ρ' contains the correct bindings for all $z \in \mathbf{FV}(\mathbf{rec}\ x(\vec{y}) = t\ \mathbf{in}\ t')$ and thus all $z \in \mathbf{FV}(x(\vec{y}) = t)$. Therefore, $\rho'[x \mapsto v][\vec{y} \mapsto \vec{v}]$ contains the correct bindings for $\mathbf{FV}(t)$.

This discussion explains the purpose of closures, which save the bindings in ρ' that must be restored upon function application.

Alternatively, we could dispense with closures altogether, representing a function at run time as the function term g instead of the closure $\langle g, \rho \rangle$. This leads to a more complex notion of well formed states.

Definition 14 (Well formed states (#2)) *A state (t, ρ) is well formed iff both*

- *ρ contains the correct bindings for all $x \in \mathbf{FV}(t)$, and*
- *ρ contains the correct bindings for all $x \in \mathbf{FV}(g)$ such that $\rho\, y = g$ for some variable y.*

Intuitively, we "flatten out" all the closure environments into a single global environment that is threaded through the execution. To accomplish this, we will need to create fresh variables on the fly, which means that we will need to rename variables at run time.

For instance, the old rule for let-binding

$$\frac{e \vdash_\rho v}{(\textbf{let}\ x = e\ \textbf{in}\ t, \rho) \ \longmapsto\ (t, \rho[x \mapsto v])}$$

no longer works. As, we explained above, $x \notin \mathbf{FV}(\textbf{let}\ x = e\ \textbf{in}\ t)$ and thus x may be safely overwritten under the first notion of well formed states. But with this second, more restricted notion of well formed states, we cannot be sure that x is not a free variable of some function g in the range of ρ. Thus, we must rename x to a fresh variable. We showed the resulting rule earlier in this section:

$$\frac{e \vdash_\rho v \quad \rho\, x' = \texttt{undef}}{(\textbf{let}\ x = e\ \textbf{in}\ t, \rho) \ \longmapsto\ (t[x'/x], \rho[x' \mapsto v])}$$

It is rather easy to see that this rule preserves well-formedness under the second notion because it never destroys any values in ρ.

But now we do not need closures, and the rule for function application threads the current environment ρ through, again renaming the newly bound variables to avoid clashes with variables already bound in ρ. For simplicity of illustration, we show the case for single-argument functions:

$$\frac{e \vdash_\rho v \quad e' \vdash_\rho v' \quad v = (x(y) = t) \quad \rho\, x' = \texttt{undef} \quad \rho\, y' = \texttt{undef}}{(e(e'), \rho) \ \longmapsto\ (t[x'/x][y'/y], \rho[x' \mapsto v][y' \mapsto v'])}$$

Once again, it is easy to see that this rule preserves well-formedness, because once again we rename the bound variables appropriately such that ρ is extended rather than updated.

We have just presented (most of) a different style of transition-system semantics for PURE in which we have traded closures for dynamic variable renaming. The resulting semantics is arguably cleaner and more natural, but we have only considered the meta-rule formulation of the semantics. To see the fundamental difficulty, consider what the single-step transfer relation

$$\Delta_{(\textbf{let}\ x=e\ \textbf{in}\ t),t}$$

should be. Without variable renaming it is

$$\boxed{x \mapsto e},$$

but with variable renaming is must be something like

$$\boxed{x\ \texttt{=}\ \texttt{undef}?\ \boxed{x \mapsto e}}$$

to perform the dynamic test that x does not need to be renamed.

But then the composed two-step transfer relation

$$\Delta_{(\textbf{let}\ x=1\ \textbf{in}\ \textbf{let}\ x=2\ \textbf{in}\ t),(\textbf{let}\ x=2\ \textbf{in}\ t),t}$$

will be the empty relation

$$\emptyset$$

instead of the expected

$$\boxed{x \mapsto 2}$$

because the variable x was not renamed along the given length-three control path.

So, in summary, the semantics based on variable renaming has the following ramifications on the transfer-relation form of the semantics:

- Whenever an analysis composes the transfer relation for a control path Γ, it is the responsibility of the analysis to rename (statically) the variables in the terms along Γ in a way that is guaranteed to capture all possible behaviors of any dynamic renaming of the terms in the meta-rule semantics. For instance, in the above length-three control path, the second occurrence of x must be renamed to a variable that is not in $\mathbf{FV}(t)$.
- A number of tests of the form x `=` `undef` will accumulate during composition of a control path Γ, complicating the presentation of the transfer relation. These tests are necessary for any fixed control path Γ because the transfer-relation terms in TR have no facility to be dynamically renamed, and hence each term in TR is defined only on initial environments for which the given choice of variables is already satisfactory. But the accumulation of these tests is a practical disadvantage.

It is because of these factors that we choose a semantics that does not have variable renaming and thus needs closures to store multiple dynamic occurrences of the same static variable.

It is possible, however, that there is a different semantic approach, based on a different treatment of PURE variables in the transfer relations, that would not suffer the above factors. For instance, perhaps the environment could be represented as a list, accessed by de Bruijn indices [dB72] instead of variables.

Chapter 7

Extending Pure with Mutable Records and Arrays

Imperative features are crucial components of almost all languages, even "functional" languages such as Scheme and Standard ML. In this chapter we extend Pure with both assignable arrays and records with assignable fields. We call the resulting language Impure.

7.1 Syntax

We first add some new terms to Pure.

$$
\begin{array}{rcll}
t & ::= & \ldots & \\
 & \mid & \textbf{let}\ x = \{f_1 = e_1, \ldots, f_n = e_n\}\ \textbf{in}\ t & \text{record creation} \\
 & \mid & e.f := e';\ t & \text{record field assignment} \\
 & \mid & e.f & \text{record dereference} \\
 & \mid & \textbf{letarray}\ x\ \textbf{in}\ t & \text{array creation} \\
 & \mid & e[e'] := e'';\ t & \text{array update} \\
 & \mid & e[e'] & \text{array dereference} \\
 & & & \\
f & \in & \mathsf{Field} & \text{field names}
\end{array}
$$

The array-creation term creates an array whose elements are initially the undefined value `undef`. For simplicity, arrays do not have bounds and are conceptually infinite.

7.2 Discussion

Records in Impure are not like records in SML; the former are mutable, but the latter are immutable. It would be simple to add SML records, because they are just a variant on tuples.

Like Pure tuples, there would be a special kind of value to model immutable records, supported by context-independent primitive operations.

In Section 4.5, we gave a discussion of various kinds of errors, and the way in which they would be handled with our semantic methodology. We now return to some of these points.

It may seem strange that arrays are infinite, and that its elements are initially `undef`. In contrast, the array creation function in SML takes both a size argument and an argument providing the value to which all elements are initialized. It is not difficult to add a size component to our arrays, which could be checked at run time for out-of-bounds errors. But an initialization value would require a model for arrays that includes an explicit default value. This would render the dereference and assignment operations nonuniform, and thus their syntactic occurrences in transfer relations would be cumbersome. Therefore, we require that programmers write their own initialization routine.

This language is rather primitive in that there is no static typechecking on records and arrays do not have bounds. For instance, the Impure program

$$\begin{array}{l}\textbf{let } x = \{\texttt{car} = 1,\ \texttt{cdr} = 2\} \\ \textbf{in } x.\texttt{bad} := 3;\ t\end{array}$$

first creates a two-field record and then adds a third field. Also, the program

$$\begin{array}{l}\textbf{let } x = \{\texttt{car} = 1,\ \texttt{cdr} = 2\} \\ \textbf{in let } y = x.\texttt{bad} \textbf{ in } t\end{array}$$

actually binds y to the undefined value `undef` and proceeds to execute t. The behavior of arrays is similar. For instance, the Impure program

$$\begin{array}{l}\textbf{letarray } x \\ \textbf{in let } y = x[3] \\ \textbf{in } x[200] := 55;\ x[200]\end{array}$$

binds y to `undef` (dereferencing an uninitialized array element), assigns 55 to the array element 200, and then successfully dereferences the element, returning 55 as the result of the program.

Of course no reasonable language would function in this manner. Augmenting this language with a static type system similar to SML would reject programs that referenced incorrect field names. But the situation with arrays is more serious, as proper handling of both uninitialized elements and bounds checking must be relegated to run-time, and thus to the dynamic semantics. Our decision to simplify the situation by using infinite arrays is a compromise aimed to simplify the transfer relations that we will develop to model the dynamic semantics of the language. A full language would have a mechanism for exception handlers, to which control would flow in the case of array-bounds errors.

7.3 Syntax Simplification

As with Mini-C, it will be more convenient to define the semantics of these new features if we rewrite the syntax to conform more closely to our language of transfer relations.

We would like to consider the terms $e.f$ and $e[e']$ as expressions. To do this, we need to add the following primitive operations to Primop.

- All field names $f \in \mathsf{Field}$ as constant nullary operations.
- The context-dependent binary operation `deref`.

Now we rewrite the term $e.f$ as the expression $\texttt{deref}(e, f)$ and the term $e[e']$ as the expression $\texttt{deref}(e, e')$.

Similarly, we consider the $e.f$ on the left-hand side of a record-field assignment to be an l-expression and rewrite the $e[e']$ on the left-hand side of an array assignment as the l-expression $e.e'$.

Our next transformation is to "compile" a record-allocation term in exactly the same way as we did for Mini-C. To review, we need the following:

- for each natural number m, a pointer value $\langle m\rangle \in \mathsf{Val}$
- the unary primitive operation `ptr` that casts an integer m to the pointer $\langle m\rangle$ (formally, $\texttt{ptr}(m) \hookrightarrow \langle m\rangle$), where we write $\langle e\rangle$ for $\texttt{ptr}(e)$
- a distinguished value $\diamond \in \mathsf{Val}$, to be used only in the reference $\diamond.\diamond$ (written as $\Diamond$) which is initialized to 1 at the beginning of execution and always holds the integer of the next free pointer on the heap

We now perform the following transformation of a Pure program t. We first transform the program to

$$\mathbf{let}\ \Diamond = 1\ \mathbf{in}\ t$$

and then rewrite every subterm

$$\mathbf{let}\ x = \{f_1 = e_1, \ldots, f_n = e_n\}\ \mathbf{in}\ t$$

in t as the term

$$\begin{array}{l}\mathbf{let}\ x = \langle\Diamond\rangle\\ \mathbf{in}\ x.f_1 := e_1;\ x.f_n := e_n;\ \Diamond := \Diamond \texttt{+} 1;\ t\end{array}$$

Next we "compile" arrays in a similar fashion. We rewrite every subterm

$$\mathbf{letarray}\ x\ \mathbf{in}\ t$$

as the term

$$\mathbf{let}\ x = \langle\Diamond\rangle\ \mathbf{in}\ \Diamond := \Diamond \texttt{+} 1;\ t.$$

Note that this treatment of allocation does not equate as many programs as one might reasonably expect. For instance,

$$\mathbf{let}\ x = \{\texttt{car} = 3,\ \texttt{cdr} = 4\}\ \mathbf{in}\ y$$

and

$$\textbf{let } x = \{\texttt{car} = 1, \texttt{cdr} = 2\} \textbf{ in let } y = \{\texttt{car} = 3, \texttt{cdr} = 4\} \textbf{ in } y$$

have different meanings because they use different pointer values to construct the record bound to y. This is a simple case of the well studied problem with full abstraction for languages that combine assignment and procedures [OT95, Sie94].

At this point, we have simplified the new imperative constructs into a set of primitive operations and a generic assignment term. Here is the final extension of Pure syntax for transformed Impure programs.

$$\begin{array}{rcll} t & ::= & \ldots & \\ & | & e.e' := e'';\ t & \text{assignment} \\ \\ p & ::= & \ldots & \\ & | & f & \text{field name (nullary)} \\ & | & \texttt{deref} & \text{dereference (binary)} \\ & | & \texttt{ptr} & \text{allocation (unary)} \end{array}$$

7.4 Semantics

The semantics of Pure required only an environment mapping variables to values. But like Mini-C, the mutable data structures require the expressiveness of stores. There are thus two steps to define the semantics of these new imperative features:

- Recast the semantics of Pure in terms of stores rather than environments, in order to support the mutable data structures of Impure.
- Give the semantics of the assignment term $e.e' := e'';\ t$.

It is fairly straightforward to recast the semantics of Pure to use stores instead of environments. As we described in the design of Pure, an environment is just a restricted form of store. Indeed, the isomorphism

$$\mathsf{Store} \simeq \mathsf{Env} \times \mathsf{Heap}$$

where

$$\mathsf{Heap} = \mathsf{Val} \times \mathsf{Val} \to \mathsf{Val}$$

makes this recasting convenient. The "heap" handles the bindings of references.[1] Below are the meta-rules of Pure rewritten such that a state pairs a term with a store rather than with an environment:

$$\mathsf{State} = \mathsf{Term} \times \mathsf{Store}$$

[1] Note that in the literature, a heap is often called a store. But we have already used the term "store" for something else.

In the following meta-rules, we freely switch notation between the isomorphic forms $\sigma \in \mathsf{Store}$ and $(\rho, \gamma) \in \mathsf{Env} \times \mathsf{Heap}$. The only rules that are not essentially identical to the corresponding rule of Pure are those for function creation and application. The interesting part about those rules is that only the environment component of the store is saved in the closure and restored upon application; the "heap" component is instead threaded through.

$$\frac{e \vdash_\sigma v}{(\mathbf{let}\ x = e\ \mathbf{in}\ t, \sigma) \longmapsto (t, \sigma[x \mapsto v])}$$

$$\frac{\sigma \simeq (\rho, \gamma)}{(\mathbf{rec}\ x(\vec{y}) = t\ \mathbf{in}\ t', \sigma) \longmapsto (t, \sigma[x \mapsto \langle x(\vec{y}) = t', \rho\rangle])}$$

$$\frac{e \vdash_\sigma v \quad (\vec{e})_i \vdash_\sigma (\vec{v})_i \quad v = \langle x(\vec{y}) = t, \rho'\rangle \quad \sigma \simeq (\rho, \gamma)}{(e(\vec{e}), \sigma) \longmapsto (t, (\rho'[x \mapsto v][\vec{y} \mapsto \vec{v}], \gamma))}$$

$$\frac{e \vdash_\sigma \texttt{true}}{(\mathbf{if}\ e\ \mathbf{then}\ t\ \mathbf{else}\ t', \sigma) \longmapsto (t, \sigma)}$$

$$\frac{e \vdash_\sigma \texttt{false}}{(\mathbf{if}\ e\ \mathbf{then}\ t\ \mathbf{else}\ t', \sigma) \longmapsto (t', \sigma)}$$

All that remains is to give the rule for the generic assignment term, which is quite straightforward:

$$\frac{e \vdash_\sigma v \quad e' \vdash_\sigma v' \quad e'' \vdash_\sigma v''}{(e.e' := e'';\ t, \sigma) \longmapsto (t, \sigma[v.v' \mapsto v''])}$$

Again, we would like to give a formulation of this transition system in terms of single-step transfer relations instead of meta-rules. Because we designed transfer relations to be relations on stores and not simply environments, the single-step transfer relations for Pure work without change for Impure. But recall that those definitions made use of the bisimulation relation $\sim$ on both values and stores. So, two tasks remain:

- Extend the bisimulation relation to states with stores.
- Define the single-step transfer relations for the generic assignment term.

The first task is straightforward; two heaps are similar if all corresponding nodes (values) are similar. This is given by the following definition.

Definition 15 (Similar states with general stores) *Two states $(t, (\rho, \gamma))$ and $(t', (\rho', \gamma'))$ are said to be* similar *(written $(t, (\rho, \gamma)) \sim (t', (\rho', \gamma'))$) if $(t, \rho) \sim (t', \rho')$ and $\gamma(v.v') \sim \gamma'(v.v')$ for all values $v, v' \in \mathsf{Val}$.*

The second task is also straightforward, as it is very similar to the assignment statement in Mini-C.

$$\Delta_{(e.e':=e'';\, t),t} = \boxed{e.e' \mapsto e''}$$

Note that the transfer relations conceptually treat both let binding and assignment as special cases of assignment of l-expressions.

7.5 Final Words on First-Class Functions

We have seen several ways in which our model of a language with first-class functions is not quite natural. For one, the transfer relations that save an environment in a closure and restore the environment upon function application process each free variable separately. This approach is an artifact of the natural choice to model language variables with store variables. Alternatively, one could model an environment as a record whose field names are the language variables, and maintain only a single store variable `E` that is always bound to the environment. This requires an extra level of indirection, through `E`, for variable operations. But for that price, one gains the ability to save and restore environments in closures easily, because they are simply records.

Related to this issue is the l-value $\Diamond$ that points to the index of the next free heap location, used for the creation of new records and arrays. In Mini-C, we used the variable `H` for this purpose, but we cannot use a variable in a language with first-class functions because in that case it could be saved in a closure and restored on function application. Instead, we need a global assignable variable. To avoid the need to treat a distinguished variable differently from all others, we instead chose to use a reference $\Diamond = \diamond.\diamond$. Again, this problem would have a nicer solution if we modeled environments as we suggested in the preceding paragraph. Then there would be only two variable bindings in the store: `E`, bound to the current environment, and `H`, bound to the index of the next free heap location.

Part IV

Analysis Applications

Chapter 8

Multi-step Program Analysis

In Chapter 1 we isolated a methodological difficulty with program analyses: they apply an abstraction between every execution step of the analyzed program. We explained that this severely cripples the quality of an analysis on source programs for which a desired property is temporarily weakened during a period of a few program steps. As an example, we gave a simple analysis of the signs of integer-valued variables, but we also explained that this is a problem for other kinds of analyses, such as shape analyses.

For instance, consider the following IMPURE program that destructively reverses a binary tree.

```
rec reverse(x, k) =
    if leaf(x)
       then k()
       else rec k1() =
                 rec k2() =
                      let temp = x.l
                      in x.l := x.r;
                         x.r := temp;
                         k()
                 in reverse(x.r, k2)
       in reverse(x.l, k1)
in reverse(x, k)
```

One would like a shape analysis to determine that when `reverse` is called with a data structure that actually is a binary tree, that the data structure is still a tree on termination of the procedure. Most shape analyses cannot determine this information. To our knowledge, only [SRW96] can achieve this result, but it is highly specialized for this and similar cases and requires quite restrictive conditions, as explained in Section 1.1.

But for now we wish to point out why this program is so difficult to analyze. Consider the following informal description of what happens every time execution reaches term 6.

6 : `x` is a tree with subtrees L and R
7 : `x` is a tree with subtrees L and R
8 : `x` is not a tree: its left and right links both point to subtree R
9 : `x` is a tree with subtrees R and L

Program analyses infer, or abstract, a property at every step, and so it is difficult to cope with the states at term 8. An analysis would need to have the ability to describe the special property at 8 with sufficient detail to infer that the assignment at 8 changes `x` back to a tree. In fact, that is what [SRW96] does to solve this particular problem.

But there is a much more general solution, and that is to avoid the *necessity* to infer a property at every step, and instead allow multiple steps of execution before abstracting. In order to explain why this is not already a part of program-analysis methodology, we must take a step back and examine the foundations of semantics-based program analysis.

8.1 A Review of Abstract Interpretation

Abstract interpretation [CC77] is a general framework for expressing semantics-based program analyses. In fact it is more than that; it is a general framework for relating different semantics of a language, some of which may be effectively computable for all programs and therefore in general approximate, or *inadequate*, as a semantic definition of the language. Such computable "semantics" are program analyses, and with abstract interpretation they are always related to some adequate semantics of the language.

A semantics of a language is a function

$$\mathcal{M} \in \mathsf{Prog} \to \mathsf{SemObj}$$

mapping program texts to *semantic objects*. The main observation of abstract interpretation is that $\mathcal{M}[\![P]\!]$ is usually defined as a fixed point, and the potential that its iterative definition may be transfinite directly reflects the potential that P may not terminate. In other words,

$$\mathcal{M}[\![P]\!] = \mathrm{fix}\,(\mathcal{S}[\![P]\!])$$

where

$$\mathcal{S} \in \mathsf{Prog} \to \mathsf{SemObj} \to \mathsf{SemObj}$$

and SemObj is equipped with a partial order and fix computes some fixed point of its parameter—usually the least, but sometimes the greatest, depending on the particular semantics.

Abstract interpretation explains how to relate such a semantics to a more *abstract* semantics. One first designs a partial order $\widehat{\mathsf{SemObj}}$ of *abstract semantic objects* and then defines the function

$$\alpha \in \mathsf{SemObj} \to \widehat{\mathsf{SemObj}}$$

called the *abstraction function* that projects a semantic object onto the abstract semantic domain. If α is additive, then one can induce a unique corresponding *concretization function*

$$\gamma \in \widehat{\mathsf{SemObj}} \to \mathsf{SemObj}$$

defined as

$$\gamma\, y = \sqcup_{\mathsf{SemObj}}\{x \in \mathsf{SemObj} \mid (\alpha\, x) \sqsubseteq_{\widehat{\mathsf{SemObj}}} y\}$$

that "coerces" an abstract semantic object into the more concrete semantic domain. Then from [CC79], the function

$$\hat{\mathcal{S}} \in \mathsf{Prog} \to \widehat{\mathsf{SemObj}} \to \widehat{\mathsf{SemObj}}$$

defined as

$$\hat{\mathcal{S}}[\![P]\!] = \alpha \circ \mathcal{S}[\![P]\!] \circ \gamma$$

corresponds to $\mathcal{S}$ in such a way that the fixed point fix $(\hat{\mathcal{S}}[\![P]\!])$ is an abstraction of the semantics $\mathcal{M}[\![P]\!]$. In other words, $\alpha(\mathcal{M}[\![P]\!])$ implies the property fix $(\hat{\mathcal{S}}[\![P]\!])$. We omit the formal details of this correspondence and refer the reader to [CC79]. Intuitively, an abstract semantic object is like a semantic object, but with some information missing, and $\hat{\mathcal{S}}[\![P]\!]$ first applies $\mathcal{S}[\![P]\!]$ to the information still present and then abstracts the result. We give an example below.

Sometimes, the information missing from abstract semantic objects is not necessary to model the language. For instance, much of the study of pure semantics is concerned with finding semantic objects that are as abstract as possible while still adequate as a semantic definition; the ultimate goal here is *full abstraction* [Mul87]. But for the purpose of program analysis, it is necessary to abstract away crucial information for the sake of computability. The choice of what to abstract away defines the program analysis.

A central intuition is that the function $\mathcal{S}[\![P]\!]$ typically corresponds in some sense to a "step" of an execution of P. We cannot formalize this correspondence because that would require a semantic definition of "execution step" in the first place, resulting in a meaningless circular definition. Nevertheless, this intuition is an invaluable aid in visualizing a semantic definition. In fact, the word "interpretation" in abstract interpretation comes from this intuition, because one can view the repeated applications of $\mathcal{S}[\![P]\!]$ in its iterative fixed-point computation as the steps of an interpreter, or an "abstract interpreter" in the case of $\hat{\mathcal{S}}[\![P]\!]$.

8.2 Abstract Interpretation of Transition Systems

The preceding discussion does does not specify or even impose any serious limitations on the semantic objects. Because the seminal work on abstract interpretation ([CC77]) uses a rather simple transition-system semantics for expository purposes, abstract interpretation is often misunderstood to be limited to flowchart-based semantics of while-loop languages. However, appearing soon after that seminal paper, Patrick Cousot's thesis ([Cou78]) showed the full maturity of the framework.

But in Chapter 4 we argued that a transition system is indeed particularly useful as a basis for program analysis, despite much work elsewhere. Our methodology is designed around transition-system semantics, and so we would like to examine abstract interpretations based on transition systems.

As explained in Chapter 4, a state of a transition system is a pair of a control point and a store:

$$\mathsf{State} = \mathsf{CtrlPoint} \times \mathsf{Store}$$

The transition relation defines the single execution steps of a particular program P as pairs of states:

$$\longmapsto \;\subseteq (\mathsf{State} \times \mathsf{State})$$

As introduced in Chapter 1, the transition-system semantics of a language is a function

$$\mathcal{M} \in \mathsf{Prog} \to \mathcal{P}(\mathsf{State}^*)$$

that, given a program P, returns a set of finite execution prefixes, represented as state sequences, defined inductively by unfolding the transition relation from a base set of initial states (length-one sequences):

$$\frac{\vec{\psi}.\psi \in \mathcal{M}[\![P]\!] \quad \psi \longmapsto \psi'}{\vec{\psi}.\psi.\psi' \in \mathcal{M}[\![P]\!]}$$

Above, we claimed that $\mathcal{M}[\![P]\!]$ should be expressible as a fixed point $\mathrm{fix}\,(\mathcal{S}[\![P]\!])$. Here,

$$\mathcal{S}[\![P]\!] \in \mathcal{P}(\mathsf{State}^*) \to \mathcal{P}(\mathsf{State}^*)$$

is the function

$$\mathcal{S}[\![P]\!]\;\vec{\Psi} = \vec{\Psi} \cup \{\vec{\psi}.\psi.\psi' \mid \vec{\psi}.\psi \in \vec{\Psi} \wedge \psi \longmapsto \psi'\}$$

defined by the above rule to perform one inductive application of the rule. Then the semantics $\mathcal{M}[\![P]\!]$ of program P is the least fixed point of $\mathcal{S}[\![P]\!]$ above a set Ψ_0 of initial states, and is precisely the set of (unbounded) finite prefixes of executions starting at Ψ_0.

For the purposes of abstract interpretation, the set SemObj of semantic objects is the set $\mathcal{P}(\mathsf{State}^*)$ of sets of finite execution sequences, ordered by inclusion. An abstract interpretation must provide a partial order $\widehat{\mathsf{SemObj}}$ of abstract semantic objects and an abstraction function

$$\alpha \in \mathcal{P}(\mathsf{State}^*) \to \widehat{\mathsf{SemObj}}.$$

The rest of the abstract interpretation is mechanical. Suppose $\hat{\Psi}$ is the least fixed point of

$$(\alpha \circ \mathcal{S}[\![P]\!] \circ \gamma) \in \widehat{\mathsf{SemObj}} \to \widehat{\mathsf{SemObj}}$$

above an initial abstract semantic object $\hat{\Psi} \in \widehat{\mathsf{SemObj}}$ such that $\Psi_0 \subseteq (\gamma\,\hat{\Psi})$. Then as we explained in the previous section, $\hat{\Psi}$ abstracts $\mathcal{M}[\![P]\!]$. In other words, $\alpha(\mathcal{M}[\![P]\!])$ implies the property $\hat{\Psi}$, or, equivalently, $\mathcal{M}[\![P]\!] \subseteq (\gamma\,\hat{\Psi})$.

8.3 Invariant Properties

Most program analyses compute *invariant properties*, or properties of the states that occur during program execution. In this case, it is convenient to perform the above abstraction in two steps. The first step abstracts a set of execution sequences by the set of states appearing in the sequences. In other words, the abstract semantic object is $\mathcal{P}(\mathsf{State})$, and the abstraction function

$$\alpha \in \mathcal{P}(\mathsf{State}^*) \to \mathcal{P}(\mathsf{State})$$

is defined as

$$\alpha\,\vec{\Psi} = \{\psi \mid \exists\vec{\psi} \in \vec{\Psi}.\ \psi \text{ appears in } \vec{\psi}\}.$$

The concretization function

$$\gamma \in \mathcal{P}(\mathsf{State}) \to \mathcal{P}(\mathsf{State}^*)$$

is induced from α as described above. Pushing this through abstract interpretation defines a function

$$\mathcal{S} \in \mathsf{Prog} \to \mathcal{P}(\mathsf{State}) \to \mathcal{P}(\mathsf{State})$$

defined as

$$\mathcal{S}[\![P]\!]\,\Psi = \{\psi' \mid \psi \in \Psi \land \psi \longmapsto \psi'\}$$

whose least fixed point $\mathcal{M}[\![P]\!] \in \mathcal{P}(\mathsf{State})$ above a set Ψ_0 of initial states is precisely the set of states reached during an execution from an initial state in Ψ_0.

An invariant property is thus a superset of $\mathcal{M}[\![P]\!]$, which is given by an abstract interpretation.

8.4 An Example

As an example, we consider the example in Chapter 1 of the analysis of the signs of integer variables. In this example, an execution state comprises a *control point*, specifying the syntactic point of execution, and an environment.

$$\psi \quad \in \quad \mathsf{State} \quad = \quad \mathsf{CtrlPoint} \times \mathsf{Env}$$

The step function $\mathcal{S}[\![P]\!]$ of a program P maps a set of states to their successors as given in the previous section. The semantics $\mathcal{M}[\![P]\!]$ is the least fixed point of $\mathcal{S}[\![P]\!]$ above $\{(C_0, \rho_0)\}$; it is the set of states reachable during an execution from the initial control point C_0 and environment ρ_0.

Following the example in Section 1.1, we define an abstract semantic object as a table of sign environments indexed by control point.

$$\hat{\Psi} \quad \in \quad \widehat{\mathsf{State}} \quad = \quad \mathsf{CtrlPoint} \to \mathsf{Var} \to \mathsf{Sign}$$

Here, Sign is the complete lattice given in Section 1.1. The function $\alpha \in \mathcal{P}(\mathsf{State}) \to \widehat{\mathsf{State}}$ abstracts a set of states by choosing for each control point C and variable x the strongest sign property satisfied by all bindings of x in environments at C.

$$\alpha\,\Psi\,C\,x = \bigwedge\{\hat{n} \in \mathsf{Sign} \mid (C, \rho) \in \Psi \Rightarrow (\rho\,x) = n \Rightarrow n \in \hat{n}\}$$

Here is an example of an abstraction of a set of three states, two of which have the same control point.

$$\alpha \begin{pmatrix} \{(C_1, [\mathtt{x}, \mathtt{y} \mapsto 2, 3]), \\ (C_1, [\mathtt{x}, \mathtt{y} \mapsto -1, 4]), \\ (C_2, [\mathtt{x}, \mathtt{y} \mapsto 0, 0])\} \end{pmatrix} = \begin{bmatrix} C_1, & \to & [\mathtt{x}, \mathtt{y} \mapsto \mathtt{int}, \mathtt{pos}], \\ C_2 & & [\mathtt{x}, \mathtt{y} \mapsto \mathtt{zero}, \mathtt{zero}] \end{bmatrix}$$

As we explained above, the concretization function $\gamma \in \widehat{\mathsf{State}} \to \mathcal{P}(\mathsf{State})$ in induced from α as

$$\gamma\,\hat{\Psi} = \{\psi \mid \alpha\,\{\psi\} \sqsubseteq \hat{\Psi}\}$$

where $\sqsubseteq$ is pointwise inclusion (in other words, pointwise property implication), but for illustration we give the alternate definition that intuitively expands the sign properties into the integers that they represent:

$$\gamma\,\hat{\Psi} = \{(C, \rho) \mid (\rho\,x) = n \Rightarrow n \in (\hat{\Psi}\,C\,x)\}$$

Another way of thinking about γ is that it specifies the states that are consistent with the given sign properties.

Next, $\hat{\mathcal{S}}[\![P]\!]$ is defined mechanically:

$$\hat{\mathcal{S}}[\![P]\!] = \alpha \circ \mathcal{S}[\![P]\!] \circ \gamma$$

In other words, $\hat{\mathcal{S}}[\![P]\!]$, given an abstract semantic object $\hat{\Psi}$, first applies γ, yielding all the states consistent with the sign properties in $\hat{\Psi}$, then applies the transition relation $\longmapsto$ to these states, yielding their successors, and finally applies α to these successor states, abstracting them by a semantic object $\hat{\Psi}'$ describing their sign properties.

Given an initial abstract semantic object $\hat{\Psi}_0$ such that $(C_0, \rho_0) \in (\gamma\,\hat{\Psi}_0)$, the least fixed point of $\hat{\mathcal{S}}[\![P]\!]$ above $\hat{\Psi}_0$ gives sign properties that hold during the execution of P. For example, if P is the while-loop program presented earlier, the result of the analysis is the the table of five sign environments shown in Section 1.1 next to their respective program points, with the last one corresponding to the "exit" program point.

It is worth considering again the analogy given in Section 1.1 of computing the rounded sum of a list of numbers. In this analogy, a semantic object $\Psi \in \mathcal{P}(\mathsf{State})$ corresponds to a precise real number, and its abstraction $(\alpha\,\Psi) \in \widehat{\mathsf{State}}$ corresponds to the rounding of that number. There is no equivalent of γ because a rounded real number is still a real number, but in general we need γ to "coerce" a member of $\widehat{\mathsf{State}}$ back into a member of $\mathcal{P}(State)$. Then applying $\hat{\mathcal{S}}[\![P]\!]$, to take one step of program execution and then abstract, corresponds to processing (adding) the next number from the list and then immediately rounding the result.

8.5 Performing Multiple Steps Between Abstractions

An abstract interpretation computes the fixed point of the abstract step function $\hat{\mathcal{S}}[\![P]\!]$. One can write this fixed point as the limit of the sequence:

$$\begin{array}{lllll} \hat{\Psi}_1 & = & \hat{\mathcal{S}}[\![P]\!]\,\hat{\Psi}_0 & & \\ \hat{\Psi}_2 & = & \hat{\mathcal{S}}[\![P]\!]\,\hat{\Psi}_1 & = & (\hat{\mathcal{S}}[\![P]\!] \circ \hat{\mathcal{S}}[\![P]\!])\,\hat{\Psi}_0 \\ \hat{\Psi}_3 & = & \hat{\mathcal{S}}[\![P]\!]\,\hat{\Psi}_2 & = & (\hat{\mathcal{S}}[\![P]\!] \circ \hat{\mathcal{S}}[\![P]\!] \circ \hat{\mathcal{S}}[\![P]\!])\,\hat{\Psi}_0 \\ \vdots & & & & \end{array}$$

This sequence first adds in the objects (for instance, states or state sequences) reachable in one step from $\hat{\Psi}_0$, abstracts, adds in the objects reachable in the next step, abstracts, and so forth. By the definition of $\hat{\mathcal{S}}[\![P]\!]$,

$$\hat{\mathcal{S}}[\![P]\!] \circ \hat{\mathcal{S}}[\![P]\!] \circ \hat{\mathcal{S}}[\![P]\!]$$

is equivalent to

$$\alpha \circ \mathcal{S}[\![P]\!] \circ \gamma \circ \alpha \circ \mathcal{S}[\![P]\!] \circ \gamma \circ \alpha \circ \mathcal{S}[\![P]\!] \circ \gamma.$$

This illustrates the abstraction (with α) at every step. But in Section 1.1 we explained that it is more accurate to defer the abstraction for a few steps. Mathematically, this is easy to express: simply remove the occurrences of $\gamma \circ \alpha$ during the desired interval. Thus,

$$\alpha \circ \mathcal{S}[\![P]\!] \circ \mathcal{S}[\![P]\!] \circ \mathcal{S}[\![P]\!] \circ \gamma.$$

performs three steps before abstracting, and consequently may yield more accurate results than applying $\hat{\mathcal{S}}[\![P]\!]$ three times. (The formal justification of this is in [CC92d].) As we explained above and in Section 1.1, this increase in accuracy can be striking. This technique yielded better sign properties in our small example of Section 1.1, in which the three steps were the three assignments of the loop body; but much more importantly, *any* analysis of properties that might be *temporarily* lost during execution, such as data shape properties, stands to gain from this technique. This class of analyses is quite large.

Implementing this technique would seem to be a simple engineering issue: just remove the selected occurrences of $\gamma \circ \alpha$. This is an illusion, however. The problem is that the function $\hat{\mathcal{S}}[\![P]\!]$ is *specified* to be $\alpha \circ \mathcal{S}[\![P]\!] \circ \gamma$, but is never *implemented* that way. Indeed, it is not possible to manipulate the semantic objects (members of SemObj, perhaps sets of states or state sequences as described above) themselves because they are usually not computer-representable. For instance, consider the sign-analysis example. It is not possible to compute γ, yielding an (almost certainly) infinite set of states, apply $\mathcal{S}[\![P]\!]$ to find their successor states, and abstract the resulting infinite set of states. Instead, one always designs a monolithic algorithm to compute $\hat{\mathcal{S}}[\![P]\!]$, or at least a function above $\hat{\mathcal{S}}[\![P]\!]$ in its pointwise ordering, along with a soundness proof. Because this algorithm is cannot be separated into the three stages of γ, $\mathcal{S}[\![P]\!]$, and α, there is no general engineering solution to omit the computation of $\gamma \circ \alpha$ between two iterative applications of the algorithm. We give an example to illustrate this in the next section.

One might attempt to attack the problem from the different angle of beginning with a semantics that uses a coarser-grained step function, such as

$$\mathcal{S}^3[\![P]\!] = \mathcal{S}[\![P]\!] \circ \mathcal{S}[\![P]\!] \circ \mathcal{S}[\![P]\!].$$

Then the function $\hat{\mathcal{S}}^3[\![P]\!] = \alpha \circ \mathcal{S}^3[\![P]\!] \circ \gamma$ specifies an analysis that abstracts only after every third execution step. However, this different line of attack again encounters a barrier in practice. Although $\mathcal{S}^3[\![P]\!]$ is certainly a reasonable mathematical function, any algorithm for $\hat{\mathcal{S}}^3[\![P]\!]$ must in general be able to handle all possible combinations of three adjacent steps. For instance, consider just the two interesting adjacent steps in the example of Section 1.1:

$$\begin{array}{l}\mathtt{y} := \mathtt{x} - 3;\\ \mathtt{x} := \mathtt{y} + 5\end{array}$$

An algorithm that combines these two steps would have to recognize the special pattern occurring here that preserves the property that `x` is positive, and this pattern would have to be included explicitly in the algorithm. Again, there does not seem to be a general approach, or at least an approach that is combinatorially reasonable to even specify.

To understand this difficulty further, consider a program analysis based on a transition-system semantics. As we explained in Chapter 4 one typically defines the single-step transition relation $\longmapsto$ with meta-rules that specify how the individual pieces of program syntax induce transitions. For instance, the semantics of PURE included the following rule for let-binding transitions.

$$(\textbf{let } x = e \textbf{ in } t, \rho) \;\longmapsto\; (t, \sigma[x \mapsto \mathcal{E}[\![e]\!]\rho])$$

In the typical approach to program-analysis design, one would "bake" the abstraction into such a rule. The program analysis designer would hand-design an algorithm that "abstractly" performs this kind of transition. For instance, if $\widehat{\mathsf{SemObj}}$ is the set of tables of sign environments indexed by control point, as described above, then a straightforward algorithm to compute $\hat{\mathcal{S}}[\![P]\!]$ for some PURE program P will be hard-wired to propagate the sign property of expression e at control point ($\textbf{let } x = e \textbf{ in } t$) to variable x at control point t for each let-binding term in P. This makes intuitive sense—the algorithm is "abstractly interpreting" the let-binding steps. But of course the analysis designer should justify these intuitions by proving that the algorithm for $\hat{\mathcal{S}}[\![P]\!]$ actually implements the function

$$\alpha \circ \mathcal{S}[\![P]\!] \circ \gamma.$$

Hence, the algorithm never directly manipulates states or state sequences, but instead performs the function $\hat{\mathcal{S}}[\![P]\!]$ in one go, where α and γ are "baked into" the transition relation $\longmapsto$. Note that:

1. To apply an existing analysis to a different language, one must separately hand-design a new algorithm for the meta-rules of that language. This is an engineering disadvantage.

2. Because the abstraction is included in the analysis algorithm and cannot be separated as a single module, there is no way to perform multiple execution steps abstracting the result. This is a more serious disadvantage because, as we explained in Section 1.1, it can have devastating effects on the quality of the analysis.

The preceding discussion formalizes the intuition behind why both the small program in Section 1.1 and the `reverse` program at the beginning of this chapter are difficult to analyze accurately. Our solution in Section 1.1 was to change the program itself, rewriting the sequence of instructions in the loop body with a single parallel instruction. In that way, we achieved an effect similar to the $\mathcal{S}^3[\![P]\!]$ idea above. Although this "compilation" of the three instructions into a single instruction was at the time for expository purposes, we now have the semantic methodology of transfer relations as a general solution.

8.6 Multi-step Abstract Interpretation with Transfer Relations

In previous chapters, we demonstrated that our language of transfer relations is expressive enough to model advanced language features such as first-class functions and mutable data structures. We now show that it may be used as a "back end" for a generalized program-analysis methodology based on abstract interpretation in which multiple program steps may be assimilated between abstractions.

In Section 8.3 we explained that a common choice of concrete semantic object for program analysis is a state set (or property). As we described in Chapter 4, in semantic methodology of transfer relations, a state is a pair of a control point and a store.

$$\mathsf{State} = \mathsf{CtrlPoint} \times \mathsf{Store}$$

A set of states is thus isomorphic to a function from control point C to the set (or property) of stores occuring in states at C:

$$\Psi \in \mathsf{SemObj} = \mathcal{P}(\mathsf{State}) \simeq \mathsf{CtrlPoint} \to \mathcal{P}(\mathsf{Store})$$

Let $\mathsf{CtrlPoint}$ be the finite set of control points occuring in a particular program P. Given a binary relation R, let $[R] = \lambda X.\{y \mid x \in X \wedge xRy\}$. Then

$$\mathcal{S}[\![t]\!]\,\Psi\,C' = \bigcup_{C} [\Delta_{C,C'}]\,(\Psi\,C).$$

Intuitively, the set of stores at control point C' comes from the stores at all control points C that might precede C' by one step in an execution, or in other words by one link in a control-flow graph of P. But now we can express multiple steps with relation composition. For instance

$$\mathcal{S}^2[\![t]\!]\,\Psi\,C'' = (\mathcal{S}[\![t]\!] \circ \mathcal{S}[\![t]\!])\,\Psi\,C'' = \bigcup_{C,C'} [\Delta_{C,C'} \circ \Delta_{C',C''}]\,(\Psi\,C) = \bigcup_{C,C'} [\Delta_{C,C',C''}]\,(\Psi\,C).$$

In general, because the set of control points of a program is finite, we need only design a join semilattice $\widehat{\mathsf{Store}}$ of abstract store properties and an abstraction function $\alpha \in \mathcal{P}(\mathsf{Store}) \to \widehat{\mathsf{Store}}$, with induced concretization function γ.

$$\hat{\Psi} \in \widehat{\mathsf{State}} = \mathsf{CtrlPoint} \to \widehat{\mathsf{Store}}$$

If α is additive[1] then

$$\hat{\mathcal{S}}[\![t]\!]\,\hat{\Psi}\,C' = \bigvee_{C} \left(\alpha \circ [\Delta_{C,C'}] \circ \gamma\right)(\hat{\Psi}\,C).$$

But now we may perform any number of steps before abstracting. For instance,

$$\hat{\mathcal{S}}^2[\![t]\!]\,\hat{\Psi}\,C'' = \bigvee_{C,C'} \left(\alpha \circ [\Delta_{C,C',C''}] \circ \gamma\right)(\hat{\Psi}\,C).$$

Although the size of this join is $O(n^2)$, and in general $O(n^k)$ for k steps, a sensible analysis would only do this in cases such as straight-line code, where it is known beforehand that only one control path yields a non-$\perp$ transfer relation.

Thus, an analysis reduces to implementing $\alpha \circ [\Delta_\Gamma] \circ \gamma$ for any control path $\Gamma \in \mathsf{CtrlPoint}^+$. This is done with an algorithm S that describes how any transfer relation maps a pre abstract store property to a post abstract store property.

$$\mathsf{S} \in \mathsf{TR} \to \widehat{\mathsf{Store}} \to \widehat{\mathsf{Store}}$$

Then S is a function that, given a transfer relation Δ_Γ describing control path Γ, describes how a store property at the control point at the beginning of Γ propagates through Γ and yields a store property at the end of Γ. Conceptually, because S describes the exact net behavior of Γ, the abstraction step only occurs at the end of Γ, no matter how long Γ is.

The following picture describes the paradigm of multi-step abstract interpretation.

control point	store property	
C	$\hat{\Psi}\,C$	store property at C given by $\hat{\Psi}$
$\vdots$		$\vdots$
Γ		execution through control path C, Γ, C'
$\vdots$		$\vdots$
C'	$(\mathsf{S}\,\Delta_{C,\Gamma,C'})(\hat{\Psi}\,C)$	store property at C' after propagation through C, Γ, C' and abstraction at C'

Standard abstract interpretation corresponds to the case in which Γ is always the empty path, and so the propagation is through a single step from C to C'.

As we have described, once one designs the abstract store $\widehat{\mathsf{Store}}$, the heart of any program analysis defined with the standard methodology of abstract interpretation is the design of an

[1] Usually, α is additive, but otherwise the equality is a property implication and still yields a correct analysis.

algorithm to "abstractly" interpret each meta-rule of the transition relation $\longmapsto$ on $\widehat{\mathsf{Store}}$. In our methodology, the heart is the design of the S function, which abstractly interprets the transfer relations in our language TR over $\widehat{\mathsf{Store}}$. We want S to satisfy

$$(\alpha \circ [\Delta] \circ \gamma) \sqsubseteq (\mathsf{S}\,\Delta)$$

where $\sqsubseteq$ is pointwise set inclusion. Ideally, the $\sqsubseteq$ would be $=$, but an analysis may always safely weaken the properties. Because our methodology works on the universal intermediate representation of transfer relations, we may at least begin to describe how S should be defined, independent of the particular source language or analysis.

We assume that $\widehat{\mathsf{Store}}$ is a join semilattice with the false property as its bottom element, written as $\bot$. We also assume that there is a function in $\mathsf{Exp} \to \widehat{\mathsf{Store}} \to \widehat{\mathsf{Store}}$ that given an expression e and store property $\hat{\sigma}$ returns a store property (written $e?\hat{\sigma}$) that is satisfied by all stores that both satisfy $\hat{\sigma}$ and evaluate e to `true`. In other words,

$$(\sigma \in \hat{\sigma} \wedge e \vdash_{\sigma} \texttt{true}) \Rightarrow \sigma \in (e?\hat{\sigma}).$$

We also assume that there is a similar function for `false`:

$$(\sigma \in \hat{\sigma} \wedge e \vdash_{\sigma} \texttt{false}) \Rightarrow \sigma \in (e\text{¿}\hat{\sigma})$$

Note that the definitions

$$e?\hat{\sigma} = e\text{¿}\hat{\sigma} = \hat{\sigma}$$

trivially satisfy these properties, but in general it may be possible to do better, and so we provide the facility.

Now we may partially define S, independent of the particular analysis or choice of $\widehat{\mathsf{Store}}$.

$$\begin{aligned}
\mathsf{S}\,\Delta\,\bot &= \bot \\
\mathsf{S}\,\emptyset\,\hat{\sigma} &= \bot \\
\mathsf{S}\,\boxed{e?\ \Delta \,\big|\, \Delta'}\,\hat{\sigma} &= (\mathsf{S}\,\Delta\,(e?\hat{\sigma})) \vee (\mathsf{S}\,\Delta'\,(e\text{¿}\hat{\sigma}))
\end{aligned}$$

The only remaining case is for assignment relations. Therefore, we have the following "recipe" for the design of a general multi-step abstract interpretation with our methodology.

1. Design a join semilattice $\widehat{\mathsf{Store}}$ of store properties.

2. Define $e?\hat{\sigma}$ and $e\text{¿}\hat{\sigma}$ or use the degenerate definitions given above.

3. Define $(\mathsf{S}\,\delta\,\hat{\sigma})$ for any assignment relation $\delta \in \mathsf{ATR}$ and store property $\hat{\sigma} \in \widehat{\mathsf{Store}}$ such that $((\alpha \circ [\delta] \circ \gamma)\,\hat{\sigma}) \subseteq (\mathsf{S}\,\delta\,\hat{\sigma})$.

Then, as we described above, one may perform a classical abstract interpretation by using the single-step transfer relations defined by the semantics of the source language, or one may choose to compose these transfer relations for better precision over selected control paths. It is up to the analysis designer to pick which control paths are of interest, but we suggest a strategy of composing all paths in the control-flow graph of the source program in which only the first and last nodes have multiple incoming edges (candidates for looping points) and performing an abstract interpretation to compute the properties (via fixed-point iteration) for just those nodes. Such paths roughly correspond to so-called *extended basic blocks* [ASU86]. Note that such a strategy would automatically compose the sequence of three assignment statements that posed a problem in the example at the beginning of this chapter. All that remains to solve that example, for instance, is to adapt an existing store analysis such as [GH96].

8.7 Value Analysis

In this section, we isolate a subcase of our our methodology for *value analyses*. The sign analysis of Section 1.1 and Section 8.4 is a simple kind of value analysis. Recall that a store is a function from l-value to value. In a value analysis, a store property is defined in terms of a set $\widehat{\mathsf{Val}}$ of value properties (sets) as follows.

$$\begin{array}{rcrcl} \hat{\sigma} & \in & \widehat{\mathsf{Store}} & = & \widehat{\mathsf{Lval}} \to \widehat{\mathsf{Val}} \\ \hat{w} & \in & \widehat{\mathsf{Lval}} & = & \mathsf{Var} \cup (\widehat{\mathsf{Val}} \times \widehat{\mathsf{Val}}) \end{array}$$

A member of $\widehat{\mathsf{Lval}}$ specifies an l-value property (set) as follows

$$x \in x \qquad\qquad \frac{v \in \hat{v} \quad v' \in \hat{v}'}{(v.v') \in (\hat{v}.\hat{v}')}$$

The abstraction function

$$\alpha \in \mathcal{P}(\mathsf{Store}) \to \widehat{\mathsf{Store}}$$

is defined as

$$\alpha\, \Sigma\, \hat{w} = \bigwedge \{ \hat{v} \in \widehat{\mathsf{Val}} \mid (\sigma \in \Sigma \wedge w \in \hat{w}) \Rightarrow (\sigma\, w) \in \hat{v} \}$$

given a lattice $\widehat{\mathsf{Val}}$ of value properties.

In our example of sign analysis, $\widehat{\mathsf{Val}} = \mathsf{Sign}$ as given in Section 1.1, and we assumed that the only l-values were variables (in other words, $\widehat{\mathsf{Lval}} = \mathsf{Var}$).

The main algorithm that the analysis designer must provide for a particular value analysis is a function

$$\hat{\mathsf{P}} \quad \in \quad \mathsf{Primop} \to \widehat{\mathsf{Store}} \to \widehat{\mathsf{Val}}^* \to \widehat{\mathsf{Val}}$$

that “abstractly evaluates” primitive operations. It must satisfy the following condition

Condition 2 (Safety of $\hat{\mathsf{P}}$) *It must be the case that*

$$\left(\bigwedge_{1 \le i \le n} v_i \in \hat{v}_i \right) \wedge (\sigma \in \hat{\sigma}) \Rightarrow (p(v_1, \ldots, v_n) \hookrightarrow_\sigma v)$$

implies that

$$v \in \hat{\mathsf{P}}[\![p]\!]\, \sigma\, (\hat{v}_1, \ldots, \hat{v}_n).$$

Note that the store parameter of $\hat{\mathsf{P}}$ may be ignored for context-insensitive primitive operations, which will typically make up the vast majority of primitive operations in any application of our analysis methodology. For instance, the following is a partial definition of $\hat{\mathsf{P}}$ for the (context-independent) operation + for sign analysis (where we omit the unused store parameter):

$$\begin{aligned}
\hat{\mathsf{P}}[\![\mathsf{+}]\!]\,(\mathtt{pos}, \mathtt{int}) &= \mathtt{int}\\
\hat{\mathsf{P}}[\![\mathsf{+}]\!]\,(\mathtt{pos}, \mathtt{nonneg}) &= \mathtt{pos}\\
\hat{\mathsf{P}}[\![\mathsf{+}]\!]\,(\mathtt{pos}, \mathtt{nonpos}) &= \mathtt{int}\\
\hat{\mathsf{P}}[\![\mathsf{+}]\!]\,(\mathtt{pos}, \mathtt{pos}) &= \mathtt{pos}\\
\hat{\mathsf{P}}[\![\mathsf{+}]\!]\,(\mathtt{pos}, \mathtt{zero}) &= \mathtt{pos}\\
\hat{\mathsf{P}}[\![\mathsf{+}]\!]\,(\mathtt{pos}, \mathtt{neg}) &= \mathtt{int}\\
\hat{\mathsf{P}}[\![\mathsf{+}]\!]\,(\mathtt{pos}, \mathtt{none}) &= \mathtt{none}
\end{aligned}$$

We mechanically extend this notion of abstract evaluation to expressions and l-expressions as follows.

$$\begin{aligned}
\hat{\mathsf{E}} &\in \mathsf{Exp} \to \widehat{\mathsf{Store}} \to \widehat{\mathsf{Val}}\\
\hat{\mathsf{L}} &\in \mathsf{Lexp} \to \widehat{\mathsf{Store}} \to \widehat{\mathsf{Val}}
\end{aligned}$$

$$\begin{aligned}
\hat{\mathsf{E}}[\![x]\!]\,\hat{\sigma} &= \hat{\sigma}\,x\\
\hat{\mathsf{E}}[\![p(e_1, \ldots, e_n)]\!]\,\hat{\sigma} &= \hat{\mathsf{P}}[\![p]\!]\,\hat{\sigma}\,(\hat{\mathsf{E}}[\![e_1]\!]\,\hat{\sigma}, \ldots, \hat{\mathsf{E}}[\![e_n]\!]\,\hat{\sigma})
\end{aligned}$$

$$\begin{aligned}
\hat{\mathsf{L}}[\![x]\!]\,\hat{\sigma} &= x\\
\hat{\mathsf{L}}[\![e.e']\!]\,\hat{\sigma} &= (\hat{\mathsf{E}}[\![e]\!]\,\hat{\sigma}).(\hat{\mathsf{E}}[\![e']\!]\,\hat{\sigma})
\end{aligned}$$

Lemma 10 *For all $e \in \mathsf{Exp}$ and $v \in \mathsf{Val}$,*

$$(\sigma \in \hat{\sigma} \wedge e \vdash_\sigma v) \Rightarrow v \in \hat{\mathsf{E}}[\![e]\!]\,\hat{\sigma},$$

and for all $l \in \mathsf{Lexp}$ and $w \in \mathsf{Lval}$,

$$(\sigma \in \hat{\sigma} \wedge l \vdash_\sigma w) \Rightarrow w \in \hat{\mathsf{L}}[\![l]\!]\,\hat{\sigma}.$$

Proof: Straightforward induction based on the safety condition of $\hat{\mathsf{P}}$. □

The next step in our "recipe" is to provide the two boolean filter functions, which are defined as follows.

$$\begin{aligned}
e?\hat{\sigma} &= \begin{cases} \hat{\sigma} & \text{if } \mathtt{true} \in (\hat{\mathsf{E}}[\![e]\!]\,\hat{\sigma})\\ \bot & \text{otherwise} \end{cases}\\
e\text{¿}\hat{\sigma} &= \begin{cases} \hat{\sigma} & \text{if } \mathtt{false} \in (\hat{\mathsf{E}}[\![e]\!]\,\hat{\sigma})\\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

The final step is to provide the function

$$\mathsf{S} \in \mathsf{ATR} \to \widehat{\mathsf{Store}} \to \widehat{\mathsf{Store}}$$

that describes how, given an assignment relation δ, one store property evolves into another from the assignments in δ. To do this, we must assume that we have the following operations on $\widehat{\mathsf{Val}}$.

$$\begin{array}{rl} !\hat{v} & \text{test if } \hat{v} \in \widehat{\mathsf{Val}} \text{ is a singleton set} \\ \hat{v} \bowtie \hat{v}' & \text{test of nonempty intersection in } \widehat{\mathsf{Val}}\text{: } \hat{v} \cap \hat{v}' \neq \emptyset \end{array}$$

We extend the last two to $\widehat{\mathsf{Lval}}$ mechanically as follows.

$$\begin{array}{rl} !x & \text{(all variables are singletons)} \\ !(\hat{v}.\hat{v}') & \text{if } !\hat{v} \wedge !\hat{v}' \\ & \\ x \bowtie x & \text{(all variables intersect with themselves)} \\ \hat{v}_1.\hat{v}_1' \bowtie \hat{v}_2.\hat{v}_2' & \text{if } (\hat{v}_1 \bowtie \hat{v}_2) \wedge (\hat{v}_1' \bowtie \hat{v}_2') \end{array}$$

Now we can define the S algorithm.

$$\mathsf{S}\boxed{l_1, \ldots, l_n \mapsto e_1, \ldots, e_n}\,\hat{\sigma}\,\hat{w} \quad = \quad \begin{cases} \hat{\mathsf{E}}[\![e_i]\!]\,\hat{\sigma} & \text{if } (\hat{w} \bowtie \hat{w}') \wedge !\hat{w} \wedge !\hat{w}' \\ & \quad \text{where } \hat{w}' = \hat{\mathsf{L}}[\![l_i]\!]\,\hat{\sigma} \\ (\hat{\sigma}\,\hat{w}) \vee \bigvee\{\hat{\mathsf{E}}[\![e_i]\!]\,\hat{\sigma} \mid \hat{w} \bowtie \hat{\mathsf{L}}[\![l_i]\!]\,\hat{\sigma}\} & \text{otherwise} \end{cases}$$

Some important kinds of analyses are not value analyses. A good example is the shape analysis with which we began this chapter. We leave the adaption of such analyses to TR as future work.

Chapter 9

Analyzing Expressions

In Chapters 2 and 3, we gave a framework for designing a language of transfer relations parameterized by a set Primop of primitive operations, and we gave an algorithm to compose one transfer relation Δ with another Δ' to get a third transfer relation $\Delta'' = \Delta \oplus \Delta'$. In Chapter 4 we further described a semantic methodology in which

- Δ_Γ is a term representing the net behavior (modeled as a modification to a store) of any segment of execution through control path Γ (a string of control points),
- $\Delta_{\Gamma'}$ is a term representing the net behavior of a segment corresponding to control path Γ', and
- if Γ ends with the same control point with which Γ' begins, $\Delta_{\Gamma;\Gamma'} = \Delta_\Gamma \oplus \Delta_{\Gamma'}$ represents the net behavior of the first execution segment followed by the second execution segment. (Recall that a control path Γ_1, C ending with control point C may be combined with a control path C, Γ_2 beginning with C with the ; operation as $(\Gamma_1, C); (C, \Gamma_2) = \Gamma_1, C, \Gamma_2$.)

For program analysis, composing transfer relations thus forms the core of reasoning about adjacent pieces of code, or even pieces of code that are not *syntactically* adjacent, but that might follow each other in an execution. An example of the latter is a piece of code that leads to a function call, followed by a piece of code at the beginning of the function body. We formalized these concepts in Chapter 4, which presented a methodology of programming-language semantics based on transfer relations. This methodology provides a uniform way to compute the transfer relation for any finite control path in a program. We demonstrate this methodology in Chapters 5, 6, and 7 for imperative and functional programming languages.

In this chapter, we show how one can use this methodology to analyze how the values of an expression change during program execution. This will turn turn out to be relatively straightforward for fixed finite control sequences, but rather subtle for executions that are infinite, or at least potentially infinite.

9.1 Analyzing Finite Control Paths

Given a program in any language that has been defined using the semantic methodology of Chapter 4—and given any finite control path $\Gamma \in \mathsf{CtrlPoint}^+$, one can compute a transfer relation $\Delta_\Gamma \in \mathsf{TR}$ that gives a description of how a store at the start of Γ can change into a store at the end of Γ. If all of the primitive operations in the language are deterministic, as is the case for Mini-C, Pure, and Impure, and as will likely be the case for any reasonable language, then Δ_Γ will be an *exact* description, in that it *defines* the semantics of the control path Γ. In Chapter 3, we called this a *translation.* If the language includes nondeterministic primitive operations, then Δ_Γ will not necessarily be an exact description, but it will be a superset of the semantics of the control path Γ; in other words, all possible executions along control path Γ will be represented in Δ_Γ. In Chapter 3, we called this an *upper approximation.* In this chapter, we will assume that all primitive operations in the source language are deterministic.

Much of the field of static program analysis is centered around the common motivation of *analyzing the values of variables or expressions.* Usually this is done with a fixed-point computation, as in abstract interpretation. But in this chapter we present an alternate approach.

Suppose that one wants information about the value of $x \in \mathsf{Var}$ when execution reaches the end of control path Γ. In our methodology, one computes

$$\mathsf{E}\, x\, \Delta_\Gamma .$$

The result is an expression e such that in *any* possible execution fragment through control path Γ, e at the beginning of that execution fragment is semantically equivalent to x at the end of that execution fragment. If Primop is nondeterministic, then e is an upper approximation of x, in that it is guaranteed to evaluate before the execution fragment to any value to which x evaluates after the execution fragment. In general, x can be an arbitrary expression; it need not be a variable. In other words,

$$\mathsf{E}\, e'\, \Delta_\Gamma$$

is an expression e such that in any possible execution fragment through control path Γ, e at the beginning of that execution fragment is semantically equivalent to e' at the end of that execution fragment. These properties are not new; they are simply rephrasings of Theorem 2. But because this notion of expression analysis is a central part of this application of our methodology, we introduce a new term.

Definition 16 (Transfer of expressions) *If*

$$e \vdash_\sigma v \iff e' \vdash_{\sigma'} v$$

whenever execution from store $\sigma \in \mathsf{Store}$ *through control path* $\Gamma \in \mathsf{CtrlPoint}^+$ *results in store* $\sigma' \in \mathsf{Store}$*, then we say that* Γ transfers e to e'*.*

The following theorems and corollary are the keystone of this chapter.

Theorem 5 *For any programming language all of whose primitive operations are deterministic, if* Δ_Γ *is the transfer relation for control path* Γ *and* $(\mathsf{E}\, e'\, \Delta_\Gamma) = e$*, then* Γ *transfers* e *to* e'*.*

Proof: Rephrasing of Theorem 2. □

Theorem 6 *If* Γ *transfers* e *to* e' *and* Γ' *transfers* e' *to* e'' *then* $\Gamma; \Gamma'$ *transfers* e *to* e''*.*

Proof: Straightforward from the definition of expression transfer. □

Corollary 2 *For any programming language all of whose primitive operations are deterministic, if* Δ_Γ *is the transfer relation for control path* Γ *and* $\Delta_{\Gamma'}$ *is the transfer relation for control path* Γ'*, and if* $(\mathsf{E}\,(\mathsf{E}\, e'\, \Delta_{\Gamma'})\, \Delta_\Gamma) = e$*, then* $\Gamma; \Gamma'$ *translates* e *to* e'*.*

9.2 Analyzing Adjacent Loop Iterations via Exponentiation

Consider the Mini-C program

$$\textbf{while}\ e\ \textbf{do}\ s$$

that repeats the execution of the code s until e becomes true. Suppose s is simply a piece of straight-line code, and Γ is the control path that tests if e is true and then performs s. As described in Chapter 5, one can automatically compute a transfer relation Δ_Γ for Γ that represents the net behavior of the test of e and execution of s. So,

- Δ_Γ represents the net behavior of any single iteration,
- $\Delta_\Gamma \oplus \Delta_\Gamma$ represents the net behavior of control path $\Gamma; \Gamma$, which is the control path of any two adjacent iterations,
- $\Delta_\Gamma \oplus \Delta_\Gamma \oplus \Delta_\Gamma$ represents the net behavior of control path $\Gamma; \Gamma; \Gamma$, which is the control path of any three adjacent iterations,
- etc.

We adopt the notation Δ^n to mean

$$\overbrace{\Delta \oplus \cdots \oplus \Delta}^{n\ \text{times}}$$

and the notation Γ^n to mean

$$\overbrace{\Gamma; \ldots; \Gamma}^{n\ \text{times}}.$$

Then $\Delta_{\Gamma^n} = (\Delta_\Gamma)^n$.

Suppose that one wants to analyze how the data in a store σ at the beginning of the loop body gets used and updated over a period of three iterations of the loop—in other words, during any segment of execution along the control path Γ^3. Then one simply computes $(\Delta_\Gamma)^3$, which gives a description of the store three iterations later[1] in terms of σ.

It is worth pointing out that this is a fundamentally different—and fundamentally advantageous—approach from those of standard program analyses. Standard approaches cannot differentiate between an infinite number of loop iterations, but the above approach can. For instance, suppose the loop is

$$\textbf{while}\ \texttt{x <> nil}\ \textbf{do}\ \texttt{x} := \texttt{x.tl}$$

that traverses `x` down a list to its end. The semantic methodology in Chapter 4 would compute

$$\Delta_\Gamma = \boxed{\texttt{x <> nil?}\ \boxed{\texttt{x} \mapsto \texttt{x.tl}}}$$

to describe one iteration of this loop. One could then compute, for instance,

$$(\Delta_\Gamma)^2 = \boxed{\texttt{x <> nil?}\ \boxed{\texttt{x.tl <> nil?}\ \boxed{\texttt{x} \mapsto \texttt{x.tl.tl}}}}$$

to describe two adjacent iterations, or

$$(\Delta_\Gamma)^3 = \boxed{\texttt{x <> nil?}\ \boxed{\texttt{x.tl <> nil?}\ \boxed{\texttt{x.tl.tl <> nil?}\ \boxed{\texttt{x} \mapsto \texttt{x.tl.tl.tl}}}}}$$

to describe three adjacent iterations.[2] The last transfer relation directly provides the following information: During *any* three adjacent iterations of the loop, the third component of the list to which `x` is bound before those iterations is bound to `x` after those iterations. This is computed and formalized by the E algorithm; one computes:

$$\begin{aligned}
\mathsf{E}\,\texttt{x}\,\Delta_\Gamma &= \texttt{x.tl}\\
\mathsf{E}\,\texttt{x}\,(\Delta_\Gamma)^2 &= \texttt{x.tl.tl}\\
\mathsf{E}\,\texttt{x}\,(\Delta_\Gamma)^3 &= \texttt{x.tl.tl.tl}
\end{aligned}$$

Therefore:

- Whenever the loop goes through *any* one iteration, the value of `x.tl` (guaranteed by Lemma 2 to be unique because all primitive operations are deterministic) at the beginning of the iteration is equal to the value of `x` at the end of the iteration. In other words, Γ transfers `x.tl` to `x`.

- Whenever the loop goes through *any* two adjacent iterations, the unique value of `x.tl.tl` at the beginning of the iteration is equal to the value of `x` at the end of the iteration. In other words, Γ^2 (which is $\Gamma;\Gamma$) transfers `x.tl.tl` to `x`.

[1] In general, one might want to examine all of the transfer relations that accumulated during a left-associative calculation of $(\Delta_\Gamma)^3$—in other words, the transfer relations that correspond to each prefix of Γ^3. These intermediate transfer relations give descriptions of the store with respect to σ at all of the intermediate points of execution during a sequence of three iterations, as described in Chapter 4.

[2] These computations assume trivial translations C and P that merely reconstruct their terms.

- Whenever the loop goes through *any* three adjacent iterations, the value of `x.tl.tl.tl` (again, guaranteed to be unique) at the beginning of the iteration is equal to the value of `x` at the end of the iteration. In other words, Γ^3 (which is $\Gamma; \Gamma; \Gamma$) transfers `x.tl.tl.tl` to `x`.

In this sense, our methodology provides a way of distinguishing between a potentially unbounded number of occurrences of the same control path. Even if the length of the list to which `x` is bound at the entry of the while loop is unknown, and thus unbounded, the above transfer relation provides precise information about how the binding of `x` at iteration k relates to the binding of `x` at iteration $k + 3$, and this information is valid for *any* k. Most approaches to program analysis that are based on fixed-point calculation would ultimately have to approximate the data structure to which `x` is bound at the entry of the while loop, and therefore inherently cannot produce precise information for an unbounded number of iterations.

9.3 The Interaction Between Effects and Exponentiation

Above, we gave some intuition about how to analyze the value of `x` in the loop

$$\textbf{while}\ \texttt{x <> nil}\ \textbf{do}\ \texttt{x} := \texttt{x.tl}.$$

First, one computes the transfer relation

$$\Delta = \boxed{\texttt{x <> nil}?\ \boxed{\texttt{x} \mapsto \texttt{x.tl}}}$$

describing one iteration of the loop. Then, one can compute

$$\mathsf{E}\,\texttt{x}\,\Delta = \texttt{x.tl}$$

to automatically determine that one iteration of the loop transfers `x.tl` to `x`. Indeed, one can go further by computing

$$\Delta^3 = \Delta \oplus \Delta \oplus \Delta = \boxed{\texttt{x <> nil}?\ \boxed{\texttt{x.tl <> nil}?\ \boxed{\texttt{x.tl.tl <> nil}?\ \boxed{\texttt{x} \mapsto \texttt{x.tl.tl.tl}}}}}$$

describing three adjacent loop iterations, and then

$$\mathsf{E}\,\texttt{x}\,\Delta^3 = \texttt{x.tl.tl.tl}.$$

to yield the expected result that, of course, if `x` traverses one element down the list in one iteration, then it must traverse three elements down the list in three iterations. Or must it? In this program, it is indeed the case, but in general a result such as

$$\mathsf{E}\,\texttt{x}\,\Delta = \texttt{x.tl}$$

can be deceptive.

To see why, consider the following Mini-C program.

```
while x <> nil do
{
  y := x;
  x := x.tl;
  y.tl := x.tl
}
```

One would think that this program is designed to modify the list bound to x by splitting it into two lists—a list of the odd elements in order and a list of the even elements in order. The following transfer relation describes one iteration of the loop.

$$\Delta = \boxed{\texttt{x <> nil?}\ \boxed{\mathtt{y}, \mathtt{x}, \mathtt{x.tl} \mapsto \mathtt{x}, \mathtt{x.tl}, \mathtt{x.tl.tl}}}$$

Now, suppose that one wants to analyze the value of x in this loop. As for the previous program, one could compute

$$\mathsf{E}\,\mathtt{x}\,\Delta = \mathtt{x.tl}$$

to automatically determine that one iteration of the loop transfers x.tl to x; in other words, x progresses to its next element in one iteration. So far, this looks just like the previous program. Going further, one could examine the value of x after two iterations:

$$\mathsf{E}\,\mathtt{x}\,\Delta^2 = \mathtt{x.tl.tl}$$

This may not seem very surprising. After all, if x moves down one element of the list in one iteration, then it seems reasonable that it moves down two elements in two iterations. However, the pattern is broken with the next iteration:

$$\begin{aligned}\mathsf{E}\,\mathtt{x}\,\Delta^3 = \mathtt{if}(&\mathtt{x.tl.tl = x},\\ &\mathtt{x.tl.tl},\\ &\mathtt{x.tl.tl.tl})\end{aligned}$$

In other words, suppose that the list is a two-element circular list with elements v_1 and v_2, and x is bound to v_1. Then:

- After one iteration, x is bound to v_2 (i.e., element $\{2, 4, 6, \ldots\}$ of the list).
- After two iterations, x is bound to v_1 (i.e., element $\{1, 3, 5, \ldots\}$ of the list).
- After three iterations, x is still bound to v_1.

Otherwise, no matter what other kind of circularity or aliasing may be present in the list, x progresses to the fourth element in its linked structure after three iterations.

Note that the those two possible behaviors are truly distinct. To illustrate, consider how each of the two cases appear in a store graph. The first case is

$$\begin{tikzcd} \bullet \arrow[r, "\texttt{x}"] & v_1 \arrow[r, shift left, "\texttt{tl}"] & v_2 \arrow[l, shift left, "\texttt{tl}"] \end{tikzcd}$$

where v_1 may or may not be equal to v_2. The second case is

$$\begin{tikzcd} \bullet \arrow[r, "\texttt{x}"] & v_1 \arrow[r, "\texttt{tl}"] & v_2 \arrow[r, "\texttt{tl}"] & v_3 \arrow[r, "\texttt{tl}"] & v_4 \end{tikzcd}$$

where we impose only that v_1 and v_3 are nonequal (and thus v_1 and v_2 must be nonequal). After three iterations, `x` points to v_1 in the first case and v_4 in the second case. But there is no *single* non-conditional expression that works for both cases. The expression

$$\texttt{x.tl.tl}$$

works for the first case, but not the second; the expression

$$\texttt{x.tl.tl.tl}$$

works for the second case, but not the first. If v_4 is equal to v_1, then the expression `x` would work for both cases, but v_4 may not be equal to v_1. The E algorithm automatically distinguishes the cases that need to be distinguished and builds a conditional expression that covers all cases.

9.4 Blowup of Conditional Expressions

These conditional expressions can become large. Let

$$\texttt{x.tl}^n = \texttt{x.}\overbrace{\texttt{tl.}\ldots\texttt{.tl}}^{n \text{ times}}$$

After four iterations of the loop in the previous section, we have

$$\begin{aligned} \mathsf{E}\,\mathtt{x}\,\Delta^4 = {} & \texttt{if}(e \texttt{ = x.tl}, \\ & e, \\ & \texttt{if}(e \texttt{ = x}, \texttt{x.tl}^2, e\texttt{.tl})) \end{aligned}$$

where

$$e = \texttt{if}(\texttt{x.tl}^2 \texttt{ = x}, \texttt{x.tl}^2, \texttt{x.tl}^3).$$

Unlike the expression ($\mathsf{E}\,\mathtt{x}\,\Delta^3$) for three iterations, this expression is rather complex for human understanding (although still much easier than hand-generating all initial aliasing conditions that might be relevant and hand-executing four iterations of the loop under all such cases). Some study uncovers the following interpretation for the value of `x` after four iterations:

- If the second element points to the first (special case: the first element points to itself), then the value is `x`.

- Otherwise, if the third element points to the second (special case: the second element points to itself), then the value is $\texttt{x.tl}$.
- Otherwise, if the third element points to the first, then the value is $\texttt{x.tl}^2$.
- Otherwise, the value is $\texttt{x.tl}^4$.

Again, these are all distinct behaviors, and it is probable that the above itemized list is the shortest description of the value of $\texttt{x}$ after four iterations. But clearly the expression computed as $(\mathsf{E}\,\texttt{x}\,\Delta^4)$ is bigger than this itemized list. A better symbolic evaluation of $\texttt{if}$ could reduce its size. For instance note that as long as all primitive operations are deterministic, the following two expressions are semantically equivalent:

$$\texttt{if}(e_1 \texttt{=} e_2, e_1, e_3) \equiv \texttt{if}(e_1 \texttt{=} e_2, e_2, e_3)$$

Therefore, $\widetilde{\texttt{if}}$ can substitute one for the other, and for instance choose the small $\texttt{x.tl}$ over the large e above, yielding instead:

$$\begin{aligned}\mathsf{E}\,\texttt{x}\,\Delta^4 = \texttt{if}(&e \texttt{=} \texttt{x.tl},\\ &\texttt{x.tl},\\ &\texttt{if}(e \texttt{=} \texttt{x}, \texttt{x.tl}^2, e\texttt{.tl}))\end{aligned}$$

This is better, but not by much. The key is to distribute the conditional expression e nested in the condition position of $(\mathsf{E}\,\texttt{x}\,\Delta^4)$ over the two branches of the latter. For this, we have the following rule of semantic equivalence of expressions, where eE denotes any expression that can be derived from e by optionally replacing occurrences of any subexpression $e_1 \in E$ in e by some other expression $e_2 \in E$:

$$\frac{\begin{array}{c} e = \texttt{if}(e_1, e_2, e_3)\\ e_5' = e_5\{e, e_2, e_4\} \quad e_6' = e_6\{e, e_2\}\\ e_5'' = e_5\{e, e_3, e_4\} \quad e_6'' = e_6\{e, e_3\}\end{array}}{\texttt{if}(e \texttt{=} e_4, e_5, e_6) \equiv \texttt{if}(e_1, \texttt{if}(e_2 \texttt{=} e_4, e_5', e_6'), \texttt{if}(e_3 \texttt{=} e_4, e_5'', e_6''))} \tag{9.1}$$

As written, this rule is nondeterministic because there are in general many choices for an expression eE. But one obvious strategy is simply to pick the smallest expression. This is easy to implement. For instance to compute e_5', pick the smallest of $\{e, e_2, e_4\}$ and substitute all occurrences in e_5 of the two larger expressions by this small expression.

Applying this rule to $(\mathsf{E}\,\texttt{x}\,\Delta^4)$ yields:

$$\begin{aligned}\mathsf{E}\,\texttt{x}\,\Delta^4 = \texttt{if}(&\texttt{x.tl}^2 \texttt{=} \texttt{x},\\ &\texttt{if}(\texttt{x.tl}^2 \texttt{=} \texttt{x.tl}, \texttt{x.tl}, \texttt{x.tl}^2),\\ &\texttt{if}(\texttt{x.tl}^3 \texttt{=} \texttt{x.tl},\\ &\quad \texttt{x.tl},\\ &\quad \texttt{if}(\texttt{x.tl}^3 \texttt{=} \texttt{x}, \texttt{x.tl}^2, \texttt{x.tl}^4)))\end{aligned}$$

This is close to optimal, but it is possible to simplify the first arm even further using the following generalization of the first semantic equivalence above:

$$\frac{e_3' = e_3\{e_1, e_2\}}{\texttt{if}(e_1 \texttt{ = } e_2, e_3, e_4) \equiv \texttt{if}(e_1 \texttt{ = } e_2, e_3', e_4)} \tag{9.2}$$

Applying this rule to the above expression yields:

$$\begin{aligned}\mathsf{E}\,\mathtt{x}\,\Delta^4 = \texttt{if}(&\texttt{x.tl}^2 \texttt{ = x},\\ &\texttt{if}(\texttt{x = x.tl}, \texttt{x.tl}, \texttt{x}),\\ &\texttt{if}(\texttt{x.tl}^3 \texttt{ = x.tl},\\ &\quad \texttt{x.tl},\\ &\quad \texttt{if}(\texttt{x.tl}^3 \texttt{ = x}, \texttt{x.tl}^2, \texttt{x.tl}^4)))\end{aligned}$$

Applying the rule again to the conditional in the first arm yields:

$$\begin{aligned}\mathsf{E}\,\mathtt{x}\,\Delta^4 = \texttt{if}(&\texttt{x.tl}^2 \texttt{ = x},\\ &\texttt{if}(\texttt{x = x.tl}, \texttt{x}, \texttt{x}),\\ &\texttt{if}(\texttt{x.tl}^3 \texttt{ = x.tl},\\ &\quad \texttt{x.tl},\\ &\quad \texttt{if}(\texttt{x.tl}^3 \texttt{ = x}, \texttt{x.tl}^2, \texttt{x.tl}^4)))\end{aligned}$$

Finally, we apply the equivalence that

$$\texttt{if}(e_1 \texttt{ = } e_2, e, e) \equiv e \tag{9.3}$$

to yield:

$$\begin{aligned}\mathsf{E}\,\mathtt{x}\,\Delta^4 = \texttt{if}(&\texttt{x.tl}^2 \texttt{ = x},\\ &\texttt{x},\\ &\texttt{if}(\texttt{x.tl}^3 \texttt{ = x.tl},\\ &\quad \texttt{x.tl},\\ &\quad \texttt{if}(\texttt{x.tl}^3 \texttt{ = x}, \texttt{x.tl}^2, \texttt{x.tl}^4)))\end{aligned}$$

This last expression directly corresponds to the bullet list above, and no more simplifications seem possible; the behavior of four iterations of the loop on the binding of `x` is inherently this complex.

Rule 9.3 is subtle; it works only because the primitive operation `=` returns either `true` or `false` on any pair of values, even `undef`.

Rule 9.1 of semantic equivalence may seem ad hoc, but it is actually more widely applicable than it may seem at first. Let us look again at the computation of $(\mathsf{E}\,\mathtt{x}\,\Delta^4)$, but this time with the observation that $(\mathsf{E}\,\mathtt{x}\,\Delta^3)$ appears in some of its subexpressions.

$$\begin{aligned}\mathsf{E}\,\mathtt{x}\,\Delta^4 = \texttt{if}(&(\mathsf{E}\,\mathtt{x}\,\Delta^3) \texttt{ = x.tl},\\ &(\mathsf{E}\,\mathtt{x}\,\Delta^3),\\ &\texttt{if}((\mathsf{E}\,\mathtt{x}\,\Delta^3) \texttt{ = x}, \texttt{x.tl}^2, (\mathsf{E}\,\mathtt{x}\,\Delta^3)\texttt{.tl}))\end{aligned}$$

Now note that the nested conditional in Rule 9.1 occurs exactly where $(\mathsf{E}\,\mathtt{x}\,\Delta^3)$ appears above. This is not an accident; often, the behavior of a piece of code (e.g., one iteration of a loop) on an expression (e.g., a variable) will function in more than one possible way depending on properties (e.g., aliasing) of the result of the preceding piece of code (e.g., the previous loop iteration) on that expression. Hence, it is often the case that $(\mathsf{E}\,e\,\Delta)$ will appear in the proper position (i.e., as e) in an application of Rule 9.1 to $(\mathsf{E}\,e\,(\Delta \oplus \Delta'))$. The rule thus serves to incrementally keep the expressions as flat as possible.

9.5 Computing Closed Forms of Loops

The previous sections showed how to automatically compute that a single iteration of the loop

```
while x <> nil do x := x.tl
```

and a single iteration of the loop

```
while x <> nil do
{
  y := x;
  x := x.tl;
  y.tl := x.tl
}
```

both transfer `x.tl` to `x`.

It is not difficult to see that for any n, n iterations of the first loop transfer $\mathtt{x.tl}^n$ to `x`, but we did not give an algorithm to compute this closed-form solution. However, we demonstrated that three iterations of the second loop do *not* transfer $\mathtt{x.tl}^3$ to `x`, and therefore it is not the case that for any n, n iterations of the second loop transfer $\mathtt{x.tl}^n$ to `x`. This section addresses the question of when such exponentiations are valid, and how to automatically compute a closed-form representation for those exponentiations. The results that we will achieve are much more general than the simple traversal of a linear data structure.

9.5.1 An example

We begin at an intuitive level by examining why the closed-form exponentiation works for the first loop, but not for the second. Consider two adjacent iterations of the first loop. We know that iteration 1 transfers `x.tl` to `x` and iteration 2 transfers `x.tl` to `x`. But to link iteration 1 with iteration 2, the "output expression" of iteration 1 should be the same as the "input expression" of iteration 2. So what we really need is to compute

$$\mathsf{E}\,(\mathtt{x.tl})\,\Delta = \mathtt{x.tl.tl}$$

which reports that any one iteration (where Δ is the transfer relation for a single iteration) transfers `x.tl.tl` to `x.tl`. Now we know that iteration 1 transfers `x.tl.tl` to `x.tl`, which is then transferred by iteration 2 to `x`. This is an application of Corollary 2.

Thus, we have simply verified that two iterations transfer `x.tl.tl` to `x`, a fact that is expressed more directly by

$$\mathsf{E}\,\mathtt{x}\,\Delta^2 = \mathtt{x.tl.tl}.$$

But deriving that result in the two steps of Corollary 2 instead of computing it directly suggests an approach for deriving a closed-form solution for n steps. Note that in the equation

$$\mathsf{E}\,(\mathtt{x.tl})\,\Delta = \mathtt{x.tl.tl}$$

the expression on the right is a dereference of `x.tl` by `tl`, the expression on the left is a dereference of `x` by `tl`, and we already have that one iteration transfers `x.tl` to `x`. Suppose an oracle magically provides the statement that for all expressions e and e', if one iteration transfers e to e', then it must be the case that it also transfers e`.tl` to e'`.tl`. In other words,

$$\mathsf{E}\,e'\,\Delta = e \quad\Rightarrow\quad \mathsf{E}\,e'.\mathtt{tl}\,\Delta = e.\mathtt{tl}.$$

Then by induction, we have

$$\begin{array}{rcll}
\mathsf{E}\,\mathtt{x}\,\Delta & = & \mathtt{x.tl} & \text{base case}\\
\mathsf{E}\,\mathtt{x.tl}\,\Delta & = & \mathtt{x.tl}^2 & \text{application of oracle to above}\\
\mathsf{E}\,\mathtt{x.tl}^2\,\Delta & = & \mathtt{x.tl}^3 & \text{application of oracle to above}\\
\mathsf{E}\,\mathtt{x.tl}^3\,\Delta & = & \mathtt{x.tl}^4 & \text{application of oracle to above}\\
\vdots & & &
\end{array}$$

And then by another induction, we have

$$\begin{array}{rcll}
\mathsf{E}\,\mathtt{x}\,\Delta & = & \mathtt{x.tl} & \text{base case}\\
\mathsf{E}\,\mathtt{x}\,\Delta^2 & = & \mathtt{x.tl}^2 & \text{Corollary 2 with above line and line 2 of previous result}\\
\mathsf{E}\,\mathtt{x}\,\Delta^3 & = & \mathtt{x.tl}^3 & \text{Corollary 2 with above line and line 3 of previous result}\\
\mathsf{E}\,\mathtt{x}\,\Delta^4 & = & \mathtt{x.tl}^4 & \text{Corollary 2 with above line and line 4 of previous result}\\
\vdots & & &
\end{array}$$

The key to this approach is the oracle that provides the statement that

$$\mathsf{E}\,e'\,\Delta = e \quad\Rightarrow\quad \mathsf{E}\,(e'.\mathtt{tl})\,\Delta = e.\mathtt{tl}.$$

Now it becomes clear why the closed-form exponentiation works for the first program, but not for the second. The intuition of this statement is that “the `tl` fields of all data structures are preserved by a single iteration”. This is clearly true of the first program, which does not perform any assignments to `tl` fields. But the second program includes the statement

$$\mathtt{y.tl} := \mathtt{x.tl}$$

which potentially alters the tl field of some value in the store. And indeed, the oracle's statement is false when Δ is the transfer function of one iteration of the second program. At first, it seems tricky to find an expression e' that makes the statement fail. Neither x nor y serves the purpose, but x.tl does:

$$\mathsf{E}\,(\texttt{x.tl})\,\Delta = \texttt{x.tl.tl}$$

but

$$\begin{aligned}\mathsf{E}\,(\texttt{x.tl.tl})\,\Delta = \texttt{if}(&\texttt{x.tl.tl = x}, \\ &\texttt{x.tl.tl}, \\ &\texttt{x.tl.tl.tl})\end{aligned}\quad.$$

Fortunately, however, there is a general technique for testing if the above oracle statement holds. The insight is that an expression that cannot possibly be altered by the program can act as a "probe" into any point in the store. So one needs merely to choose e to be a variable x that does not appear in the program. It will always be the case that $(\mathsf{E}\,x\,\Delta) = x$, and if the oracle statement fails for any e' then it will fail for x. Furthermore, if the oracle statement passes for $e = e' = x$ then it will pass for *all* expressions e and e'. For instance, for both of our programs,

$$\mathsf{E}\,\texttt{z}\,\Delta = \texttt{z},$$

but while

$$\mathsf{E}\,(\texttt{z.tl})\,\Delta = \texttt{z.tl}$$

for the first program, thus implying that the oracle statement holds and thus the closed-form exponentiation is valid,

$$\begin{aligned}\mathsf{E}\,(\texttt{z.tl})\,\Delta = \texttt{if}(&\texttt{z = x}, \\ &\texttt{x.tl.tl}, \\ &\texttt{z.tl})\end{aligned}\quad.$$

for the second program, thus demonstrating that the oracle statement fails and thus the closed-form exponentiation is not valid.

The above is merely an example of exponentiating a tl dereference. Now we generalize these results to a much larger class of exponentiations.

9.5.2 Expression constructors

We begin with the observation that $\texttt{x.tl}^n$ is the result of n repeated applications of the function

$$\lambda e.\,(e.\texttt{tl})$$

to the expression x. Exponentiating a dereference is thus a special case of exponentiating a function of type

$$\mathsf{Exp} \rightarrow \mathsf{Exp}.$$

In this section, we present a foundation for these kinds of functions and how, given a loop, to automatically find such functions that can be exponentiated in the loop.

Definition 17 *The set* ExpCon_k *of* expression constructors of arity k *is defined as follows.*

$$\mathcal{E} \quad \in \quad \mathsf{ExpCon}_k \quad ::= \quad x \mid p(\mathcal{E}_1, \ldots, \mathcal{E}_n) \mid \textcircled{1} \mid \ldots \mid \textcircled{k}$$

ExpCon_k *is isomorphic to* $\mathsf{Exp}^k \to \mathsf{Exp}$, *and these may be used interchangeably.*

Intuitively, a k-ary expression constructor is an expression in which "holes" may appear, each hole labeled with a number from 1 to k. A hole may appear multiple times or not at all. When a k-ary expression constructor is applied to k expressions $(e_1, \ldots e_k)$, then each occurrence of hole $\textcircled{i}$ is "filled" with e_i. Note that ExpCon_0, the set of nullary expression constructors, is just the set Exp of expressions. Also note that $\mathsf{ExpCon}_k \supset \mathsf{Exp}$ for any k because an expression constructor need not contain any holes.

The definition of k-ary expression constructors as $\mathsf{Exp}^k \to \mathsf{Exp}$ functions is as follows:

$$\begin{array}{rcl} x\,\vec{e} & = & x \\ p(\mathcal{E}_1, \ldots, \mathcal{E}_n)\,\vec{e} & = & p(\mathcal{E}_1\,\vec{e}, \ldots, \mathcal{E}_n\,\vec{e}) \\ \textcircled{i}\,(e_1, \ldots, e_k) & = & e_i \end{array}$$

Unary expression constructors are especially important because they are the only ones that can be exponentiated, as they are the only ones with matching domain and codomain. Because they are distinguished, we simply call them *expression constructors* and use slightly specialized notation for them:

$$\mathcal{E} \quad \in \quad \mathsf{ExpCon} \quad ::= \quad x \mid p(\mathcal{E}_1, \ldots, \mathcal{E}_n) \mid \bigcirc \qquad \text{unary expression constructors}$$

We also specialize the definition above for the case of expression constructors as $\mathsf{Exp} \to \mathsf{Exp}$ functions:

$$\begin{array}{rcl} x\,e & = & x \\ p(\mathcal{E}_1, \ldots, \mathcal{E}_n)\,e & = & p(\mathcal{E}_1\,e, \ldots, \mathcal{E}_n\,e) \\ \bigcirc\,e & = & e \end{array}$$

In the previous section, we considered loops in which one iteration transfers `x.tl` to `x`. We started by calculating

$$\mathsf{E}\,\mathsf{x}\,\Delta = \texttt{x.tl}.$$

and then computing a test to determine if the "`.tl`" part could be exponentiated. The discussion in the previous section generalizes elegantly via the following three theorems.

Theorem 7 (Abstraction of expression transfer) *Let* Δ_Γ *be the transfer relation for control path* Γ, $\mathcal{E}$ *and* $\mathcal{E}'$ *be* k*-ary expression constructors, and* $x_1, \ldots, x_k$ *be variables that do not appear in either the syntax of* Δ_Γ, $\mathcal{E}$, *or* $\mathcal{E}'$. *If*

- Γ *transfers* $\mathcal{E}(x_1, \ldots, x_k)$ *to* $\mathcal{E}'(x_1, \ldots, x_k)$, *and*
- Γ *transfers* e_i *to* e_i *for all* $i \in \{1, \ldots, k\}$

then Γ transfers $\mathcal{E}(e_1,\ldots,e_k)$ to $\mathcal{E}'(e_1,\ldots,e_k)$.

Proof: We need to show that whenever $\sigma\,\Delta_\Gamma\,\sigma'$,

$$(\mathcal{E}(e_1,\ldots,e_k)) \vdash_\sigma v \iff (\mathcal{E}'(e_1,\ldots,e_k)) \vdash_{\sigma'} v.$$

Choose any values $v_1,\ldots,v_k$. Because Γ transfers e_i to e_i for all $i \in \{1,\ldots,k\}$, we have that

$$\bigwedge_{i=1}^{k} e_i \vdash_\sigma v_i \iff \bigwedge_{i=1}^{k} e_i \vdash_{\sigma'} v_i.$$

Now let

$$\begin{array}{rcl} \sigma'' & = & \sigma[x_1 \mapsto v_1]\ldots[x_k \mapsto v_k] \\ \sigma''' & = & \sigma'[x_1 \mapsto v_1]\ldots[x_k \mapsto v_k] \end{array}$$

Because none of $x_1,\ldots,x_k$ appears in the syntax of Δ_Γ, we have that

$$\sigma''\,\Delta_\Gamma\,\sigma'''$$

and hence, because Γ transfers $\mathcal{E}(x_1,\ldots,x_k)$ to $\mathcal{E}'(x_1,\ldots,x_k)$, that

$$(\mathcal{E}(x_1,\ldots,x_k)) \vdash_{\sigma''} v \iff (\mathcal{E}'(x_1,\ldots,x_k)) \vdash_{\sigma'''} v.$$

Combining the above, we have that

$$\left(\bigwedge_{i=1}^{k} e_i \vdash_\sigma v_i\right) \wedge (\mathcal{E}(x_1,\ldots,x_k)) \vdash_{\sigma''} v \iff \left(\bigwedge_{i=1}^{k} e_i \vdash_{\sigma'} v_i\right) \wedge (\mathcal{E}'(x_1,\ldots,x_k)) \vdash_{\sigma'''} v.$$

But because none of $x_1,\ldots,x_k$ appears in $\mathcal{E}$ we have that for any σ,

$$\exists v_1,\ldots,v_k.\ \left[\left(\bigwedge_{i=1}^{k} e_i \vdash_\sigma v_i\right) \wedge (\mathcal{E}(x_1,\ldots,x_k)) \vdash_{\sigma[x_1\mapsto v_1]\ldots[x_k\mapsto v_k]} v\right] \iff (\mathcal{E}(e_1,\ldots,e_n)) \vdash_\sigma v$$

and similarly for $\mathcal{E}'$. Therefore,

$$(\mathcal{E}(e_1,\ldots,e_n)) \vdash_\sigma v \iff (\mathcal{E}'(e_1,\ldots,e_n)) \vdash_{\sigma'} v.$$

□

The above theorem is used primarily for the next two theorems, which we will use to compute automatically closed solutions in loops.

Theorem 8 (Left closed-form exponentiation) *Given a language in which all primitive operations are deterministic, let Δ_Γ be the transfer relation for control path Γ, $\mathcal{E}$ be a (unary) expression constructor, and x be some variable that does not appear in either the syntax of Δ_Γ or $\mathcal{E}$. If*

$$\mathsf{E}\,e\,\Delta_\Gamma = \mathcal{E}\,e$$

and

$$\mathsf{E}\,(\mathcal{E}\,x)\,\Delta_\Gamma = \mathcal{E}\,x$$

then for all $n \geq 0$ and $k \geq 0$, Γ^n transfers $(\mathcal{E}^{(n+k)}\,e)$ to $(\mathcal{E}^k\,e)$.

Proof: Straightforward application of Theorems 5, 6, and 6abs-exptr. □

Theorem 9 (Right closed-form exponentiation) *Given a language in which all primitive operations are deterministic, let Δ_Γ be the transfer relation for control path Γ, $\mathcal{E}$ be a (unary) expression constructor, and x be some variable that does not appear in either the syntax of Δ_Γ or $\mathcal{E}$. If*

$$\mathsf{E}\,(\mathcal{E}\,e)\,\Delta_\Gamma = e$$

and

$$\mathsf{E}\,(\mathcal{E}\,x)\,\Delta_\Gamma = \mathcal{E}\,x$$

then for all $n \geq 0$ and $k \geq 0$, Γ^n transfers $(\mathcal{E}^k\,e)$ to $(\mathcal{E}^{(n+k)}\,e)$.

Proof: Straightforward application of Theorems 5, 6, and 6abs-exptr. □

The following example illustrates the above development.

Example 30 *Let Δ be the transfer relation for one iteration of the loop:*

$$\textbf{while}\ \texttt{x <> nil}\ \textbf{do}\ \texttt{x} := \texttt{x.tl}$$

Note that variable `z` *does not appear in Δ. Let $\mathcal{E} = \bigcirc.\texttt{tl}$ and compute*

$$\mathsf{E}\,\texttt{x}\,\Delta = \texttt{x.tl} = \mathcal{E}\,\texttt{x}$$

and

$$\mathsf{E}\,(\mathcal{E}\,\texttt{z})\,\Delta = \mathsf{E}\,(\texttt{z.tl})\,\Delta = \texttt{z.tl} = \mathcal{E}\,\texttt{z}.$$

By Theorem 8, we conclude that any n iterations of the loop transfers $\mathcal{E}^n\,\texttt{x} = \texttt{x.tl}^n$ to `x`*, and further that it transfers $\texttt{x.tl}^{n+k}$ to $\texttt{x.tl}^k$ for any $k \geq 0$.*

9.5.3 Computing closed forms automatically

In order to compute these closed forms automatically for expression e and control path Γ, it is necessary to determine automatically an expression constructor $\mathcal{E}$ such that

$$\mathsf{E}\,e\,\Delta_\Gamma = \mathcal{E}\,e$$

where Δ_Γ is the transfer relation for control path Γ. In general, there may be many choices for $\mathcal{E}$ that make this equation true. For instance, for both of our example while-loops above,

$$\mathsf{E}\,\texttt{x}\,\Delta = (\bigcirc.\texttt{tl})\,\texttt{x}$$

and

$$\mathsf{E}\,\texttt{x}\,\Delta = (\texttt{x.tl})\,\texttt{x}.$$

To understand this nondeterminism, consider the task of determining from any two expressions e and e' an expression constructor $\mathcal{E}^{e}_{e'}$ such that $\mathcal{E}^{e}_{e'}\, e = e'$. Using the first two lines of the above definition of expression constructors as functions, we can derive the following scheme:

$$\begin{aligned} \mathcal{E}^{e}_{x} &= x \\ \mathcal{E}^{e}_{p(e_1,\ldots,e_n)} &= p(\mathcal{E}^{e}_{e_1},\ldots,\mathcal{E}^{e}_{e_n}) \end{aligned}$$

But this merely reduces to the degenerate $\mathcal{E}^{e}_{e'} = e'$, an expression constructor without any holes. Taking into consideration the third line of the definition, we can add the following equation:

$$\mathcal{E}^{e}_{e} = \bigcirc$$

Now whenever e' matches e, we have a choice between applying this equation to introduce a hole or use the first two to reconstruct e.

One obvious deterministic strategy is to introduce holes whenever possible, yielding the following algorithm:

$$\mathcal{E}^{e}_{e'} = \begin{cases} \bigcirc & \text{if } e = e' \\ x & \text{if } e \neq e' = x \\ p(\mathcal{E}^{e}_{e_1},\ldots,\mathcal{E}^{e}_{e_n}) & \text{if } e \neq e' = p(e_1,\ldots,e_n) \end{cases}$$

Example 31 *The expression-constructor algorithm computes:*

$$\begin{aligned} \mathcal{E}^{\mathtt{x}}_{\mathtt{x.tl}} &= \mathcal{E}^{\mathtt{x}}_{\mathtt{x}}.\mathcal{E}^{\mathtt{x}}_{\mathtt{tl}} \\ &= \bigcirc.\mathtt{tl} \end{aligned}$$

Example 32 *The expression-constructor algorithm computes:*

$$\begin{aligned} \mathcal{E}^{\mathsf{j}}_{\mathsf{a[j\ +\ j]}} &= \mathcal{E}^{\mathsf{j}}_{\mathsf{a}}[\mathcal{E}^{\mathsf{j}}_{\mathsf{j\ +\ j}}] \\ &= \mathsf{a}[\mathcal{E}^{\mathsf{j}}_{\mathsf{j}} \mathbin{\mathsf{+}} \mathcal{E}^{\mathsf{j}}_{\mathsf{j}}] \\ &= \mathsf{a}[\bigcirc \mathbin{\mathsf{+}} \bigcirc] \end{aligned}$$

This suggests the following algorithm for computing closed forms of expression transfer in loops.

Algorithm 1 (Closed-forms of expressions in loops) *Given the following input:*

- *A control path* $\Gamma \in \mathsf{CtrlPoint}^{+}$.
- *An expression* $e \in \mathsf{Exp}$.

Perform the following steps:

1. *Compute the transfer relation* Δ_Γ *for control path* Γ *as described in Chapter 4.*
2. *Compute the expression* $(\mathsf{E}\, e\, \Delta_\Gamma)$. *Call this* e'.

3. *Compute the expression constructor* $\mathcal{E}^{e}_{e'}$ *as described above. Call this* $\mathcal{E}_L$*.*

4. *Compute the expression constructor* $\mathcal{E}^{e'}_{e}$ *as described above. Call this* $\mathcal{E}_R$*.*

5. *Choose a variable* x *not appearing in* Δ_Γ*.*

6. *Compute the expression* $(\mathcal{E}_L\, x)$ *as described by the definition of expression constructors. Call this* e_L*.*

7. *Compute the expression* $(\mathcal{E}_R\, x)$ *as described by the definition of expression constructors. Call this* e_R*.*

8. *Compute the expression* $(\mathsf{E}\, e_L\, \Delta_\Gamma)$ *and test if it is syntactically equal to* e_L*. If so, output "***left(*$\mathcal{E}_L$***)***". Otherwise, output "***left-exponentiation not found***".*

9. *Compute the expression* $(\mathsf{E}\, e_R\, \Delta_\Gamma)$ *and test if it is syntactically equal to* e_R*. If so, output "***right(*$\mathcal{E}_R, e'$***)***". Otherwise, output "***right-exponentiation not found***".*

If this algorithm given Γ and e outputs "**left(**$\mathcal{E}_L$**)**" then for all $n \geq 0$ and $k \geq 0$, Γ^n transfers $(\mathcal{E}^{(n+k)}\, e)$ to $(\mathcal{E}^{k}\, e)$. In addition, if it outputs "**right(**$\mathcal{E}_R, e'$**)**" then for all $n \geq 0$ and $k \neq 0$, Γ^n transfers $(\mathcal{E}^{k}\, e')$ to $(\mathcal{E}^{(n+k)}\, e')$.

Example 33 *The above algorithm, given the control path corresponding to one iteration of the program*

```
while x <> nil do x := x.tl
```

and given the expression `x`*, outputs "***left(*○`.tl`***)***" and "***right-exponentiation not found***".*

Example 34 *The above algorithm, given the control path corresponding to one iteration of the program*

```
while x <> nil do
{
   y := x;
   x := x.tl;
   y.tl := x.tl
}
```

and given the expression `x`*, outputs "***left-exponentiation not found***" and "***right-exponentiation not found***".*

Part V

Conclusion

In this dissertation, we have presented a new way of approaching the problem of statically analyzing a program to determine properties of its run-time behavior.

In our methodology, the semantic definition of a language is given by a translation from the source program to an intermediate form in which all single step transitions between two control points are described by a single transfer relation term in TR. For instance, we have shown how to translate assignments and let-bindings into assignment relations, conditionals into filter relations, allocation into assignment relations that maintain an explicit heap pointer, and function calls into filter relations with assignment for argument passing. Our language TR if transfer relations is thus a universal intermediate representation for programming languages, parameterized by a set Val of values and Primop of primitive operations.

The semantics itself merely defines the single-step transfer relations, which amounts to a translation of the source program into TR. But the fundamental property of TR that sets it apart from other intermediate representations and makes it useful for program analysis is that it is closed under composition. We have given an algorithm $\oplus$ to perform this composition.

Given this view, one way to think of our analysis methodology is as a kind of *symbolic execution.* Given the translation of a source program into TR, one uses $\oplus$ to compose the steps in order to generate a transfer relation (term in TR) of a particular finite control path. The single-step transfer relations yielded by the semantics correspond to the length-two control paths and are simply a rewriting of the program text. But as an analysis composes these steps with $\oplus$, it symbolically uncovers more and more dynamic information about the program.

Unlike usual approaches to program analysis that begin by defining an abstract language of run-time properties, our methodology never discards information about the program. In fact, given a closed program (in other words, no parameters or unknown data), it is possible actually to execute the program with $\oplus$. To do this, build the transfer relation for the control path that starts at the beginning of the program. At each point during this incremental composition, all information about the run-time state up to that point will be represented precisely in the transfer relation, and every time a branch point is reached (for instance, conditional or function call), only one branch will result in a non-$\emptyset$ transfer relation. Of course, if the program never terminates then this process will never terminate. But it demonstrates that our analysis methodology includes all information needed to perform a precise execution of the source program, which sets it apart from other approaches to program analysis.

But the point of program analysis is usually to analyze a program or program fragment that is not closed. One may want to analyze a function relative to its parameters, or a segment of C code apart from its surrounding context. Or the entire program itself may not be closed because of unknown input data. It is these situations for which our methodology is designed. As the analysis symbolically builds the transfer relation for a control path, it may encounter unknown quantities (variables, heap references, and so forth). The analysis represents these as expressions and l-expressions in the transfer relations, precisely describing quantities that are *relative* to the state of execution at the beginning of the control path. Still, no semantic information is discarded. If a transfer relation that describes the definition of a variable or value on the heap is composed with a transfer relation that includes a reference to that variable or heap location, then the $\oplus$ algorithm will inline the data defined in the first relation into the references in the second transfer relation and simplify the result.

In short, our primary philosophy is that program analysis should focus on the relationship between an execution state at the beginning of a given control path and the resulting execution state at the end of the path. Given this philosophy, our primary technical result is that one can in fact effectively compute a concise symbolic description of this precise relationship for a fixed control path.

So, our methodology truly is a general framework for program analysis in the sense that it involves no abstraction or approximation, and so there is nothing in the framework itself that *necessarily* prohibits the computation of any given computable program property. Of course, this is true of the text of the source program itself! But repeated applications of $\oplus$ reveal more and more dynamic information about the source program, and in the limit actually represent the entire program execution.

One may view repeated applications of $\oplus$ as an imperative analog of the reduction of terms in the λ-calculus. The redex rules of the λ-calculus are symbolic, just like the composition of transfer relations, and repeated reductions of a λ-term reveal in some sense more and more dynamic information about the original term. The reduction may terminate in a unique (up to α-conversion) normal form, which is a canonical representation of the original term. One may think of repeated applications of $\oplus$ in a similar way, gradually moving toward a more canonical representation of the source program, in principle resulting in a single TR-term in the limit. In our case, these normal forms are not unique. But this is not surprising, given the wide variety of languages that we can describe in this way—languages with assignment, heap-allocated data structures, mutable arrays and records, and pointers.

A new methodology of program analysis opens up numerous avenues for future work. In this dissertation, we have just begun to explore the applications of our methodology to real analysis problems, but there is much more work to be done. Some thoughts:

- There is potential for our methodology to help in software development as a debugging tool. The transfer relation terms in TR have an intuitive presentation as symbolic conditional parallel assignments. Imagine, for instance, dragging a mouse through an execution path in a source program—around loops, down conditionals, into function calls, and so forth—and watching the transfer relation describing the precise behavior of that path build up incrementally. There may arise a few terms in the transfer relation that would correspond to the internals of the semantics rather than anything appearing explicitly in the source program—the variable `H` that we used as a heap pointer in the MINI-C semantics, for instance—and the user would have to learn these. But for the most part, the output would appear quite natural. Such a debugger would not only be useful to help understand what the code does, but could catch bugs or potential sources of bugs. In particular, because $\oplus$ computes the precise composition, it is guaranteed to cover all aliasing possibilities, and these are revealed as aliasing tests in the resulting transfer relation.
- We have isolated the problem with existing approaches to program analysis that they construct an abstract property a single step at a time, often resulting in dramatic loss

of precision. If an existing analysis is retooled to work for TR, then this fundamental limitation is eliminated because the analysis is free to apply $\oplus$ to compose multiple steps and thus build the abstract property multiple steps at a time. The great advantage to this idea is that it is completely general and can potentially improve the accuracy of *any* existing program analysis. For any given existing program analysis, however, there are the following practical issues.

  - Retooling the analysis for TR. The TR language is fairly simple, but still contains parallel assignment, and most existing analyses would need to be extended to handle parallel assignment. For some analyses, this is probably not difficult, and we described a general approach for value analyses. But for others, such as shape analyses, a general solution may be more difficult.
  - Designing a strategy of when to apply $\oplus$ to build multiple steps and when to apply the analysis on those compound steps. The obvious general approach to this task is to use $\oplus$ on control paths whose endpoints are control points with potentially more than one incoming edge in a control graph. This is a generalization of composing steps in a basic block, but still guarantees termination. There may be instances, however, where further composition would improve precision, and we leave the design of such strategies as a problem for future work.

- Some analysis problems, such as lifetime analysis and dependency analysis, deal directly with the relation between one point in the execution and some later point in the execution. Therefore, these analyses are actually abstractions of transfer relations. This implies that our methodology may be fundamentally better suited to such problems than the traditional approach of computing a property of the states reached during execution.

- Our methodology is probably well suited for the analysis of concurrent programs. One may treat a process-creation point as a branch in the control-flow graph of a program. Then one may build the transfer relations for each branch of the path, to relate precisely the data in the old process with the data in the new process, at least up to a certain finite path of execution in each process. For instance, this is useful for determining that a communication channel is used in a restricted fashion between two processes.

- We have demonstrated that it is possible to achieve some symbolic closed-form solutions that track data through loops. There may be other ways in which information about a loop can be computed symbolically. For instance, if we switch the order of the two arguments of the E function, yielding functionality

$$\mathsf{E} \in \mathsf{TR} \to \mathsf{Exp} \to \mathsf{Exp},$$

then any fixed point of $(\mathsf{E}\,\Delta)$ is an expression that evaluates to the same value before and after transfer relation Δ. If Δ is the transfer relation of a path through one iteration of a loop, then these fixed points are loop-invariant expressions, and any binary-valued fixed point is a loop invariant. The ability to express a single iteration of a loop as a concise term Δ gives some hope that there are useful ways to compute effectively these fixed points.

- Program transformations, such as classical compiler optimizations, are ultimately based on semantic equivalence of code fragments. One code fragment may be replaced with another if and only if they are semantically equivalent, for some appropriate notion of semantic equivalence. We have given a way of producing a term describing the semantic behavior of any finite control path in the source program. This term is not canonical, but it is more abstract than the source program itself and thus is more amenable to reasoning about semantic equivalence. Indeed, syntactic equivalence of composed transfer relations can be quite useful in practice as an approximation to semantic equivalence. There is hope that our methodology could form the basis of a generic calculus of program transformations for use in optimizing compilers for a wide variety of languages, including imperative languages.

Bibliography

[AH87] S Abramsky and C Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis-Horwood, 1987. (p 68)

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, 1992. (p 100)

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986. (pp 3, 42, 52, 136)

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994. (p 12)

[Bar81] D. W. Barron. *Pascal – the language and its implementation.* John Wiley & Sons Ltd, Chichester, England, 1981. (p 97)

[Bar84] H. P. Barendregt. *The Lambda Calculus.* North Holland, revised edition, 1984. (pp 67, 98, 112)

[BHA86] G L Burn, C L Hankin, and S Abramsky. Strictness analysis for higher-order functions. *Sci. Comp. Prog.*, 7:249–278, 1986. (pp 68, 102)

[Bou93a] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation: Albuquerque, New Mexico, June 23–25, 1993*, volume 28-6 of *SIGPLAN notices; ISSN: 0362-1340; v. 28, no. 6 (June 1993)*, pages 46–55, New York, NY, USA, June 1993. ACM Press. (pp 3, 11)

[Bou93b] F. Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. *Lecture Notes in Computer Science*, 717:501–??, 1993. (p 3)

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM. (pp 4, 42, 126, 127)

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282. ACM, ACM, January 1979. (pp 4, 127)

[CC92a] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, Leuven, Belgium, 1992. LNCS 631, Springer-Verlag. (p 4)

[CC92b] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992*, pages 83–94, New York, NY, USA, 1992. ACM Press. ACM order number 54990. (pp 4, 11, 69)

[CC92c] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, July 1992. (p 4)

[CC92d] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992. (pp 4, 131)

[CC94] Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proceedings of the 1994 International Conference on Computer Languages, ICCL'94*, pages 95–112, Toulouse, France, May 1994. IEEE Computer Society Press. (p 4)

[CC95] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 170–181, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press. (p 4)

[CH78] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth annual ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM, ACM, January 1978. (p 4)

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximation de point fixes d'opérateurs monotone sur un treillis analyse sémantique des programmes.* PhD thesis, Grenoble, 1978. (p 127)

[Cou81] P. Cousot. Semantic foundations of program analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981. (p 4)

[Cou90] Patrick Cousot. Methods and logics for proving programs. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 15, pages 841–993. The MIT Press, New York, N.Y., 1990. (p 4)

[CWZ90] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation*, pages 296–310, June 1990. (p 3)

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972. (p 115)

[Deu92] Alain Deutsch. *Modèles opérationnels de langages de programmation et représentations de relations sur des langages rationnels avec application à la détermination statique de propriétés de partages dynamiques de données.* Thèse de doctorat d'université, Université Pierre et Marie Curie (Paris 6), Paris (France), April 1992. (pp 4, 102)

[Deu94] Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.* (pp 3, 4)

[Dij76] E W Dijkstra. *A Discipline of programming.* Prentice-Hall, 1976. (p 22)

[FF86] M. Felleisen and D.P. Friedman. Control operators, the secd-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, August 1986. (p 69)

[FRT95] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 379–392, San Francisco, California, January 1995. (p 4)

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 1–15, St. Petersburg, Florida, January 21–24, 1996. ACM Press. (pp 9, 136)

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1989. (pp 67, 98)

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, pages 165–199, 1989. (p 4)

[Gra91a] P. Granger. *Analyses Sémantiques de Congruence.* PhD thesis, Ecole Polytechnique, Palaiseau, France, July 1991. (p 4)

[Gra91b] P. Granger. Static analysis on linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer Verlag, 1991. (p 4)

[H$^+$92] P. Hudak et al. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5):Section R, 1992. (p 97)

[Hei92] Nevin Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992. (pp 12, 102)

[How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980. (p 98)

[HP79] M. Hennessy and G. D. Plotkin. Full abstraction for a simple programming language. In J. Bečvář, editor, 8th *Symposium on Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120. Springer-Verlag, 1979. (p 39)

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990. (p 4)

[JM79] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Conference Record of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 244–256, January 1979. (p 3)

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Passau, Germany, February 1987. Proceedings published as Springer-Verlag Lecture Notes in Computer Science 247. The paper is also available as INRIA Report 601, February, 1987. (p 69)

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976. (p 4)

[KMP84] G. Kahn, D. B. MacQueen, and G. Plotkin. *Semantics of Data Types.* Springer Verlag (Heidelberg, FRG and NewYork NY, USA), Internat.Symp.Proc., ; ACM CR 8504-0266, 1984. (p 12)

[Knu71] D. E. Knuth. An empirical study of FORTRAN programs. *Software - Practice and Experience*, 1(12):105–134, 1971. Motivation for optimization. (p 97)

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Progamming Language.* Prentice Hall, Englewood Cliffs, 1 edition, 1978. (p 97)

[Kra88] David Kranz. Orbit: An optimizing compiler for scheme. Computer science technical report #632 (ph.d. dissertation), Yale University, 1988. (p 100)

[KU76] J. B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976. (p 3)

[Lan91] W. Landi. Interprocedural aliasing in the presence of pointers. Phd thesis, Rutgers University, 1991. (p 3)

[LD93] Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In *Conference Record of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 124–136, Charleston, South Carolina, January 1993. (p 99)

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *International Conference on Functional Programming and Computer Architecture*, June 1995. (pp 24, 69)

[MJ81] S. S Muchnick and N. D. Jones. *Program flow analysis: theory and applications.* Prentice-Hall, Englewood Cliffs, NJ, 1981. (pp 3, 12)

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990. (pp 3, 24, 97)

[Mul87] Ketan Mulmuley. *Full Abstraction and Semantic Equivalence.* MIT Press, Cambridge, Massachusetts, 1987. Ph. D. Dissertation, Carnegie Mellon University, August 1985. (pp 39, 127)

[Myc81] Alan Mycroft. Abstract Interpretation and Optimising Transformations for Applicative Programs. Ph.D. Thesis, Univ. of Edinburgh, December 1981. (p 68)

[Nie84] Flemming Nielson. Abstract Interpretation using Domain Theory. Ph.D. Thesis, Univ. of Edinburgh, October 1984. (p 68)

[Nie86] F Nielson. Abstract Interpretation of Denotational Definitions. In *STACS'86*, volume 210 of *LNCS*, pages 1–20. Springer-Verlag, 1986. (p 68)

[OT95] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995. (p 120)

[Plo75] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975. (pp 100, 101)

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981. (p 69)

[ReC86] Jonathan A. Rees and eds. Clinger, William C. Revised^3 report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986. (pp 23, 97)

[Sch95] D. A. Schmidt. Natural-semantics-based abstract interpretation. *Lecture Notes in Computer Science*, 983:1–??, 1995. (p 69)

[Sco70] Dana Scott. Outline of a mathematical theory of computation. In *Proceedings, Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–176. Princeton University, 1970. Also, Programming Research Group Technical Monograph PRG–2, Oxford University. (p 67)

[Sco76] Dana S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–587, 1976. (p 67)

[Sco82] Dana S. Scott. Domains for denotational semantics. In *Proceedings International Colloquium on Automata, Languages, and Programming '82*, 1982. (p 67)

[Shi91] Olin Grigsby Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, Carnige-Mellon Univeristy, May 1991. Also available as CMU-CS-91-145. (pp 3, 102)

[Sie94] Kurt Sieber. Full abstraction for the second order subset of an algol-like language. In Igor Prívara, Branislav Rovan, and Peter Ruzicka, editors, *Mathematical Foundations of Computer Science 1994 19th International Symposium*, volume 841 of *LNCS*, pages 608–617, Kosice, Slovakia, 22–26 August 1994. Springer. (p 120)

[SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 16–31, St. Petersburg, Florida, January 21–24, 1996. ACM Press. (pp 9, 125, 126)

[SS90] Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990. (p 97)

[Ste78] Guy L. Steele. Rabbit: A compiler for Scheme. Technical Report 474, Massachusetts Institute of Technology, Cambridge, MA, May 1978. (p 100)

[Sto77] J. Stoy. *Denotational Semantics.* The MIT Press, Cambridge, Mass, 1977. (pp 67, 99)

[TJ92] J.-P. Talpin and P. Jouvelot. The type and effects discipline. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 162–173, 1992. (p 102)

[Wad87] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation). In S Abramsky and C Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis-Horwood, 1987. (p 102)

[Wan77] Mitchell Wand. A characterization of weakest preconditions. *Journal of Computer and Systems Science*, 15:209–212, 1977. (p 22)