Path Analysis and the Optimization of Non-strict Functional Languages

Adrienne Gael Bloss

YALEU/DCS/RR-704
May 1989

A dissertation presented to the faculty of the Graduate School of Yale University in candidacy for the degree of Doctor of Philosophy.

Path Analysis and the Optimization of Non-strict Functional Languages

A Dissertation

Presented to the Faculty of the Graduate School

of

Yale University

in Candidacy for the Degree of

Doctor of Philosophy

by

Adrienne Gael Bloss

May 1989

ABSTRACT

Path Analysis and the Optimization of Non-strict Functional Languages

Adrienne Gael Bloss

Yale University

1989

The functional programming style is increasingly popular in the research world, but functional languages still execute slowly relative to imperative languages. This is largely because the power and flexibility of functional languages restrict the amount of information readily available to the compiler, hindering its ability to generate good code. This dissertation demonstrates that information about the *order of evaluation of expressions* can be statically inferred for non-strict functional programs, and that optimizations based on this information can provide substantial speedups at runtime.

We present an exact, non-standard semantics called *path semantics* that models order of evaluation in a non-strict sequential functional language, and its computable abstraction, *path analysis*. We show how the information inferred by path analysis can be used for two important optimizations: destructive aggregate updating, in which updates on functional aggregates that are provably not live are done destructively; and more efficient thunks, in which the evaluation status of delayed objects is determined statically whenever possible, eliminating the need for runtime tests. Benchmarks for these optimizations are presented, along with benchmarks for the analyses themselves. Although the full analysis is found to be impractical for large programs, it should serve as the basis for further abstraction.

Alternative models of order of evaluation are also discussed, including a less expensive but less general model for a sequential system, and the extensions that would be required to apply path analysis to a parallel system.

Acknowledgements

I would first like to thank my advisor Paul Hudak, whose insight and guidance were invaluable to this thesis. I would also like to thank the other members of my committee, Alan Perlis and Paul F. Reynolds, Jr., for their timely and careful reading of this document and their helpful comments.

Financial support for this work was provided in part by the National Science Foundation Presidential Young Investigator Grant #DCR-8451415.

This work benefited greatly from many enlightening discussions with Jonathan Young. It also benefited from the stimulating environment and suggestions provided by other members of the Lisp and functional programming group at Yale, including Ben Goldberg, Jim Philbin, David Kranz, Steve Anderson, Juan Guzman, Richard Kelsey, and Lauren Smith. I would like to thank my officemate Rick Mohr, who assured me daily that I would in fact finish eventually (and who was right!), and my other friends at Yale for making my graduate school years enjoyable as well as productive.

Most of all, I would like to thank my family: my husband Andy, who got me through all the ups and downs of graduate school, and without whom this thesis never would have been written; my brother Greg, who provided much support and inspiration; my mother, who encouraged my graduate school career in every way and never lost faith in me; and my father, who gave his daughter his love of puzzles and problems and who would have been very proud. This thesis is dedicated to his memory.

Contents

Chapter 1

Introduction

1.1 Overview

Functional programming languages are becoming increasingly popular in the research community because they offer clean semantics, lazy evaluation, and no side effects. However, their acceptance into the "real world" has been hindered by their typically slow execution relative to that of imperative languages. This slower execution stems largely from the power and flexibility of functional languages; since fewer restrictions are placed on the programmer, fewer assumptions can be made by the compiler and it is more difficult for it to generate good code. It is therefore natural to optimize functional languages by using compile-time inferencing techniques to gather as much specific information as possible about each program.

This thesis explores the optimization of sequential lazy functional languages through the compile-time inference of the *order of evaluation of expressions*. We focus on an abstract non-standard semantics called *path analysis* to compute the order of evaluation information, and show how path analysis can be applied to strictness analysis, destructive aggregate update analysis, and thunk analysis. Although each of these problems has been studied to some extent in the past, we extend previous work for aggregate updating [34,32,18,15,6] and thunk analysis [24] and present a new approach to strictness analysis [27,10,8,21,22]. Furthermore, we

know of no other work that recognizes the importance of a general model of order of evaluation, and that uses such a model as a basis for a variety of optimizations. We also present an alternative model of order of evaluation for a sequential system, and discuss the implications of extending path analysis to a parallel system.

This introduction gives a brief overview of the features of functional languages; presents some of the issues involved in optimizing lazy functional languages; gives an overview of denotational semantics and abstract interpretation, the theoretical foundations on which path analysis is based; and presents the syntax and semantics for the generic functional language on which our analyses are based. An outline of the body of the thesis is also presented.

1.2 Functional Languages

Functional languages have been emerging and changing for nearly 30 years, and have many different forms. However, most *modern* functional languages share the following characteristics:

- *Mathematical notation.* Functional programs are typically written in a concise mathematical notation which can be viewed as a syntactic sugaring of Church's λ-calculus.[9].

- *Lexical scoping.* Although one of the early functional languages (Lisp) had dynamic scoping, virtually all other functional languages are lexically scoped.

- *No side effects.* Functional languages have no concept of a modifiable state. They are made up entirely of *expressions*, where the meaning of an expression is the value to which that expression evaluates. The evaluation of an expression is guaranteed to have no other effect on the program, that is, no *side-effect.* In imperative languages side-effects are most commonly achieved

through the assignment statement, but other forms include functions that contain concealed assignment (such as Lisp's RPLACA) and I/O.[1]. The absence of side-effects guarantees the mathematical property of *referential transparency*, by which identical expressions within the same lexical scope have identical values.

- *Non-strict semantics.* In *strict* semantics, typical of imperative languages, when a function f is applied all of its arguments are evaluated *at the time of the function call*, before entering the body of f. In *non-strict* semantics, an argument is not evaluated unless and until its value is demanded *inside* the body of f. A non-strict semantics has the advantage of terminating more often than a strict semantics, since the evaluation of a non-terminating expression may be avoided, but if both semantics terminate they are guaranteed to produce the same answer. Thus the non-strict semantics is more expressive than the strict semantics.

  Although there are several ways of implementing non-strict semantics, the method used by most functional languages is referred to as *lazy evaluation* because it guarantees that the arguments to a function will be evaluated the minimum possible number of times. The term *lazy* is often used ambiguously to mean non-strict with or without the connotation of actual lazy evaluation. It is sometimes used as such in this document, but disambiguation should be possible from context.

[1]Of course, functional languages do permit input and output; an excellent discussion of functional models of I/O may be found in [19]

1.3 The Role of Order of Evaluation in Optimizing Lazy Functional Languages

The inefficiencies that arise in implementing functional languages are usually from one of the following sources:

1. *Storage management.* Since functional languages have no notion of explicit storage management, the programmer is relieved of the burden (and robbed of the power) of managing memory efficiently. Since additional runtime overhead is usually undesirable, it is up to the compiler and runtime system to provide effective memory management.

2. *Lazy evaluation.* Lazy evaluation can provide efficiency benefits, since no argument is evaluated unnecessarily, but there is substantial overhead in maintaining an expression and its environment until they are demanded (or it can be shown that they never will be). Hence lazy evaluation often results in a net loss in efficiency.

The problem of storage management has received some attention, but most of the previous work has addressed only languages with strict semantics [32,18,16]. This is because effective memory management requires knowledge of future memory requirements, which in turn requires information about when expressions will be evaluated. Languages with strict semantics offer a substantial amount of information *at compile-time* about the order in which expressions will be evaluated, but lazy evaluation vastly reduces the availability of such information.

Most of the work in optimizing lazy evaluation has been through *strictness analysis* [27,7,21], in which a function's arguments are evaluated at the time of the function call if it can be shown that doing so will not change the termination properties of the program. Of course this "early evaluation" is not always safe, and

when evaluation must be delayed some of the associated costs are unavoidable. In true lazy evaluation, however, the cost of determining whether or not an expression has already been evaluated (so that re-evaluation can be avoided) can be eliminated if the evaluation status of the expression can be determined at compile-time, but this requires compile-time information about when expressions are evaluated.

In both of these examples, the missing piece is *compile-time information about the order of evaluation of expressions.* This thesis presents an interpretation for lazy functional programs called *path semantics* that extracts order of evaluation information, and a computable approximation to path semantics called *path analysis* that supports optimizations such as those suggested above.

The next section provides an introduction to *abstract interpretation*, the technique used to derive path analysis from path semantics.

1.4 Abstract Interpretation

As new optimizations arise for functional languages, so do new *semantic analyses.* In particular, *denotational semantics* and *abstract interpretation* have been recognized as powerful tools with a wide range of applications. Denotational semantics is a formal way of describing the meaning of a program in terms of mathematical domains that properly capture our intuition about program behaviors. In functional languages the "standard" meaning, or *standard interpretation* of an expression is what we intuitively think of as its *value*, whether that be a number, list, function, or whatever. However, in some applications a less precise meaning may be sufficient. For example, suppose we wish to know the sign of the product of two integers; we could perform the multiplication and then extract the sign of the result, or we could deduce its sign directly from the signs of the operands. The latter approach is arguably the easiest path to finding the desired result, in that manipulating the signs directly requires only part of the information required to do the actual

multiplication. For example, extracting $(-)$ from the result of $(+7) * (-5)$ takes more work than having a simple rule that says "$(+)*(-) = (-)$." Loosely speaking, an approximation to a value, such as the sign of an integer, is called an *abstraction*, and a computation over such abstract values is called an *abstract interpretation.*

Abstract interpretation has recently become popular as a general and effective semantic analysis technique, primarily in functional language circles, but also in other areas. The reasons for its popularity include the fact that it is a *formal* methodolgy that can be related directly back to the denotational semantics of the source language. This allows one to prove the correctness of an optimization at an abstract level, independently of operational concerns.

The formal theory of abstract interpretation has itself witnessed remarkable growth. It began with the Cousots' seminal work [11,12], but Mycroft's reformulation in an applicative (i.e. functional) idiom [28,27] and more recently Nielsons' [31,30] work have laid down a useful framework in which rather general theorems can be re-cast in particular application domains. The formal framework is itself outside the scope of this thesis, but the reader may find an excellent description of state-of-the-art developments in this area in the the recent textbook edited by Abramsky and Hankin [1].

1.5 A Generic Functional Language

The work in this thesis is implemented for a compiler for ALFL, a lazy functional language developed at Yale. However, the theoretical work is based on a "generic" functional language whose syntax and semantics are presented in this section. The first-order and higher-order languages are presented separately, as are the first-order and higher-order analyses.

1.5.1 A First-Order Lazy Functional Language

Notation and Abstract Syntax

The abstract syntax for the first-order language is given below:

c	$\in$	Con	constants
x	$\in$	Bv	bound variables
p	$\in$	Pf	primitive functions
f	$\in$	Fv	function variables
e	$\in$	Exp	expressions, where $e = c \mid x \mid p(e_1, ..., e_n) \mid f(e_1, ..., e_n)$
pr	$\in$	$Prog$	programs, where $pr = \{f_i(x_1, ..., x_n) = e_i\}$

Note that we assume that all nested lambda abstractions have been lifted to the top level [23], and we find paths through these top-level functions.

In the semantic equations that follow, double brackets surround syntactic objects, as in $\mathcal{E}[\![x_i]\!]$, and single brackets indicate environment update, as in $env[y/x]$; $[y_i/x_i]$ is shorthand for $\perp[y_1/x_1, ..., y_n/x_n]$, where the subscript bounds are inferred from context. For any domain D, D^n refers to the domain of n-tuples with each element drawn from D.

Standard First-Order Semantics

Semantic Domains

Int	the standard flat domain of integers.
$Bool$	the standard flat domain of boolean values.
$Bas = Int + Bool$	the domain of basic values
$Fun = \bigcup_{n=1}^{\infty}(Bas^n \rightarrow Bas)$	the domain of first-order functions.
$D = Bas + Fun + \{error\}$	the domain of denoteable values.
$Bve = Bv \rightarrow D$	the domain of bound variable environments

$Env = Fv \rightarrow D$ the domain of function environments

Semantic Functions

$$\mathcal{K} : Con \rightarrow Bas$$

$$\mathcal{P} : Pf \rightarrow Fun$$

$$\mathcal{E} : Exp \rightarrow Bve \rightarrow Env \rightarrow D$$

$$\mathcal{E}_p : Prog \rightarrow Env$$

$$\begin{aligned}
\mathcal{K}[\![n]\!] &= n, \text{integer } n \\
\mathcal{K}[\![true]\!] &= true \\
\mathcal{K}[\![false]\!] &= false \\
\mathcal{P}[\![+]\!] &= \lambda(x,y).\ (Int?(x) \text{ and } Int?(y)) \rightarrow x + y, error \\
\mathcal{P}[\![IF]\!] &= \lambda(x,y,z).\ (Bool?(x)) \rightarrow (x \rightarrow y, z), error
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\![c]\!]bve\ env &= \mathcal{K}[\![c]\!] \\
\mathcal{E}[\![x_i]\!]bve\ env &= bve[\![x_i]\!] \\
\mathcal{E}[\![p(e_1,\ldots,e_n)]\!]bve\ env &= \mathcal{P}[\![p]\!](\mathcal{E}[\![e_1]\!]bve\ env,\ldots,\mathcal{E}[\![e_n]\!]bve\ env) \\
\mathcal{E}[\![f(e_1,\ldots,e_n)]\!]bve\ env &= env[\![f]\!](\mathcal{E}[\![e_1]\!]bve\ env,\ldots,\mathcal{E}[\![e_n]\!]bve\ env) \\
\mathcal{E}_p[\![\{f_i(x_1,\ldots,x_n) = e_i\}]\!] &= env \text{ whererec} \\
env &= [\![(\lambda(y_1,\ldots,y_n).\mathcal{E}[\![e_i]\!][y_k/x_k]\ env)/f_i]\!]
\end{aligned}$$

1.5.2 A Higher-Order Lazy Functional Language

Higher-Order Syntax

First, we define the syntax of our lazy, higher-order functional language.

c	$\in$	Con	constants
x	$\in$	Bv	bound variables
f	$\in$	Fv	function variables
e	$\in$	Exp	expressions, where $e = c \mid x \mid f \mid e_1\ e_2 \mid \lambda x.e$
pr	$\in$	$Prog$	programs, where $pr = \{f_i\ x_1 ... x_n = e_i\}$

Again, we assume that all nested lambda abstractions have been lifted to the top level [23], and we find paths through these top-level functions.

Higher-Order Semantics

Semantic Domains

$$\begin{array}{lcll} Bas & = & Int + Bool, & \text{domain of basis values} \\ D & = & Bas + (D \to D), & \text{domain of denotable values} \\ Env & = & (Bv + Fv) \to D, & \text{the function environment} \end{array}$$

Semantic Functions

$$\mathcal{H} : Exp \to Env \to D$$

$$\mathcal{H}_p : Prog \to Env$$

$$\mathcal{H}_k : Pf \to D^n \to D$$

$$\begin{array}{rcl} \mathcal{H}_k[\![IF]\!]\ env & = & \lambda e_1.\lambda e_2.\lambda e_3.\ (Bool?\ e_1) \to (e_1 \to e_2, e_3), error \\ \mathcal{H}_k[\![+]\!] env & = & \lambda e_1.\lambda e_2.\ (Int?\ e_1) \text{ and } (Int?\ e_2) \to e_1 + e_2, error \end{array}$$

$$\begin{array}{rcl} \mathcal{H}[\![x_i]\!]\ env & = & env[\![x_i]\!] \\ \mathcal{H}[\![c]\!]\ env & = & \mathcal{H}_k[\![c]\!] \\ \mathcal{H}[\![\lambda x.e]\!] & = & \lambda y.\mathcal{H}[\![e]\!]\ env[y/x] \\ \mathcal{H}[\![e_1\ e_2]\!]\ env & = & (\mathcal{H}[\![e_1]\!]env)(\mathcal{H}[\![e_2]\!]env) \\ \mathcal{H}_p[\![\{f_i = e_i\}]\!] & = & henv \text{ whererec} \\ \multicolumn{3}{c}{henv = [(\mathcal{H}[\![e_i]\!]\ henv)/f_i]} \end{array}$$

1.6 Outline

This dissertation is organized as follows:

- *Chapter 2: The Path Model of Order of Evaluation.* This chapter describes *path semantics*, an exact but incomputable model of order of evaluation, and *path analysis*, an inexact but computable model. The relationships among these two semantics and the standard semantics are shown, and theoretical issues such as domain construction are discussed.

- *Chapter 3: Aggregate Updating.* This chapter discusses the efficiency problems associated with aggregate updating in functional languages and presents a solution called *update analysis* that is based on path analysis. Theoretical issues concerning update analysis are discussed here; implementation issues appear in Chapter 6.

- *Chapter 4: Other Applications.* This chapter shows how path analysis can be used for strictness analysis and optimizing lazy evaluation.

- *Chapter 5: Implementation Issues.* This chapter discusses the issues that arise in implementing path analysis and update analysis, and suggests where a practical system might deviate from the models.

- *Chapter 6: Benchmarks.* This chapter benchmarks the analyses that are described in Chapters 3 and 4, both in time required for the analyses and in the speedup they produce.

- *Chapter 7: Other Models of Order of Evaluation.* This chapter discusses issues in modeling order of evaluation in a parallel system and presents an alternative model of order of evaluation for a sequential system.

- *Chapter 8: Related Work, Conclusions, and Future Work.*

Chapter 2

The Path Model of Order of Evaluation

As discussed in Chapter 1, compilers for functional languages lack information about the order in which expressions are evaluated, thus restricting the potential for optimization. In this chapter the *path model* for representing order of evaluation of expressions in a lazy sequential functional language is presented, first intuitively and then formally via *path semantics* and *path analysis*. Finally, the order in which expressions are *used* is shown to be as interesting as the order in which they are evaluated, and it is shown that order of use can also be described with the path model.

2.1 The Basic Model

Define a *path* through a function $f(x_1, ..., x_n)$ to be an *ordering on the evaluation of the arguments to* f. In a sequential system, at runtime there is exactly one path through each invocation of f. For example, consider the following simple function:

$$f(x, y) = x + y$$

Assuming left-to-right evaluation of the arguments to strict primitive operators such as + (this assumption is discussed further below and in Section 5.1.3), and using

angle brackets $\langle\rangle$ to enclose paths, the path through f is $\langle x, y\rangle$. Of course, the path taken through one function may depend on the path taken through another function, as illustrated by g, where f is defined as above:

$$g(a, b, c) = b + f(c, a)$$

The path through g is $\langle b, c, a\rangle$, but determining this relies on knowing that the path through $f(c, a)$ is $\langle c, a\rangle$.

Now consider the following function:

$$h(x, y, z) = \text{if } (x = 0) \text{ then } y \text{ else } z$$

Here the predicate $(x = 0)$ must be evaluated before we can tell whether y or z will be evaluated next. Thus some invocations of h might take the path $\langle x, y\rangle$, while others take $\langle x, z\rangle$. Without knowing the value of x, h could be said to have two *possible* paths, $\langle x, y\rangle$ and $\langle x, z\rangle$.

Consider the following points about paths:

- *A path through f need not contain every one of f's bound variables.* This follows directly from lazy evaluation. A bound variable x is evaluated if and when its value is demanded, and if f can compute a value without evaluating x, the paths that reflect such sequences of evaluations will not contain x.

- *A bound variable may appear at most once in any given path.* Again, this follows from lazy evaluation; since no recomputation is performed, a bound variable whose value is demanded more than once is still only *evaluated* once (at the first demand), and it is this point of evaluation that determines where the bound variable appears in the path.

- *There are a finite number of possible paths through a function.* Since no bound variable can appear more than once in a path, the number of possible paths

through a function of n arguments is bounded by $n! + (n-1)! + (n-2)! + \ldots + (n-n)! + 1$, where the extra path represents the non-terminating path, which is discussed below.

If a function does not evaluate any of its arguments but does return a value (e.g., $f(x,y) = 5$), it is said to take the *empty path*, denoted $\langle\rangle$. If a function does not return any value, that is, it fails to terminate, it is said to take the *bottom path*, denoted $\perp_p$. Note that a path may fail to terminate in many different ways, since it may evaluate any number of its arguments in any order during its non-terminating computation. These different non-terminating paths could be distinguished just as distinct terminating paths are distinguished, by recording which arguments are evaluated during the computation. For now we will equate all non-terminating paths with $\perp_p$; the issues that arise in distinguishing among these paths are discussed in Section 2.3.5.

The path model just described assumes a sequential model of evaluation in that it imposes left-to-right evaluation of arguments to strict primitive operators. While this could easily be modified to right-to-left, or even a runtime choice between the two, neither of these alternatives fits a model of *parallel* computation in which the arguments could be *interleaved.* For now, the reader should keep in mind that the path model assumes a sequential model of evaluation; order of evaluation in the context of a parallel system is discussed in Section 7.1.

We have presented an informal model for describing order of evaluation in a lazy sequential functional language, but have not discussed issues of precision, computability or correctness. These issues can be addressed by formalizing the model through denotational semantics, which we do in the following sections.

2.2 First-Order Path Semantics

As discussed in Chapter 1, denotational semantics is an exact and concise method of attaching meaning to syntax. In the context of paths, it is the "order of evaluation meaning" that should be attached to the syntax, where order of evaluation information is represented by the path model described above. Examination of the standard semantics presented in Section 1.5 reveals that it contains no order of evaluation information: under the standard semantics the meaning of a program is the *answer* that it returns, without regard to the order in which expressions are evaluated to obtain that answer. In this section we introduce *path semantics*, an exact but *non-standard* semantics that describes the operational notion of order of evaluation in a program. Since the order of evaluation at runtime depends on the values themselves (e.g., at the conditional the value of the predicate determines which arm will be evaluated), path semantics contains the information of the standard semantics as well as path information. Indeed, the standard interpretation can be shown to be an abstraction of path semantics.

Let $Path$ be the flat domain of paths, with bottom element $\perp_p$ representing non-termination. Thus any two terminating paths are considered incomparable, and non-termination is considered weaker than any form of termination. $Path$ is defined by:

$$Path = \{\perp_p\} \cup \{\langle d_1, ..., d_n\rangle \mid n \geq 0, \forall i, 1 \leq i < n, d_i \in Bv\}$$

Note that the elements of a path are bound variables. We are assuming that each function has a unique set of bound variables, i.e., that the program has been alpha-converted to ensure that no bound variable name is shared by more than one function.

The only operator required on paths is the path-append operator ":", defined below:

$\forall p \in Path,\ x_i \in D,\ 1 \leq i \leq n,\ n > 0$

$$
\begin{array}{rcll}
p : \bot_p & = & \bot_p & \\
\bot_p : p & = & \bot_p & \\
p : \langle\rangle & = & p & \\
\langle\rangle : p & = & p & \\
\langle x_1, ..., x_m\rangle : \langle x_{m+1}, ..., x_n\rangle & = & \text{if} & x_{m+1} \in \{x_1, ..., x_m\} \\
 & & \text{then} & \langle x_1, ..., x_m\rangle : \langle x_{m+2}, ..., x_n\rangle \\
 & & \text{else} & \langle x_1, ..., x_m, x_{m+1}\rangle : \langle x_{m+2}, ..., x_n\rangle
\end{array}
$$

Note that $\bot_p$ is strict in both arguments, since non-termination in any portion of a path will cause the entire path to be non-terminating. Also, when two paths are appended, the elements of the second path that already appear in the first path do not appear a second time in the resulting path.

The semantics presented below returns the path value of an expression in a given environment, but computing this path value requires references to the standard semantics. We assume that *env*, the standard meaning of the program, has already been computed as its computation is independent of that of path semantics.

Semantic Domains

$$
\begin{array}{lcl}
D, & & \text{the flat domain of denotable values} \\
 & & \quad \text{(from the standard semantics)} \\
Path, & & \text{the flat domain of paths} \\
Pfun & = & \bigcup_{n=1}^{\infty}(D^n \rightarrow Path^n \rightarrow Path) \\
Penv & = & Fv \rightarrow Pfun, \\
 & & \text{the function envronment} \\
Pbve & = & Bv \rightarrow Path, \\
 & & \text{the bound variable environment}
\end{array}
$$

Semantic Functions

$$
\begin{array}{rcl}
\mathcal{P} & : & Exp \rightarrow Bve \rightarrow Pbve \rightarrow Penv \rightarrow Path \\
\mathcal{P}_k & : & Pf \rightarrow Pfun \\
\mathcal{P}_p & : & Prog \rightarrow Penv
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}[\![exp]\!]\ bve\ pbve\ penv = \text{case } [\![exp]\!] \text{ of} & & \\
\quad \mathcal{P}[\![c]\!]\ bve\ pbve\ penv & = & \langle\rangle \\
\quad \mathcal{P}[\![x]\!]\ bve\ pbve\ penv & = & pbve[\![x]\!] \\
\mathcal{P}[\![p(e_1,...,e_n)]\!]\ bve\ pbve\ penv & = & \text{let } d_i = \mathcal{E}[\![e_i]\!]\ bve \\
 & & \quad p_i = \mathcal{P}[\![e_i]\!]\ bve\ pbve\ penv \\
 & & \text{in } \mathcal{P}_k[\![p]\!](d_1,...,d_n,p_1,...,p_n) \\
\mathcal{P}[\![f(e_1,...,e_n)]\!]\ bve\ pbve\ penv & = & \text{let } d_i = \mathcal{E}[\![e_i]\!]\ bve \\
 & & \quad p_i = \mathcal{P}[\![e_i]\!]\ bve\ pbve\ penv \\
 & & \text{in } penv[\![f]\!](d_1,...,d_n,p_1,...,p_n)
\end{array}
$$

$$\mathcal{P}_p[\![\{f_i(x_1,...,x_n) = e_i\}]\!] = penv \text{ whererec}$$

$$
\begin{array}{rcl}
penv & = & [(\lambda(y_1,...,y_n,z_1,...,z_n).\ \mathcal{P}[\![e_i]\!]\ [y_k/x_k]\ [z_k/x_k]\ penv)/f_i] \\
env & = & \mathcal{E}_p[\![\{f_i(x_1,...,x_n) = e_i\}]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}_k[\![+]\!] & = & \lambda(x_e,y_e,x_p,y_p).\ x_p : y_p \\
\mathcal{P}_k[\![IF]\!] & = & \lambda(p_e,c_e,a_e,p_p,c_p,a_p).\ p_e \rightarrow p_p : c_p, p_p : a_p
\end{array}
$$

Note that all references to $\mathcal{E}$ in the above semantics could be removed by having path semantics compute the standard semantics directly. In this case, $\mathcal{P}$ would operate on elements in the domain $D \times Path$, and it could be shown that the standard semantics was an abstraction of path semantics.

2.3 First-Order Path Analysis

2.3.1 Domain Issues

The Powerdomain of Paths

The semantics presented in the last section provides path information, but since it relies on the standard semantics it is not computable at compile-time. We must now *abstract* away from path semantics only the path information, leaving behind the information contained in the standard semantics. Ideally, we would like to simply omit the standard semantics, but continue to compute the path information as before. Unfortunately, this is not possible since in the conditional the path result consults the standard result for the value of the predicate before returning the path through the alternate or the consequent. Without the standard information from

the predicate, the path must be approximated with the assumption that either arm could be taken. This initially leads to *two* possible paths through each conditional, and ultimately to a *set* of possible paths through any expression.

The domain $Path$ must now be abstracted to the *powerdomain* of $Path$ to reflect that the information from the standard semantics is no longer available, and the order of evaluation information is contained in a *set* of possible paths. We use the discrete Egli-Milner powerdomain construction, defined as follows [33]:

> For a flat pointed cpo D, the discrete Egli-Milner powerdomain of D, written $\mathbf{P_{EM}}(D)$, consists of the nonempty subsets of D which are either finite or contain bottom, partially ordered as follows: $\forall A, B \in \mathbf{P_{EM}}(D), A \sqsubseteq_{EM} B$ if and only if:
>
> 1. For every $a \in A$, there exists some $b \in B$ such that $a \sqsubseteq_D b$.
> 2. For every $b \in B$, there exists some $a \in A$ such that $a \sqsubseteq_D b$.

The bottom element of $\mathbf{P_{EM}}(Path)$ is the set containing only $\perp_p$, indicating that non-termination is the only possibility, and for two sets A and B, $A \sqsubseteq_{EM} B \iff (\perp_p \in B \Rightarrow \perp_p \in A) \wedge ((A - \{\perp_p\}) \subseteq B)$. We chose the Egli-Milner ordering because it can model *non-termination*, which is an interesting feature of the path model. We could have chosen the Hoare (or "relational") powerdomain construction which orders by set inclusion; on the appropriate domains this would have given equivalent information for terminating paths, but it would not give termination information.

Tuples of Paths

We also need a powerdomain construction for *tuples* of paths, where a set of path tuples of form $(p_1, ..., p_n)$ is formed by taking the cross-product of n elements of $\mathbf{P_{EM}}(Path)$. We assume the usual ordering on tuples:

$$\forall a_i, b_i \in Path, 1 \leq i \leq n, (a_1, ..., a_n) \sqsubseteq (b_1, ..., b_n) \iff \forall i \; a_i \sqsubseteq b_i$$

We define a similar ordering on the powerdomain of path tuples as follows:

$\forall S_i, S_i' \in \mathbf{P_{EM}}(Path), 1 \leq i \leq n,$

$$(S_1 \times ... \times S_n) \sqsubseteq (S_1' \times ... \times S_n') \iff \forall i \; S_i \sqsubseteq S_i'$$

We will use the symbol $\mathbf{P_T}$ for this construction of the powerdomain of tuples.

2.3.2 Semantic Description

Semantic Domains

$Path$,		the domain of paths
$\mathbf{P_{EM}}(Path)$,		the powerdomain of $Path$
$Pfun$	$=$	$\bigcup_{n=1}^{\infty}(\mathbf{P_T}(Path^n) \to \mathbf{P_{EM}}(Path))$,
		the function space mapping paths to paths
$Aenv$	$=$	$Fv \to Pfun$,
		the function environment
Bve	$=$	$Bv \to Path$,
		the bound variable environment

Semantic Functions

$$\begin{aligned} \mathcal{A} &: Exp \to Bve \to Aenv \to \mathbf{P_{EM}}(Path) \\ \mathcal{A}_k &: Pf \to Pfun \\ \mathcal{A}_p &: Prog \to Aenv \end{aligned}$$

$$\begin{aligned} \mathcal{A}_k[\![+]\!] &= \lambda s.\{x : y \mid (x,y) \in s\} \\ \mathcal{A}_k[\![IF]\!] &= \lambda s.\{p : c, p : a \mid (p,c,a) \in s\} \end{aligned}$$

$$\begin{aligned} \mathcal{A}[\![c]\!]\ bve\ aenv &= \{\langle\rangle\} \\ \mathcal{A}[\![x]\!]bve\ aenv &= \{bve[\![x]\!]\} \\ \mathcal{A}[\![p(e_1,...,e_n)]\!]bve\ aenv &= \mathcal{A}_k[\![p]\!](\mathcal{A}[\![e_1]\!]bve\ aenv \times ... \times \mathcal{A}[\![e_n]\!]bve\ aenv) \\ \mathcal{A}[\![f(e_1,...,e_n)]\!]bve\ aenv &= aenv[\![f]\!](\mathcal{A}[\![e_1]\!]bve\ aenv \times ... \times \mathcal{A}[\![e_n]\!]bve\ aenv) \\ \mathcal{A}_p[\![\{f_i(x_1,...,x_n) = e_i\}]\!] &= aenv \text{ whererec} \end{aligned}$$

$$aenv = [(\lambda s. \bigcup\{\mathcal{A}[\![e_i]\!]\ [y_j/x_j]\ aenv \mid (y_1,...,y_n) \in s\})/f_i]$$

Note that the ordering on the arguments to + is still fixed from left-to-right. At this point we could allow both left-to-right and right-to-left orderings, that is,

$$\mathcal{A}_k[\![+]\!] = \lambda s.\{x : y, y : x \mid (x,y) \in s\}.$$

We discuss issues in choosing (or not choosing) an ordering on strict operators in Section 5.1.3, but for now our discussion is simplified by the assumption of a fixed ordering.

Theorem 1 *(Effectiveness)* $\mathcal{A}_p[\![pr]\!]$ *is computable for any finite program* pr.

Proof: Using the standard iterative method for computing Kleene's ascending chain, our first approximation is $aenv^0 = \mathcal{A}_p^0[\![pr]\!] = [(\lambda s.\{\bot_p\})/f_i]$. For each subsequent approximation $aenv^n$, $aenv^n = \mathcal{A}_p^n[\![pr]\!] = (\lambda s.\bigcup\{\mathcal{A}[\![e_i]\!]\ [y_j/x_j]\ aenv | (y_1, ..., y_n) \in s\})/f_i]$. Since all of our domains are finite, we need only show that all operators are monotonic to be guaranteed to arrive at a least upper bound in a finite number of steps. We show this below for the primitive operators $\times$ and $\bigcup$:

1. $\bigcup$: Suppose $x, y, z \in \mathbf{P_{EM}}(Path), x \sqsubseteq y, s_1 = x \cup z, s_2 = y \cup z$. According to the ordering on $\mathbf{P_{EM}}(Path)$, y must contribute every non-$\bot_p$ element to s_2 that x contributes to s_1, and if y contributes $\bot_p$ to s_2, then x must also contribute bot_p to s_1. This implies that $(\bot_p \in s_2 \Rightarrow \bot_p \in s_1) \wedge \forall p \neq \bot_p, p \in s_1 \Rightarrow p \in s_2$. This is precisely the requirement for $s_1 \sqsubseteq s_2$.

2. $\times$: Monotonicity of $\times$ follows directly from the domain definition of $\mathbf{P_T}$, in which one element of $\mathbf{P_T}$ is defined to be weaker than another element precisely when one of the sets from which the cross-product is formed is weaker.□

Theorem 2 *(Safety:) Let* pr, env, $penv$, *and* $aenv$ *be defined as follows:*

$$\begin{aligned} pr \in Prog &= \{f_i(x_1, ..., x_n) = e_i\} \\ env \in Env &= \mathcal{E}_p[\![pr]\!] \\ penv \in Penv &= \mathcal{P}_p[\![pr]\!] \\ aenv \in Aenv &= \mathcal{A}_p[\![pr]\!] \end{aligned}$$

Then $\forall p_1, ..., p_n \in Path, d_1, ..., d_n \in D$,

$$(p_1, ..., p_n) \in s \Rightarrow (penv[\![f_i]\!](d_1, ..., d_n, p_1, ..., p_n) \in aenv[\![f_i]\!]s)$$

Proof: This is easily shown by structural and fixpoint induction. First, *penv* and *aenv* can be shown to be the limits of the chains of $penv^i$ and $aenv^i$ defined below:

$$\begin{array}{lcl} penv^i & = & [(\lambda(a_1,...,a_n,b_1,...,b_n).\mathcal{P}[\![e_i]\!]\ [a_k/x_k]\ env\ [b_k/x_k]\ penv^{i-1})/f_i] \\ aenv^i & = & [(\lambda s.\bigcup\{\mathcal{A}[\![e_i]\!][y_k/x_k]\ aenv^{i-1} \mid (y_1,...,y_n) \in s\})/f_i] \\ penv^0 & = & [\bot_p/f_i] \\ aenv^0 & = & [\{\bot_p\}/f_i] \end{array}$$

Second, consider that $aenv[\![f_i]\!]s = \bigcup\{\mathcal{A}[\![e_i]\!]\ [y_k/x_k]\ aenv \mid (y_1,...,y_n) \in s\}$, and that by assumption, $(p_1,...,p_n) \in s$. Since this clearly implies that $\forall p \in Path, (p \in \mathcal{A}[\![e_i]\!]\ [p_k/x_k]\ aenv) \Rightarrow (p \in \bigcup\{\mathcal{A}[\![e_i]\!]\ [y_j/x_j]\ aenv | (y_1,...,y_n) \in s\})$, and by definition $penv[\![f_i]\!](d_1,...,d_n,p_1,...,p_n) = \mathcal{P}[\![e_i]\!]\ [d_k/x_k]\ env\ [p_k/x_k]\ penv$, we will show the following:

$$(P[\![e_i]\!]\ [d_k/x_k]\ env\ [p_k/x_k]\ penv) \in (\mathcal{A}[\![e_i]\!]\ [p_k/x_k]\ aenv)$$

To proceed by structural induction, we enumerate the cases of e_i:

1. c: $\langle\rangle \in \{\langle\rangle\}$, trivially true.

2. x_i: $p_k \in \{p_k\}$, again trivially true.

3. $+(e_1,e_2)$: Substituting the semantic definitions, we get

$$\mathcal{P}_k[\![+]\!](a_1,a_2,b_1,b_2) \in \mathcal{A}_k[\![+]\!](c_1 \times c_2)$$

$$\begin{array}{lcl} \text{where} \quad a_j & = & \mathcal{E}[\![e_i]\!]\ [d_k/x_k]\ env \\ b_j & = & \mathcal{P}[\![e_i]\!]\ [d_k/x_k]\ env\ [p_k/x_k]\ penv \\ c_j & = & \mathcal{A}[\![e_i]\!]\ [p_k/x_k]\ aenv \end{array}$$

By structural induction, $\forall i, b_i \in c_i$, so again we will consider only the tuple $(b_1,...b_n) \in (c_1 \times ... \times c_n)$. Applying the definitions of $\mathcal{P}_k$ and $\mathcal{A}_k$, we must now show that

$$b_1 : b_2 \in \{b_1 : b_2, b_2 : b_1\},$$

which is trivially true.

4. $IF(e_1, e_2, e_3)$: We follow the method above of extracting the relevant tuple from the cross-product and applying $\mathcal{P}_k$ and $\mathcal{A}_k$ to the primitive IF. Here we must show the condition holds for all possible values of d_i, that is,

$$(b_1 : b_2 \in \{b_1 : b_2, b_1 : b_3\}) \wedge (b_1 : b_3 \in \{b_1 : b_2, b_1 : b_3\}),$$

which again is trivially true.

5. $f(e_1, ..., e_n)$: We must now appeal to fixpoint induction. The base case is simple:

$$(penv^0[\![f]\!](a_1, ..., a_n, b_1, ..., b_n) \in aenv^0[\![f]\!](c_1 \times ... \times c_n)) \Rightarrow (\bot_p \subseteq \{\bot_p\})$$

Using the definitions already established for a_i, b_i, and c_i, but assuming they are evaluated in environments $penv^{i-1}$ and $aenv^{i-1}$, we must now show the following:

$$penv^{i-1}[\![f]\!](a_1, ..., a_n, b_1, ..., b_n) \in aenv^{i-1}[\![f]\!](c_1 \times ... \times c_n)$$

$$\Rightarrow penv^{i}[\![f]\!](a_1, ..., a_n, b_1, ..., b_n) \in aenv^{i}[\![f]\!](c_1 \times ... \times c_n)$$

Applying the techniques used before to extract the appropriate tuple from $c_1 \times ... \times c_n$, we must show that

$$(\mathcal{P}[\![body_f]\!]\ [a_i/x_i]\ env\ [b_i/x_i]\ penv^{i-1}) \in (\mathcal{A}[\![body_f]\!]\ [b_i/x_i]\ aenv^{i-1})$$

$$\Rightarrow (\mathcal{P}[\![body_f]\!]\ [a_i/x_i]\ env\ [b_i/x_i]\ penv^{i}) \in (\mathcal{A}[\![body_f]\!]\ [b_i/x_i]\ aenv^{i})$$

But in applying the definitions of $\mathcal{P}$ and $\mathcal{A}$ the fixpoint induction hypothesis immediately applies, and the implication holds. □

At this point it should be clear that path semantics contains all the information of the standard semantics plus all the information of path analysis. That is, an "abstraction hierarchy" of the three semantics is as shown in Figure 2.1. Keep this diagram in mind; it will be updated in Chapter 4.

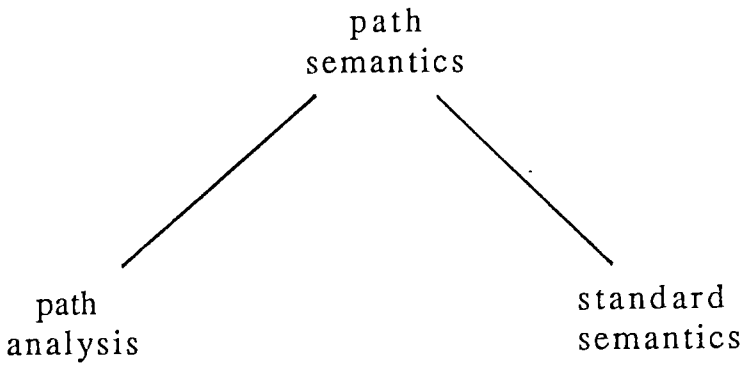


Figure 2.1: Ordering by information content of path semantics, path analysis, and standard semantics

2.3.3 Complexity of Path Analysis

In [21] it was shown that the lower-bound complexity of any method of computing strictness must be exponential in the number of arguments to a function for the general case. In Section 4.2 we show that path analysis subsumes strictness analysis, so the complexity of path analysis must be at least exponential in the number of arguments to a function. However, while in the set-theoretic approach to strictness analysis there are at most 2^n possible strictness sets through an n-ary function, in path analysis there are at most 2^s such sets, where $s = n!+(n-1)!+...+(n-n)!+1$.[1] Although the computation of a tight lower bound on the complexity of path analysis is beyond the scope of this thesis, we conjecture that it is considerably higher than that of strictness analysis.

[1]To see this, recall that in Section 2.1 we saw that there were at most $n!+(n-1)!+...+(n-n)!+1$ paths through an n-ary function. The number of elements in the powerdomain of paths is 2 raised to this power.

2.3.4 Relational vs. Independent Attribute Methods

The function space *Pfun* maps *sets of tuples* of paths to an element in $\mathbf{P}_{\mathbf{EM}}(Path)$, while the reader may have expected it to map *tuples of sets* of paths to an element in $\mathbf{P}_{\mathbf{EM}}(Path)$. Using terminology developed in [29], this is a result of our choosing a *relational* attribute method instead of an *independent* attribute method for associating bound variables with values. In the relational attribute method, the relationship between a binding and its environment is maintained so that two pieces of information from different environments cannot coexist. In the independent attribute method no such provision is made, and bindings from different environments may appear together. This leads to a less precise (although still safe) approximation. A first-order semantics for the independent attribute method appears below.

Semantic Domains

$$
\begin{array}{lcl}
Path, & & \text{the domain of paths} \\
\mathbf{P}_{\mathbf{EM}}(Path), & & \text{the powerdomain of } Path \\
Pfun & = & \bigcup_{n=1}^{\infty}(\mathbf{P}_{\mathbf{EM}}(Path)^n \rightarrow \mathbf{P}_{\mathbf{EM}}(Path)), \\
 & & \text{the function space mapping paths to paths} \\
Aenv & = & Fv \rightarrow Pfun, \\
 & & \text{the function environment} \\
Bve & = & Bv \rightarrow \mathbf{P}_{\mathbf{EM}}(Path), \\
 & & \text{the bound variable environment}
\end{array}
$$

Semantic Functions

$$
\begin{array}{rcl}
\mathcal{A} & : & Exp \rightarrow Bve \rightarrow Aenv \rightarrow \mathbf{P}_{\mathbf{EM}}(Path) \\
\mathcal{A}_k & : & Pf \rightarrow Pfun \\
\mathcal{A}_p & : & Prog \rightarrow Aenv
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{A}_k[\![+]\!] & = & \lambda(x, y).\ x :: y \\
\mathcal{A}_k[\![\text{IF}]\!] & = & \lambda(p, c, a).\ (p :: c) \cup (p :: a)
\end{array}
$$

$$\begin{array}{rcl}
\mathcal{A}[\![c]\!]\ bve\ aenv & = & \{\langle\rangle\} \\
\mathcal{A}[\![x]\!] bve\ aenv & = & bve[\![x]\!] \\
\mathcal{A}[\![p(e_1,...,e_n)]\!] bve\ aenv & = & \mathcal{A}_k[\![p]\!](\mathcal{A}[\![e_1]\!] bve\ aenv,...,\mathcal{A}[\![e_n]\!] bve\ aenv) \\
\mathcal{A}[\![f(e_1,...,e_n)]\!] bve\ aenv & = & aenv[\![f]\!](\mathcal{A}[\![e_1]\!] bve\ aenv,...,\mathcal{A}[\![e_n]\!] bve\ aenv) \\
\mathcal{A}_p[\![\{f_i(x_1,...,x_n) = e_i\}]\!] & = & aenv \text{ whererec} \\
\multicolumn{3}{c}{aenv = [(\lambda(y_1,...,y_n).\mathcal{A}[\![e_i]\!]\ [y_j/x_j]\ aenv)/f_i]}
\end{array}$$

The new "set" path-append operator :: may be defined in terms of the original path-append operator : as follows:

$$\{p_1,...,p_m\} :: \{p_{m+1},...,p_n\} =$$

$$\{(p_1 : p_{m+1}),(p_1 : p_{m+2}),...,(p_1 : p_n),...,(p_m : p_{m+1}),...,(p_m : p_n)\}$$

That the relational attribute method may give more precise information can be seen in this example:

$$\begin{array}{rcl}
f(x,y,z) & = & g(\text{if } x \text{ then } y \text{ else } z); \\
g(a) & = & a + a
\end{array}$$

Using the relational attribute method, g is called with a set of path tuples (each tuple of length one, since g takes only one argument), $\{(\langle x,y\rangle),(\langle x,z\rangle)\}$. The expression $a + a$ is computed in each environment, with a bound to $\langle x,y\rangle$ and with a bound to $\langle x,z\rangle$, and the results are unioned, giving the set $\{\langle x,y\rangle,\langle x,z\rangle\}$. However, in the independent attribute method, g is called with a tuple of sets of paths (again the tuple is of length one), $(\{\langle x,y\rangle,\langle x,z\rangle\})$. Now a can be bound to any member of the set, and we take all possible resulting paths, including the one in which the first occurrence of a is bound to $\langle x,y\rangle$ and the second occurrence is bound to $\langle x,z\rangle$. This results in the set $\{\langle x,y\rangle,\langle x,z\rangle,\langle x,y,z\rangle\}$, which is safe in that it contains all possible paths, but weak in that it contains $\langle x,y,z\rangle$, which clearly can never occur. A comparison of the computation costs of the two methods may be found in [29].

2.3.5 Using a Non-Flat Path Domain

In Section 2.1 it was pointed out that a (non-flat) path domain that distinguishes among non-terminating paths gives more information than a (flat) domain in which

$\perp_p$ represents any non-terminating path. This section explores some of the practical and theoretical implications of basing path analysis on such a non-flat domain.

Define a new domain $Path'$ in which distinct non-terminating paths are distinguished just as terminating paths are in $Path$. Non-terminating paths are delimited by square brackets [ ]; as before, terminating paths are delimited by angle brackets $\langle\rangle$. The elements of $Path'$ are defined as follows:

$$Path' = \{[d_1, ..., d_n]\} \cup \{\langle d_1, ..., d_n\rangle \mid n \geq 0, \forall i, 1 \leq i < n, d_i \in D\}$$

The flat domain construction was appropriate for $Path$ because a terminating path represented a complete computation and thus could not be "improved," while all non-terminating paths, which could be improved, were identified with $\perp_p$. The first condition still holds, but in $Path'$ non-terminating paths are *not* identified. Instead, they are considered distinct paths with varying degrees of "completeness" (and thus varying places in the domain ordering), and are constructed and manipulated much as complete paths are in $Path$. First, the ordering on $Path'$ must be redefined:

$\forall p \in Path, x_i, y_j \in D,$

$$\begin{array}{rcl} p & \sqsubseteq & p \\ [] & \sqsubseteq & p \\ [x_1, ..., x_n] & \sqsubseteq & \langle y_1, ..., y_m\rangle \text{ iff} \\ & & x_1 = y_1 \text{ and } [x_2, ..., x_n] \sqsubseteq \langle y_2, ..., y_m\rangle \\ [x_1, ..., x_n] & \sqsubseteq & [y_1, ..., y_n] \text{ iff} \\ & & [x_1, ..., x_n] \sqsubseteq \langle y_1, ..., y_m\rangle \end{array}$$

The path append operator must be redefined as well:

$$
\begin{array}{rcll}
\langle\rangle : p & = & p & \\
p : \langle\rangle & = & p & \\
[] : p & = & [] & \\
\langle x_1, ..., x_n \rangle : [] & = & [x_1, ..., x_n] & \\
[x_1, ..., x_n] : p & = & [x_1, ..., x_n] & \\
\langle x_1, ..., x_m \rangle : [x_{m+1}, ..., x_n] & = & \langle x_1, ..., x_m \rangle : \langle x_{m+1}, ..., x_n \rangle : [] & \\
\langle x_1, ..., x_m \rangle : \langle x_{m+1}, ..., x_n \rangle & = & \text{if} & x_{m+1} \in \{x_1, ..., x_m\} \\
 & & \text{then} & \langle x_1, ..., x_m \rangle : \langle x_{m+2}, ..., x_n \rangle \\
 & & \text{else} & \langle x_1, ..., x_m, x_{m+1} \rangle : \langle x_{m+2}, ..., x_n \rangle
\end{array}
$$

Theorem 3 *Path' contains strictly more information than Path.*

Proof: Let *Path''* be the domain formed by identifying all incomplete paths in *Path'* with []. The elements and ordering of *Path''* are as follows:

$$Path'' = \{[]\} \cup \{\langle d_1, ..., d_n \rangle \mid n \geq 0, \forall i, 1 \leq i < n, d_i \in D\}$$

$$
\begin{array}{rcl}
p & \sqsubseteq & p \\
[] & \sqsubseteq & p \\
[] & \sqsubseteq & \langle y_1, ..., y_m \rangle \text{ iff} \\
 & & x_1 = y_1 \text{ and } [] \sqsubseteq \langle y_2, ..., y_m \rangle \\
[] & \sqsubseteq & [y_1, ..., y_n] \text{ iff} \\
 & & [] \sqsubseteq \langle y_1, ..., y_m \rangle
\end{array}
$$

It is clear that the third and fourth ordering rules collapse into the second rule, and that by substituting $\perp_p$ for [], the elements and ordering for *Path* are obtained; *Path* can therefore be derived directly from *Path'*, and so contains strictly less information.□

The next step is to construct a powerdomain for *Path'*; again, the operational nature of the Egli-Milner powerdomain makes it a prime candidate. However, the Egli-Milner construction as described before is defined only for flat domains; attempts to apply it directly to non-flat domains lead to problems in ordering and continuity. The *Plotkin* powerdomain is a generalized Egli-Milner powerdomain that can be applied to non-flat domains. The changes to the Egli-Milner construction are as follows:

1. To become $\sqsubseteq_P$, the ordering relation $\sqsubseteq_{EM}$ is redefined in terms of open sets in the Scott-topology. While this is necessary to ensure proper behavior at limit points, $\sqsubseteq_P$ is equivalent to $\sqsubseteq_{EM}$ everywhere else; since our discussion does not rely directly on the domain's behavior at limit points, the reader can think of $\sqsubseteq_P$ as behaving like $\sqsubseteq_{EM}$.

2. The elements of $\mathbf{P_P}(Path')$ are quotiented by the relation $\approx_P$, where $\forall A, B \in \mathbf{P_P}(Path'), A \approx_P B \iff (A \sqsubseteq_P B) \wedge (B \sqsubseteq_P A)$. Without this restriction on the domain elements, $\sqsubseteq_P$ is not a partial order.

Consider the following elements of $\mathbf{P_P}(Path')$:

$$S_1 = \{[x_1], \langle x_1, x_2 \rangle\}$$

$$S_2 = \{[x_1], [x_1, x_2], \langle x_1, x_2 \rangle\}$$

These sets represent different possible computation paths for a function; the first contains only one possible means of not terminating, while the second describes two. Yet these sets are equivalent under $\approx_P$ since they have the same total information content. That is, since both stronger and weaker elements than $[x_1, x_2]$ exist in S_1, the addition of $[x_1, x_2]$ neither increases nor decreases its total information content. Yet in the non-flat model we would like to distinguish between these sets; for example, if these sets represents possible computation paths through a function f, the question, "Can f evaluate x_2 without terminating?" is answered differently by the two sets. Thus in some sense the Plotkin powerdomain construction does not satisfy our intuitive sense of desirable information. Partly for this reason, and partly because of the additional complexity of the powerdomain construction on $Path'$, we chose to work with the flat domain of paths described in Section 2.2.

2.4 Paths of Occurrences

If f's bound variable x appears lexically n times in the definition of f, we say that x has n *occurrences* in f. In the framework of lazy evaluation, the demand for at most one occurrence of x will cause the corresponding expression to be evaluated; subsequent demands to other occurrences of x will return the previously computed value. As described above, path analysis yields information about the order in which the bound variables to a function are evaluated. However, it does *not* tell the order in which the *occurrences* of a particular bound variable are *used*. Yet as we will see in later chapters, this is valuable information. Consider the factorial function:

$$fac(n, acc) = \text{if } (n = 0) \text{ then } acc \text{ else } fac(n - 1, n * acc)$$

Path analysis will find the possible paths through fac to be $\{\langle n, acc\rangle, \perp_p\}$. But this tells nothing about the relative orders in which the three occurrences of n, or the two occurrences of acc, are demanded. Fortunately, once we have the information from path analysis, this "ordering on occurrences" information is easy to compute. For each function f_i, define a new function f_i' that is identical to f_i except that each *occurrence* of a bound variable in f_i becomes a unique bound variable in f_i'. Thus if f_i has j bound variables, each of which has k occurrences, then f_i' will have $j * k$ bound variables, each of which occurs once. The body of f_i' is otherwise identical to the body of f_i — for example, if f_i is recursive, then f_i' will call f_i internally, *not* f_i'. (In fact, f_i' is never called from anywhere; it cannot be substituted for f_i in a call since it has the wrong number of arguments.) The paths through f_i''s bound variables are effectively paths through the *occurrences* of f_i's bound variables.

As an example, consider the "primed" version of the factorial function defined above:

$$fac'(n_1, n_2, n_3, acc_1, acc_2) = \text{if } (n_1 = 0) \text{ then } acc_1 \text{ else } fac(n_2 - 1, n_3 * acc_2)$$

Note that fac' calls fac, not fac'. The paths through fac' are as follows:

$$\{\langle n_1, acc_1\rangle, \langle n_1, n_2, n_3, acc_2\rangle, \bot_p\}$$

Note that removing all but the first occurrence of each bound variable yields the original paths through f. Intuitively, we can say that the paths through f_i characterize f_i's *external* behavior, while the paths through f_i' characterize f_i's *internal* behavior. The semantics for occurrence paths is straightforward as no fixpoint is required; once *aenv* has been computed, giving the basic paths through every function, occurrence paths may be computed in one pass. If we let *OPath* be the domain of occurrence paths, where occurrence paths are defined just like regular paths except that their elements are drawn from the domain of bound variable occurrences, the semantic functions for occurrence paths are defined as follows:

$$\mathcal{O} : exp \rightarrow \mathbf{P_{EM}}(OPath)$$

$$\begin{aligned}
\mathcal{O}[\![c]\!] &= \{\langle\rangle\} \\
\mathcal{O}[\![x_{ij}]\!] &= \{\langle x_{ij}\rangle\} \\
\mathcal{O}[\![p(e_1, ..., e_n)]\!] &= \mathcal{A}_k[\![p]\!](\mathcal{O}[\![e_1]\!] \times ... \times \mathcal{O}[\![e_n]\!]) \\
\mathcal{O}[\![f(e_1, ..., e_n)]\!] &= aenv[\![f]\!](\mathcal{O}[\![e_1]\!] \times ... \times \mathcal{O}[\![e_n]\!]) \\
\mathcal{O}[\![\{f(x_1, ..., x_n) = e_i\}]\!] &= [\mathcal{O}[\![e_i]\!]/f_i]
\end{aligned}$$

For the remainder of this document we will assume that when computing paths for a program we compute paths through all f_i and f_i'; the reader should understand that when we discuss a path through f_i we are referring to order of *evaluation*, while a path through f_i' refers to order of *use*.

2.5 Paths for a Higher-Order Language

Extending path semantics to the higher-order case is straightforward. We extend the domain to contain *path pairs* in the style of strictness pairs developed by Hudak and Young [21]. For each expression e we are interested not only in the path through the bound variables in e, but also in the path that will be taken when e is *applied* to

another expression. This information is represented in a path pair (p_p, p_f), where p_p represents the path and p_f represents the higher-order behavior. Of course, some expressions have no higher-order behavior; for these the higher-order portion of the pair is $perr = \lambda d.\lambda x.(\perp_p, perr)$. The domain of path pairs $PPair$ has $perr$ as its bottom element and is ordered as follows:

$$\forall x, y \in PPair, (x_p, x_f) \sqsubseteq_{PPair} (y_p, y_f)$$

$$\Longleftrightarrow$$

$$(x_p \sqsubseteq_{EM} y_p) \wedge ((x_f\ a) \sqsubseteq_{PPair} (y_f\ a) \forall a \in PPair)$$

As for first-order path semantics, we need an exact higher-order semantics that contains both path and standard information. This semantics is presented below.

2.5.1 Higher-Order Path Semantics

Semantic Domains

$Path$,			domain of paths
$PPair$	$=$	$Path \times (D \rightarrow PPair \rightarrow PPair)$,	"path pairs"
$Henv$	$=$	$(Bv + Fn) \rightarrow Ans$,	the function environment

Semantic Functions

$$\mathcal{H}^p : Exp \rightarrow Env \rightarrow Henv \rightarrow Ans$$

$$
\begin{array}{rcl}
\mathcal{H}^p[\![c]\!]\ env\ henv & = & (\langle\rangle, \mathcal{H}^p_k[\![c]\!]) \\
\mathcal{H}^p[\![x]\!]\ env\ henv & = & henv[\![x]\!] \\
\mathcal{H}^p[\![\lambda x.e]\!]\ env\ henv & = & (\langle\rangle, \lambda d.\lambda p.\ \mathcal{H}^p[\![e]\!]\ env[d/x]\ henv[p/x]) \\
\mathcal{H}^p[\![e_1\ e_2]\!]\ env\ henv & = & \text{let } (p_1, f_1) = \mathcal{H}^p[\![e_1]\!]\ env\ henv \\
 & & \quad (p_2, f_2) = f_1(\mathcal{H}^p[\![e_2]\!]\ env\ henv) \\
 & & \text{in } (p_1 : p_2, f_2) \\
\mathcal{H}^p[\![IF(p,c,a)]\!]\ env\ henv & = & \text{let } d = \mathcal{H}[\![p]\!]\ env \\
 & & \quad (p_1, f_1) = \mathcal{H}^p[\![p]\!]\ env\ henv \\
 & & \quad (p_2, f_2) = \mathcal{H}^p[\![c]\!]\ env\ henv \\
 & & \quad (p_3, f_3) = \mathcal{H}^p[\![a]\!]\ env\ henv \\
 & & \text{in } d \rightarrow (p_1 : p_2, f_2), (p_1 : p_3, f_3) \\
\mathcal{H}^p[\![x+y]\!]\ env\ henv & = & \text{let } (p_1, f_1) = \mathcal{H}^p[\![x]\!]\ env\ henv \\
 & & \quad (p_2, f_2) = \mathcal{H}^p[\![y]\!]\ env\ henv \\
 & & \text{in } (p_1 : p_2, perr)
\end{array}
$$

$$\mathcal{H}^p_p[\![\{f_i = e_i\}]\!] = henv \text{ whererec}$$

$$
\begin{array}{rcl}
henv & = & [(\mathcal{H}^p[\![e_i]\!]\ env\ henv)/f_i] \\
env & = & \mathcal{H}_p[\![\{f_i = e_i\}]\!]
\end{array}
$$

2.5.2 Higher-Order Path Analysis

The abstract higher-order semantics bears the same relationship to the exact higher-order semantics that the abstract first-order semantics bears to the exact first-order semantics. (Note that again we are using the *relational* attribute method.) The path pair domain changes only because its functional component no longer requires an argument in D; the ordering on this new domain of path pairs PP is analogous to the ordering on $PPair$ and the powerdomain of PP is constructed using the Plotkin powerdomain.

Semantic Domains

$$
\begin{array}{lcll}
Path, & & & \text{domain of paths} \\
PP & = & Path \times (PP \rightarrow PP), & \text{“path pairs”} \\
\mathbf{P_P}(PP), & & & \text{the powerdomain of PP} \\
Henv & = & (Bv + Fn) \rightarrow \mathbf{P_P}(PP), & \text{the function environment}
\end{array}
$$

Semantic Functions

$\mathcal{H}^a : Exp \rightarrow Henv \rightarrow \mathbf{P_P}(PP)$

$$
\begin{array}{rcll}
\mathcal{H}^a[\![x_i]\!]\ henv & = & & \{henv[\![x_i]\!]\} \\
\mathcal{H}^a[\![c]\!]\ henv & = & & \{(\langle\rangle, herr)\} \\
\mathcal{H}^a[\![\lambda x.e]\!]\ env\ henv & = & & \{(\langle\rangle, \lambda p.\ \mathcal{H}[\![e]\!]\ henv[p/x]\} \\
\mathcal{H}^a[\![e_1\ e_2]\!]\ henv & = & let & \{x_1, ..., x_m\} = \mathcal{H}^a[\![e_1]\!]\ henv \\
 & & & \{y_1, ..., y_n\} = \mathcal{H}^a[\![e_2]\!]\ henv \\
 & & & \{z_{ij1}, ..., z_{ijl}\} = (x_i^f)(y_j) \\
 & & in & \{\langle x_i^v : z_{ijk}^v, z_{ijk}^f\rangle | i = 1..m, j = 1..n, k = 1..l\} \\
\mathcal{H}^a[\![IF(p,c,a)]\!]\ henv & = & let & \{p_1, ..., p_m\} = \mathcal{H}^a[\![p]\!]\ henv \\
 & & & \{c_1, ..., c_n\} = \mathcal{H}^a[\![c]\!]\ henv \\
 & & & \{a_1, ..., a_l\} = \mathcal{H}^a[\![a]\!]\ henv \\
 & & in & \{\langle p_i^v : c_j^v, c_j^f\rangle, \langle p_i^v : a_k^v, a_k^f\rangle | i = 1..m, j = 1..n, k = 1..l\} \\
\mathcal{H}^a[\![a+b]\!]\ henv & = & let & \{a_1, ..., a_m\} = \mathcal{H}^a[\![a]\!]\ henv \\
 & & & \{b_1, ..., b_n\} = \mathcal{H}^a[\![b]\!]\ henv \\
 & & in & \{\langle a_i^v : b_j^v, herr\rangle | i = 1..m, j = 1..n\} \\
\mathcal{H}_p^a[\![\{f_i\ x_1 ... x_n = e_i\}]\!] & = & henv & \text{whererec} \\
 & & & henv = [(\lambda y_1 ... y_n.\mathcal{H}^a[\![e_i]\!]\ henv[y_j/x_j])/f_i]
\end{array}
$$

Like higher-order strictness analysis, higher-order path analysis is not guaranteed to terminate on every expression in the untyped lambda-calculus. For example, $\mathcal{H}^a[\![(\lambda x.\ x\ x)(\lambda x.\ x\ x)]\!]$ generates path pair of infinite depth. However, this sort of expression is generally considered anomolous and is disallowed under type schemes such as that presented in [13].

Chapter 3

Aggregate Updating

3.1 Overview

The inefficiency stemming from the lack of side-effects in functional languages is perhaps most apparent in *aggregate data structures.*[1] In particular, *arrays* are aggregate data structures that are typically contiguously allocated in memory and in imperative languages they support constant-time access and update. The update is usually performed destructively with an assignment statement, e.g.,

$$a[i] := x.$$

Note that the previous value of $a[i]$ is overwritten by x.

In a functional language, the assignment statement above cannot exist; instead, an *expression* must return the updated array. Furthermore, a cannot be modified, so we cannot write a function to set $a[i]$ to x and return a as its value. Thus functional languages typically provide an *update* function which takes an array a, an index i, and a value x, and returns a new array a' such that $a'[i] = x$, but a is (at least conceptually) unchanged. Since $a[i]$ no longer refers to a memory location but simply to a value, we will use the notation $sel(a, i)$ to mean the i^{th} element of

[1] We will use arrays in examples throughout this section, but the work extends to other aggregate structures.

a. Now suppose $sel(a, i)$ initially has the value v. Then $a' = upd(a, i, x)$ implies that

$$\begin{aligned} sel(a', i) &= x \\ sel(a, i) &= v \\ sel(a', j) &= sel(a, j)\ \forall j \neq i. \end{aligned}$$

Conceptually, a is *copied* into a' except in the i^{th} element, where a' has the value x. Thus the functional semantics is preserved, but the efficiency of the update suffers: we have gone from a single destructive assignment (no new space, constant time), to copying an entire n-element array (space and time proportional to n). To see how quickly this inefficiency can propagate in a program, consider the simple function to initialize an array:

$$init(a, i, x) = \text{if } (i = 0) \text{ then } a \text{ else } init(upd(a, i, x), i - 1, x)$$

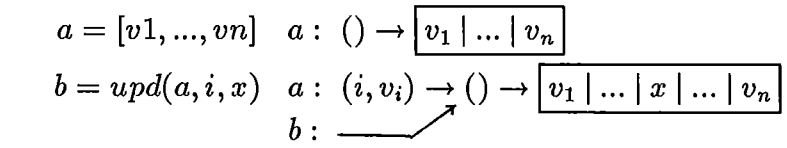
In an imperative language, the call $init(a, n, x)$ would take time $O(n)$ and constant space. But in a functional language using a copying update, it would take time *and* space $O(n^2)$. Clearly, this is unacceptable, yet semantically it is essential that the old version of a remain available. How can we get this functionality but improve the efficiency? In the next two sections we discuss two approaches, *trailer* updating and *destructive* updating. Both approaches initially assume that *upd* is strict in all of its arguments; we discuss individually the consequences of relaxing this assumption.

3.2 Trailers

Instead of copying the entire array, we could *shadow* only the cell being updated. Using this method, an array a is represented as before but is preceded by a list of *trailers*, where a trailer is an $(index, value)$ pair that indicates where an element of a is being shadowed. If no element is shadowed, a is preceded by the empty trailer. To access the value of a's i^{th} element, first the trailer list must be searched to see if that element is shadowed; if so, the value in the trailer is returned, otherwise the i^{th} element of the array is returned. Updating the i^{th} element is more complicated.

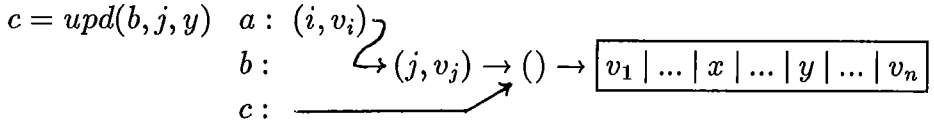
Suppose a is a new array of length n with values $v_1, ..., v_n$, and let b be the result of updating a's i^{th} value to be x. The array portion of a's representation will be *destructively modified* so that its i^{th} element is x, and b will point to this structure with an empty trailer list. Meanwhile, a's trailer list gets the pair (i, v_i), so that accesses to a will see that its i^{th} element is actually v_i. Note that we could have left a alone and constructed b to point to a's array representation with a trailer list containing (i, x); however, it has been observed that the *most recent version* of an array is the most commonly accessed, and we are optimizing for this case by putting the trailer in a's list. The property of referring *only* to the most recent version is sometimes called *single-threadedness.*[32]

The diagram below shows how the representation changes after the update:

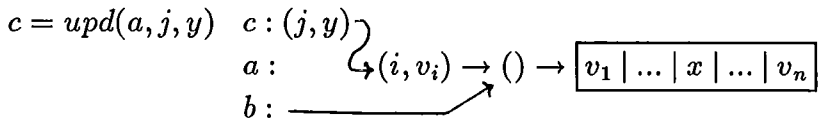


Note that no array ever points directly to the array representation; there is always a trailer list preceeding it, although it may contain only the empty trailer. It is this empty trailer that provides a handle on the references to the array and ensures that these references will see all modifications to the array and the trailer list.

In the example above, if b is subsequently updated, say by $c = upd(b, j, y)$, a similar process takes place: b's empty trailer becomes (j, y), the array is modified destructively, a new empty trailer is inserted in front of the array, and c points to it. Although the array is modified, both a and b see the trailer that retains the old value for $a[j]$. The new representation is as follows:



But suppose a is updated instead of b, in $c = upd(a, j, y)$. There is no way to modify the array destructively and then add a trailer to everyone's list except c's, because c must see all of a's trailer list. So the trailer must appear on c's list, which then points to a's list. This is illustrated below:



In general, when the same logical array (e.g., a) is updated more than once, all but the first update must be done by putting a $(index, newval)$ trailer on the trailer list for the new logical array and leaving the array portion of the representation unchanged. Although this increases access time for the new array, and we have speculated that the new array is the most likely to be referenced in the future, this situation occurs only when that assumption has already failed, i.e., in the presence of non-single-threaded updates. Just as trailer placement was optimized for single-threaded accesses, the update procedure is optimized for single-threaded updates.

We have assumed that *upd* is strict in all of its arguments; to what extent can that assumption be relaxed? If *upd* is lazy in its first argument, the trailer structures will not be built until the updated array is accessed. While this is possible, the runtime expense could be substantial. Similarly, if *upd* is lazy in its second argument the destructive array update would not take place until forced by an access, again contributing runtime expense without clear advantage. If *upd* is lazy in its third argument, a thunk will be stored in the array or trailer, as appropriate, and forced when retrieved. This is a reasonable and not unlikely option.

Using trailers is clearly much more efficient than copying an entire array at each update, but it still introduces substantial time and space overhead. Each update requires two new cells, one to hold the pointer for the new array and one to hold the trailer for the old array. Furthermore, array accesses are slowed by a constant factor, since *even for single-threaded accesses* the trailer list must be checked first.

It may not be obvious to the reader how much overhead this adds to a program, but as the benchmarks in Chapter 6 show, the effect can be substantial. Although a major improvement, trailers do not offer imperative-style efficiency. The next section explores a way to get this efficiency through "behind-the-scenes" destructive updating.

3.3 Destructive Updating

3.3.1 Overview

In the last section we optimized trailers for the single-threaded case, that is, we guessed that the new version of the array would be the most commonly used. But if it could be proven that the old array would never be used again, there would be no need to create the trailer to save the old element; the array could be updated destructively, with the modified array returned as the value of the update. This would clearly be an enormous gain in efficiency over copying and even over trailers, but there are two important questions: How often is it safe to do an update destructively, and how can these cases be detected?

The first question clearly depends on programming style, but empirical evidence indicates that it is overwhelmingly the case that the new version of the array is used in future computation and the old version is discarded. In imperative programming this is the only possibility, since the old version is no longer available once it has been updated. But even in functional programming, where the old version is available, it is rarely used. Consider again the *init* function:

$$init(a, i, x) = \text{if } (i = 0) \text{ then } a \text{ else } init(upd(a, i, x), i - 1, x)$$

In a copying implementation, executing the call $init(A, n, x)$ will produce n copies of partially-initialized versions of A, $n - 1$ of which will be discarded. (Note that when $upd(a, i, x)$ is passed to *init*, the value of a is implicitly discarded since it is

not used anywhere else in the function.) The n^{th} copy contains a fully initialized array, which is returned as the value of the function call. It is likely that the *initial* value of a, that is, A, will never be used again either, having been empty or having contained information that was no longer needed.[2] Therefore even the initial copy was probably unnecessary, and so the entire initialization could be done in place.

The next question is more difficult: how can we detect when it is safe to update an aggregate a destructively? This requires knowing when a is used, or more specifically, whether or not it will be used *again* after it is updated.[3] One possibility is to add temporal language constructs that allow the user to provide information about the lifetimes of aggregates.[4] While this may be a practical approach for some applications, we would like to do as much optimization as possible in the absence of such user hints by *inferring* such temporal information. This sounds suspiciously like the order of evaluation information computed by path analysis in Chapter 2, and indeed it is. However, a restriction and two additional pieces of information are required for update analysis. The restriction is that *upd* must be strict in it arguments. This allows us to treat aggregates as flat structures, that is, structures whose elements are fully defined. The information required starts with information about the update itself: which aggregate is being updated, what lexical occurrence of update is called (since our goal is to convert as many occurrences as possible of *upd* to *destructive upd*), and where the update occurs relative to other elements in its path. Second, we need to know if *aliasing* will interfere with destructive updating, that is, if two variables that appear to refer to different aggregates could in fact be aliases for the same one. The next two sections describe *update semantics* and *update analysis*, which provide the basic information about what is updated and

[2]Note that if the intent is not to throw away A the intent must be to copy it, since every one of its elements is updated. In this case a copy must be made, but it would also have to be made in an imperative language.

[3]In the terminology typical for imperative langauges, we must determine whether a is *live* when it is updated.

[4]Thanks to Alan Perlis for this suggestion.

where. Section 3.3.4 gives examples of applications of update analysis, and shows how we deal with aliasing.

3.3.2 Update Semantics

To derive the update information, paths are extended to *update paths*, where an update path may contain *update elements* in addition to bound variables. An update element is a pair (u, a), where u is the index of the lexical occurrence of *upd* being applied, and a is the aggregate being updated. An update element appears in a path wherever an update occurs in that path. Formally, update paths are defined as follows:

$$Updpath = \{\perp_{upd}\} \cup \{\langle x_1, ..., x_n\rangle | x_i \in Bv + Ue\}$$

where

$$\begin{array}{lcl} Ue & = & \{(upd\_index, bv) | upd\_index \in Nat, bv \in Bv\} \\ Bv & = & \text{the set of bound variables} \end{array}$$

Besides update elements, we also need information about how aggregates are propagated so that we can tell what aggregates could be affected by a call to *upd*. We introduce Agg, the flat domain of aggregates that could be returned by a path, with bottom element $\perp_a$:

$$Agg = \{\perp_a\} + \{none\} + Bv$$

The non-terminating path $\perp_p$ is said to return the aggregate $\perp_a$; a terminating path that cannot return a named aggregate because of type restrictions or anonymity is said to return the aggregate *none*. All other paths return a bound variable that might be an aggregate, and the aggregate associated with one of these paths is that bound variable.

Now we can define update paths that also carry the aggregate information; we call these *update pairs*:

$$Upair = Agg \hat{\times} Updpath$$

The constructor $\hat{\times}$ represents the smash product, essentially a strict cross product. In this case, this implies the following:

$$\forall (a,p) \in Upair, (a = \perp_a) \Leftrightarrow (p = \perp_p)$$

Thus $Upair$ is a flat domain with bottom element $(\perp_a, \perp_p)$.

We say that an update pair $u = (a,p)$ has two components, an aggregate component a and a path component p. We will sometimes write u^a for the aggregate component and u^p for the path component. The semantics for update pairs for the first-order case is given below.

Semantic Domains

$$\begin{array}{lcl} Upair, & & \text{the flat domain of update pairs} \\ Ufun & = & \bigcup_{n=1}^{\infty}(D^n \rightarrow Upair^n \rightarrow Upair) \\ Uenv & = & Fv \rightarrow Ufun, \\ & & \text{the function envronment} \\ Ubve & = & Bv \rightarrow Upair, \\ & & \text{the bound variable environment} \end{array}$$

Semantic Functions

$$\begin{array}{rcl} \mathcal{U} & : & Exp \rightarrow Bve \rightarrow Ubve \rightarrow Uenv \rightarrow Upair \\ \mathcal{U}_k & : & Pf \rightarrow Ufun \\ \mathcal{U}_p & : & Prog \rightarrow Uenv \end{array}$$

$$\begin{array}{rcl} \mathcal{U}[\![c]\!]\ bve\ ubve\ uenv & = & (none, \langle\rangle) \\ \mathcal{U}[\![x]\!]\ bve\ ubve\ uenv & = & ubve[\![x]\!] \\ \mathcal{U}[\![p(e_1,...,e_n)]\!]\ bve\ ubve\ uenv & = & \text{let } d_i = \mathcal{E}[\![e_i]\!]\ bve \\ & & \quad p_i = \mathcal{U}[\![e_i]\!]\ bve\ ubve\ uenv \\ & & \text{in } \mathcal{U}_k[\![p]\!](d_1,...,d_n,p_1,...,p_n) \\ \mathcal{U}[\![f(e_1,...,e_n)]\!]\ bve\ ubve\ uenv & = & \text{let } d_i = \mathcal{E}[\![e_i]\!]\ bve \\ & & \quad p_i = \mathcal{U}[\![e_i]\!]\ bve\ ubve\ uenv \\ & & \text{in } uenv[\![f]\!](d_1,...,d_n,p_1,...,p_n) \end{array}$$

$$\mathcal{U}_p[\![\{f_i(x_1,...,x_n) = e_i\}]\!] = uenv \text{ whererec}$$

$$
\begin{aligned}
uenv &= [(\lambda(y_1,...,y_n,z_1,...,z_n).\ \mathcal{U}[\![e_i]\!]\ [y_i/x_i]\ [z_i/x_i]\ uenv)/f_i] \\
env &= \mathcal{E}_p[\![\{f_i(x_1,...,x_n) = e_i\}]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{U}_k[\![+]\!] &= \lambda(x_e,y_e,x_u,y_u).\ (none, x_u^p : y_u^p) \\
\mathcal{U}_k[\![IF]\!] &= \lambda(p_e,c_e,a_e,p_u,c_u,a_u).\ p_e \rightarrow (c_u^a, p_u^p : c_u^p), (a_u^a, p_u^p : a_u^p) \\
\mathcal{U}_k[\![UPD_j]\!] &= \lambda(a_e,i_e,x_e,a_u,i_u,x_u).\ (none, x_u^p : i_u^p : a_u^p : \langle (j, a_u^a) \rangle) \\
\mathcal{U}_k[\![SEL]\!] &= \lambda(a_e,i_e,a_u,i_u).\ (none, i_u^p : a_u^p)
\end{aligned}
$$

The body of this semantics has the same form as that of path semantics, but the primitives show the additional information being provided. + cannot return an aggregate for type reasons, and so the first member of an aggregate pair returned from + is always *none*. IF can propagate the value returned by either of its arms, and so the appropriate aggregate is that associated with the arm taken. SEL is assumed not to return an aggregate, which means that aggregates cannot be stored inside other aggregates. And although UPD returns an aggregate, it is anonymous and so cannot be shared until it becomes named, e.g., by being passed as a parameter to a function. At that point sharing will be detected inside the function to which it is passed; there is no possibility of its being shared by the function in which it is produced.

Note that the semantics for UPD indicates that its arguments are evaluated from *right to left*, not left to right as the reader may have expected. Like +, UPD could take its arguments in any order, but there is often an advantage to the right-to-left ordering. Similarly, it is often advantageous to evaluate SEL from right to left. This and other issues in choosing an ordering for primitives are discussed in Section 5.1.3.

As for path semantics, all references to $\mathcal{E}$ could be eliminated by incorporating the standard semantics directly into update semantics. Furthermore, it should be clear that update semantics computes strictly more information than path semantics, since the path portion of update semantics is precisely equivalent to path semantics.

3.3.3 Update Analysis

Like path semantics, update semantics is not useful for static program optimization since it relies on the standard semantics. However, update semantics can be abstracted to update *analysis* in a manner similar to that in which path semantics was abstracted to path analysis. Again, the conditional holds the key, since this is where update semantics relies on the standard semantics. Following the form of path analysis, we see that if we return a set of update pairs instead of a single update pair, we can perform the abstraction. The form of each update pair does not change, but we now operate on the powerdomain of update pairs, again choosing the Egli-Milner powerdomain construction.

Semantic Domains

$$
\begin{array}{lcll}
Upair, & & & \text{the flat domain of update pairs} \\
\mathbf{P_{EM}}(Upair), & & & \text{the powerdomain of Upair} \\
\hat{Ufun} & = & \bigcup_{n=1}^{\infty}(\mathbf{P_T}(Upair^n) \to \mathbf{P_{EM}}(Upair)) & \\
\hat{Uenv} & = & Fv \to \hat{Ufun}, & \text{the function envronment} \\
\hat{Ubve} & = & Bv \to Upair, & \text{the bound variable environment}
\end{array}
$$

Semantic Functions

$$
\begin{array}{rcl}
\hat{\mathcal{U}} & : & Exp \to \hat{Ubve} \to \hat{Uenv} \to \mathbf{P_{EM}}(Upair) \\
\hat{\mathcal{U}}_k & : & Pf \to \hat{Ufun} \\
\hat{\mathcal{U}}_p & : & Prog \to \hat{Uenv}
\end{array}
$$

$$
\begin{array}{rcl}
\hat{\mathcal{U}}[\![c]\!]\ bve\ aenv & = & \{\langle\rangle\} \\
\hat{\mathcal{U}}[\![x]\!]bve\ aenv & = & \{bve[\![x]\!]\} \\
\hat{\mathcal{U}}[\![p(e_1,...,e_n)]\!]bve\ aenv & = & \hat{\mathcal{U}}_k[\![p]\!](\hat{\mathcal{U}}[\![e_1]\!]bve\ aenv \times ... \times \hat{\mathcal{U}}[\![e_n]\!]bve\ aenv) \\
\hat{\mathcal{U}}[\![f(e_1,...,e_n)]\!]bve\ aenv & = & aenv[\![f]\!](\hat{\mathcal{U}}[\![e_1]\!]bve\ aenv \times ... \times \hat{\mathcal{U}}[\![e_n]\!]bve\ aenv) \\
\hat{\mathcal{U}}_p[\![\{f_i(x_1,...,x_n) = e_i\}]\!] & = & aenv \text{ whererec}
\end{array}
$$

$$aenv = [(\lambda s. \bigcup\{\hat{\mathcal{U}}[\![e_i]\!]\ [y_j/x_j]\ aenv \mid (y_1,...,y_n) \in s\})/f_i]$$

$$\begin{aligned}
\hat{\mathcal{U}}_k[\![+]\!] &= \lambda s.\{(none, x^p : y^p) \mid (x, y) \in s\} \\
\hat{\mathcal{U}}_k[\![IF]\!] &= \lambda s.\{(c^a, p^p : c^p), (a^a, p^p : a^p) \mid (p, c, a) \in s\} \\
\hat{\mathcal{U}}_k[\![UPD_j]\!] &= \lambda s.\{(none, x^p : i^p : a^p : \langle (j, a^a) \rangle) \mid (a, i, x) \in s\} \\
\hat{\mathcal{U}}_k[\![SEL]\!] &= \lambda s.\{(none, y^p : x^p) \mid (x, y) \in s\}
\end{aligned}$$

Theorem 4 *$\hat{\mathcal{U}}_p[\![pr]\!]$ is computable for any finite program pr.*

Proof: The proof follows the same form as that for path analysis, and depends on showing that the domains are finite and the operations are monotonic. We have already shown the domain of paths to be finite, and clearly *Agg* is finite, so *Upair* must be finite also. We must still shown monotonicity of $\times$ and $\bigcup$ on domain $\mathbf{P}_{\mathbf{EM}}(Upair)$, but this follows directly from the construction in Section 2.3.2 for $\mathbf{P}_{\mathbf{EM}}(Path)$, and the existence of a least fixpoint is guaranteed. □

3.3.4 Applying Update Analysis

Update analysis now seems to contain the information required to detect when destructive aggregate updating is safe. The method is simple: Compute the set of update pairs for each function in a program, and for the occurrence version of each function. Then look at the update paths through the occurrence functions: if in any path in which an update element (upd_i, a) occurs there is later another occurrence of a, then upd_i cannot be done destructively. (In this discussion we will use the notation (upd_i, a) instead of (i, a) as it is easier to read. Also, often integer or unspecified values (e.g., indices or values to be stored in an array) are represented by a single argument, usually either i or j. Although this makes some of the functions rather trivial, it simplifies the presentation and has no effect on the update analysis.) Consider once again the *init* example:

$$init(a, i, x) = \text{if } i = 0 \text{ then } a \text{ else } init(upd(a, i, x), i - 1, x)$$

Recalling that *upd* evaluates its arguments from right-to-left, and discarding the aggregate portion of the final update pair, the update paths through *init* are:

$$\{\bot_p, \langle i, a\rangle, \langle i, x, a, (upd\ a)\rangle\}$$

But it is the *occurrence paths*, or paths through *init'*, that indicate whether or not an update can be done destructively. Numbering the occurrences of each bound variable lexically from left to right, we get *init'* and its paths:

$$init'(a_1, a_2, i_1, i_2, i_3, x_1, x_2) = \text{if } i_1 = 0 \text{ then } a_1 \text{ else } init(upd_1(a_2, i_2, x_1), i_3 - 1, x_2)$$

$$\{\bot_p, \langle i_1, a_1\rangle, \langle i_1, i_3, x_1, i_2, a_2, (upd_1, a_2)\rangle, \langle i_1, i_3, x_2, x_1, i_2, a_2, (upd_1, a_2)\rangle\}$$

In both of the paths that contain update elements, the aggregate being updated is not used again after the update, so upd_1 can be done destructively.

Another example shows how the effect of updating in one function can be accounted for in another function:

$$\begin{array}{rcl} g(a,b,i) & = & \text{if } i = 0 \text{ then } b \text{ else } upd(a,i,i) \\ f(x,y,j) & = & sel(g(x,y,j),j) + sel(y,j) \end{array} \qquad (1)$$

The paths and occurrence paths are shown below:

$$\begin{array}{rl} g\_paths: & \{\langle i, b\rangle, \langle i, a, (upd_1, a)\rangle\} \\ f\_paths: & \{\langle j, y\rangle, \langle j, x, (upd_1, x), y\rangle\} \\ g'\_paths: & \{\langle i_1, b\rangle, \langle i_1, i_3, i_2, a, (upd_1, a)\rangle\} \\ f'\_paths: & \{\langle j_2, j_1, y_1, j_3, y_2\rangle, \langle j_2, j_1, x, (upd_1, x), j_3, y_2\rangle\} \end{array}$$

Note that upd_1 appears not only in the paths through g, but also in the paths through f, as the update information in g is "exported" into every function that uses it. Again, the occurrence paths through f' and g' show that upd_1 can be done destructively.

While it is instructive to consider examples in which destructive updating is possible, it is even more instructive to consider examples in which it is *not* possible. Failure to catch a potential optimization is disappointing, but performing an unsafe

optimization renders the entire analysis useless. The next few examples examine ways in which it can be unsafe to update destructively, and show how update analysis catches these cases. First, take the simplest case:

$$f(a,i) = sel(upd_1(a,i,i),i) + sel(a,i)$$

The regular paths and occurrence paths for f appear below:

$$\begin{array}{ll} f\text{-}paths: & \{\langle i,a,(upd_1,a)\rangle\} \\ f'\text{-}paths: & \{\langle i_3,i_2,i_1,a_1,(upd_1,a_1),i_4,a_2\rangle\} \end{array}$$

Since an occurrence of a (a_2) is used after another occurrence of a (a_1) is updated, the path through f' indicates that upd_1 cannot be done destructively.

Next, consider a slight modification of the functions (1) above:

$$\begin{array}{rcll} g(a,b,i) & = & \text{if } i=0 \text{ then } a \text{ else } upd(b,i,i) & (2) \\ f(x,y,j) & = & sel(g(x,y,j),j) + sel(y,j) & \end{array}$$

$$\begin{array}{ll} g\text{-}paths: & \{\langle i,a\rangle,\langle i,b,(upd_1,b)\rangle\} \\ f\text{-}paths: & \{\langle j,x,y\rangle,\langle j,y,(upd_1,y)\rangle\} \\ g'\text{-}paths: & \{\langle i_1,a\rangle,\langle i_1,i_2,i_3,b,(upd_1,b)\rangle\} \\ f'\text{-}paths: & \{\langle j_2,j_1,x,j_3,y_2\rangle,\langle j_2,j_1,y_1,(upd_1,y_1),j_3,y_2\rangle \end{array}$$

Now the paths through f' indicate that upd_1 cannot be done destructively, although it should be noted that the paths through g' do not show this. This emphasizes that a system of functions must be analyzed *as a whole.*

Now consider a third example:

$$f(a,b) = sel(upd_1(a,i_1,x),i_2) + sel(b,i_3)$$

$$g(c) = f(c,c)$$

The appropriate paths are as follows:

$$\begin{array}{ll} f\text{-}paths: & \{\langle a,(upd_1,a),b\rangle\} \\ f'\text{-}paths: & \{\langle a_1,(upd_1,a_1),b_1\rangle\} \\ g\text{-}paths: & \{\langle c,(upd_1,c)\rangle\} \\ g\text{-}paths: & \{\langle c_1,(upd_1,c_2)\rangle\} \end{array}$$

None of these paths indicates that upd_1 cannot be done destructively, yet this is clearly the case. The problem is that a and b are *aliases* for c, and although the conflict really occurs inside of f, it can't be detected by examining only f and the functions it relies on; instead, information is required about the functions that *use* f. Note that this is different from the situation that arose in functions (2) above, where f could detect its conflict because g exported its update information to f. In that case, the information flow from callee to caller was sufficient; in this case, we need information flow from caller to callee as well.

We accomplish this by doing a simple transitive closure of the bound variables that are passed as arguments, thus statically detecting all possible cases of aliasing. This is admittedly a very operational appraoch, and it is safe only for the first-order case; a full higher-order analysis would require a *collecting interpretation*[20], a formal denotational description of how the meaning of an expression can depend on its context. However, our limited treatment of higher-order constructs, described in Section 5.1.4, permits us to use the simpler transitive closure.

Using this technique, each function is associated with a set of aggregate tuples with which that function might be called. If we substitute those tuples in for the appropriate bound variables in the occurrence paths through the function, we can see the effect of aliasing. Going back to the last example, we find that f is called with the argument tuple (c, c), and so we substitute c for each occurrence of a in f's occurrence paths, and also for each occurrence of b. The resulting path looks like this:

$$new\_f'\_paths : \{\langle c_1, (upd_1, c_1), c_1\rangle\}$$

The effect of the aliasing is now clear, and the path shows that upd_1 cannot be done destructively because of a conflict in f.

3.3.5 Cleaning Up With Trailers

No matter how good our update analysis is, there will be times when it cannot detect that an update can be done destructively. This might be caused by information lost in our abstraction, or a copy might truly be the programmer's intent. In either case, we would like to use trailers whenever possible instead of copying the entire array. Combining the two approaches involves some subtleties, and in this section we discuss some of the issues that arise.

The central issue is that we assume that a call to *upd* returns an array with *no other references to it.* This is manifested in the definition of update pairs, where the aggregate element of the pair for a call to *upd* is *none.* In an all-copying or all-destructive implementation this assumption is valid, but trailers introduce *sharing* that must be taken into account. When an array is updated with a trailer, there are two references to that array, both of which are preceded by trailer lists. If the array is updated destructively through either reference and no modification is made to the trailer lists, *both* references will see the effect of that update. Since the trailer was used in the first place because our analysis indicated that either reference could be used again, this is unacceptable. Thus once an array is updated with trailers, *all subsequent updates to that array must be done with trailers also.* This implies that we must be able to tell whether an array has been updated with a trailer. This is easy, since destructive updating and trailer updating suggest different representations — a plain array and an array preceded by a trailer list, respectively. However, there is some overhead in checking to see what representation is being used at each update and select. To avoid penalizing performance in programs where destructive

updating is universally possible, we propose the following constructs:

dupd:	destructive update
tupd:	update with trailer, changing representation if necessary
cupd:	udpate carefully: if array has trailer representation use trailer, otherwise update destructively
dsel:	select straight from array, no trailers
csel:	check for trailer representation, do appropriate select

In a program in which all updates are found to be destructive, *dupd* is used for every update and *dsel* for every select. However, when some updates are destructive and others are not, *dupd* is no longer safe; *cupd* must be used to ensure that a trailer-update is not followed by a destructive update. Furthermore, *csel* must be used to ensure that selection is done for the appropriate representation.

An alternative to this is to do a *reaching analysis* to determine which updates and selects can be reached by a non-destructive update. This analysis can be arbitrarily complex, but even in its most basic form it must include a collecting interpretation. We conjecture that the occurrence of non-destructive updates is sufficiently rare that they can be dealt with effectively either through the method described above or by copying at the necessary points.

Chapter 4

Other Applications of Path Analysis

4.1 Overview

Although the development of path analysis was motivated by the aggregate update problem, the information obtained by path analysis may be applied to other optimizations as well. These optimizations will be discussed in detail in this chapter, but in order to understand (and even anticipate) them it is important to keep in mind some characteristics of path analysis. In particular:

1. Path analysis reveals the possible orders in which the bound variables to a function might be evaluated; that is, it describes the *order of evaluation behavior* of each function.

2. Path analysis can be extended to reveal the possible orders in which distinct occurrences of a particular bound variable are *used.* (Recall that in lazy evaluation an expression is *evaluated* at most once.)

3. For both of the above analyses, every possible path will be found; however, since compile-time information is inexact, some paths that will not be taken, and perhaps even some that could never be taken, will be found as well.

4. Path analysis models a sequential system; this is discussed further in Chapter 7, but for the purpose of this chapter we will discuss only applications for sequential systems.

The third item listed above has a particularly strong impact on the types of optimizations for which path analysis is suitable. Since a *superset* of the possible paths is computed, "for all" questions are easily answered; any condition that holds for all of the paths found by path analysis will also hold for all of the actual paths. However, "there exists" questions cannot be answered, as the path that satisfies that existence criterion might be an extraneous path, one that could never occur.

In this chapter two additional applications of path analysis are presented.

4.2 Strictness Analysis

4.2.1 Definition

In lazy evaluation, the arguments to a function are evaluated if and when their values are demanded. However, it is often the case that it can be determined statically that a given function will always evaluate one or more of its arguments, in which case that function is said to be *strict* in those arguments. A function f is strict in its i^{th} argument x if f is always non-terminating when x is non-terminating; formally,

$$f(e_1, ..., e_{i-1}, \perp, e_{i+1}, ..., e_n) = \perp \quad \forall e_j, j \neq i$$

The value of strictness analysis lies in the fact that call-by-value is typically more efficient than lazy evaluation, and that call-by-value and lazy evaluation are guaranteed to give the same result *if they both give a result.* That is, semantically the only difference between them is that call-by-value may fail to terminate in cases where lazy evaluation will terminate, and so lazy evaluation is more expressive in that it produces results for a larger set of inputs. However, if we can determine

that a function is strict in its argument, then it is safe to evaluate that argument at the time of the function call, that is, to pass it in by value, since its failure to terminate will cause the function to not terminate in any case. Strictness has been widely studied [27,7,21], but the approach taken in this chapter is quite different from any we have seen.

4.2.2 Applying Path Analysis

Once we have computed the set of all possible paths through a function f, computing the strictness properties of f is straightforward. A function f is strict in its i^{th} argument x_i if and only if x_i appears in every terminating path through f. That is,

$$\forall p \in aenv[\![f]\!]\{(\langle x_1\rangle, ..., \langle x_n\rangle)\}, (x_i \in p) \vee (p = \bot_p)$$

For example, consider the following "wrapped-up conditional" function:

$$f(x, y, z) = \text{if } x \text{ then } y \text{ else } z$$

The paths through f are $\{\langle x, y\rangle, \langle x, z\rangle, \bot_p\}$. f is strict in x, as it appears in both terminating paths, but not in y (since it is not in $\langle x, z\rangle$) or z (not in $\langle x, y\rangle$).

It is clear that we can do *some* strictness analysis with paths, but we would like to show that we can do a *good* analysis. Specifically, we show here that we do precisely the same analysis that Hudak and Young do in their first-order work [21]. More formally:

Let *senv* be the strictness environment found by Hudak and Young's first-order strictness analysis; *senv* takes a function variable f_i and a list of sets $(s_1, ..., s_n)$, where s_j represents the set of variables in which f_i's j^{th} argument is strict, and returns the strictness properties of f_i (that is, those variables in which f_i is strict) in terms of $(s_1, ..., s_n)$. Then

$$x \in senv[\![f_i]\!](s_1, ..., s_n) \iff x \in F(aenv[\![f_i]\!]\{(\langle y_1\rangle, ..., \langle y_n\rangle)\})(y_1, ..., y_n)(s_1, ..., s_n)$$

where F is defined as follows:

$$F\{p_1, ..., p_m\}(y_1, ..., y_n)(s_1, ..., s_n) = p'_1 \cap p'_2 \cap ... \cap p'_m$$

$$\text{where } p'_i = \bigcup\{s_j \mid y_j \in p_i\}$$

Thus F takes a set of paths, a list of the elements from which those paths are composed, and a list of the sets passed to the strictness environment and "translates" from the path model to the strictness model.

As an example, again let $f(x, y, z) =$ if x then y else z . Then using Hudak and Young strictness, we get:

$$senv[\![f]\!](s_1, s_2, s_3) = s_1 \cup (s_2 \cap s_3)$$

Using paths, we get:

let

$$P = aenv[\![f]\!]\{(\langle y_1\rangle, \langle y_2\rangle, \langle y_3\rangle)\} = \{\langle y_1, y_2\rangle, \langle y_1, y_3\rangle\}$$

then

$$F(P)(y_1, ..., y_n)(s_1, ..., s_n) = (s_1 \cup s_2) \cap (s_1 \cup s_3)$$

$$= s_1 \cup (s_2 \cap s_3)$$

In fact, this equality holds in general.

Theorem 5 *Let "senv" be the first-order strictness environment found by Hudak and Young's analysis of [21], "aenv" the path environment defined in our Section 2.3, and the function "F" defined as above. Then*

$$x \in senv[\![f_i]\!](s_1, ..., s_n) \iff x \in F(aenv[\![f_i]\!]\{(\langle y_1\rangle, ..., \langle y_n\rangle)\})(y_1, ..., y_n)(s_1, ..., s_n)$$

Proof: Appendix A.

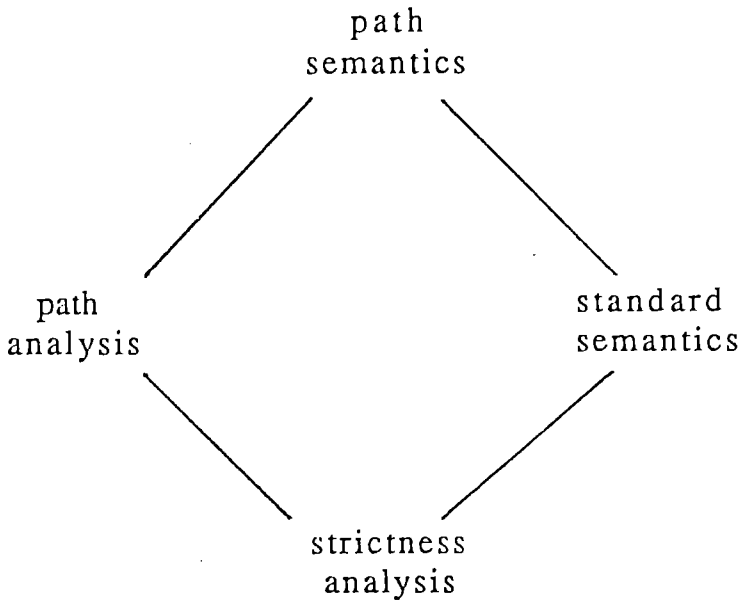


Figure 4.1: Information ordering on four semantics

At this point we would like to update (as promised) the diagram in Figure 2.1 that shows the relative information content of path semantics, path analysis, and the standard semantics. As shown in Figure 4.1, we can now add strictness analysis to the diagram; we have just shown that the information obtained in strictness analysis is also contained in path analysis, and it has been shown elsewhere [21,27] that strictness analysis is also an abstraction of the standard semantics. It is interesting that although path analysis was obtained by "removing" the standard semantics from path semantics, both the standard semantics and path analysis retain the information contained in strictness analysis.

4.3 Optimizing Thunks

4.3.1 Overview

As discussed in the last section, strictness analysis is an important optimization for lazy evaluation. However, not all functions will be found to be strict in all of their arguments; certainly some functions do not always require all of their arguments, and even when they do, it will sometimes be the case that a compile-time analysis, operating with imperfect information, will not detect it. Thus in spite of strictness analysis, the overhead of lazy evalation remains in some circumstances. This overhead comes both with creating the thunk and with accessing it; although creating it is unavoidable, accessing it may be optimized, as discussed below.

The basic access mechanism in lazy evaluation involves checking to see whether or not the expression has been evaluated, then evaluating it if necessary, or returning the previously stored result otherwise. Making this check at each access can be expensive, for although the check is simple, variable access occurs often. But if we could determine at compile-time which occurrence of a bound variable would force evaluation and which occurrences would return a stored value, the check could be eliminated. Note that this is fundamentally different from strictness analysis; finding that f is strict in x_i means that some occurrence of x_i will always be demanded, while finding that the j_{th} occurrence of x_i always forces evaluation means that *if* x_{ij} is demanded, no other occurrence of x_i will have been demanded before it. The fundamental issue is *order of use* which, as discussed in Chapter 3, is easily derived from the order of evaluation information obtained using path analysis.

4.3.2 The Costs of Using Thunks

The costs associated with creating and maintaining thunks are as follows:

1. Space complexity:

(a) Space for the *environment* (i.e., bindings for free variables).

(b) A cell to hold the computed value for later use.

(c) Tag bits to indicate the status of the evaluation

2. Time complexity:

(a) The time required to create the thunk.

(b) The time required to invoke ("force") the thunk.

(c) The time required to update the thunk with the computed value.

(d) The time required to test the status of a thunk.

Potential for optimization is clear here. For example, if we can determine at compile-time that an occurrence of a bound variable will always (or never) force evaluation, the status test (item 2d) can be eliminated. If we can determine that x will never be used again after it is evaluated, there is no need to store its value. It turns out that there are a wide range of such optimizations, but their availability depends on the way in which thunks are modeled. In the next section we discuss different ways of implementing thunks and the optimizations that are available for them.

4.3.3 Representing Thunks

Two basic constructs are required to implement lazy evaluation: `DELAY`, which takes an expression and "wraps it up" into a thunk for later evaluation, and `FORCE`, which determines the value of a thunk. However, the way in which `DELAY` and `FORCE` are implemented can greatly affect their potential for optimization. We now present four different methods, or *modes*, for delaying and accessing values — closure mode (CL), cell mode (C), optimized cell mode (CO), and value mode (V) — and discuss

various optimizations for each. We distinguish between different implementations of DELAY and FORCE by subscripting by the mode, e.g. $\texttt{DELAY}_{CL}$ or $\texttt{FORCE}_{CO}$.

In the following sections, examples of target code are written in Scheme, and appear in typewriter font, e.g. `(car x)`, while examples of source code appear as usual in our generic functional language in italics.

4.3.4 The Closure Mode (*CL*)

The most obvious way to delay the evaluation of an expression is to enclose it within a parameterless procedure:

```
(DELAY exp) ⇒ (lambda () exp)
```

In this case forcing a thunk is simply a function call:

```
(FORCE exp) ⇒ (exp)
```

Of course, this doesn't "cache" the computed value, as required by lazy evaluation. To store the value, and return it upon subsequent demands, we simply do the appropriate check and storage operations within the procedure:

```
(DELAY_CL exp) ⇒ (let ((done '#F)
                       (val nil))
                   (lambda ()
                     (cond (done val)
                           (else (set! val exp)
                                 (set! done '#T)
                                 val))))
```

The FORCE operation is unchanged:

```
(FORCE_CL exp) ⇒ (exp)
```

The subscript "*CL*" emphasizes that this is a *closure*. (This representation of a thunk might aptly be called a "self-modifying thunk.")

This mode may be optimized in special cases. Suppose that *exp* is either a constant or an expression that we know will be evaluated at most once; then caching the value is superfluous, since in the former case there is no evaluation to be done, and in the latter case there is no need to retain the computed value. Thus in both cases we can simplify the implementation to:

$$\texttt{(TRIVIAL-DELAY}_{CL}\ \texttt{exp)} \Rightarrow \texttt{(lambda () exp)}$$

A less obvious optimization involves the "passing along" of bound variables. Consider the following function:

$$f(x, y) = \text{if } y \text{ then } 0 \text{ else } x + g(x)$$

In general f will create a thunk for the argument it passes to g. But that argument is just x, which was passed delayed to f, and so must be forced to get its value. So the call to g is implemented as:

$$\texttt{((FORCE}_{CL}\ \texttt{g) (DELAY}_{CL}\ \texttt{(FORCE}_{CL}\ \texttt{x)))}$$

However, it seems redundant to delay the force of an already delayed object, so we perform the following optimization:

$$\texttt{(DELAY}_{CL}\ \texttt{(FORCE}_{CL}\ \texttt{x))} \Rightarrow \texttt{x}$$

Thus we have $\texttt{((FORCE}_{CL}\ \texttt{g) x)}$ for the call to g.

Despite these optimizations, closure mode has two significant disadvantages. First, every strict reference to a bound variable x requires an *unknown procedure call*, that is, a call to the parameterless procedure representing the thunk bound to x. This is true even if x has already been evaluated and we are just returning

a stored value. This call becomes expensive when variables are referenced often, as even very efficient procedure call mechanisms do not reduce the cost of context switching.

The second disadvantage is that most of the order of evaluation optimizations cannot be used. The reason is that the mechanism that decides whether to evaluate the expression or return the stored value is in `DELAY`, but the information used for the optimizations is available only when the expression is forced. Although occasionally we may be able to perform optimizations when we *create* the thunk (e.g., the `TRIVIAL-DELAY` optimization), a thunk is typically forced in several different places, and we would like to optimize each place separately. The next mode allows such optimizations.

4.3.5 The Cell Mode (C)

In this mode a thunk is not represented as a function but as a pair, whose first element is a boolean flag indicating the status of the second element: if true, the second element contains a value; if false, it contains a parameterless procedure which will return the value when called:

```
(DELAY_C exp) ⇒ (cons '#F (lambda () exp))
(FORCE_C exp) ⇒ (if (car exp)
                    (cdr exp)
                    (let ((v ((cdr exp)) ))
                       (set! (cdr exp) v)
                       (set! (car exp) '#T)
                       v))
```

Using this mode we can perform optimizations analogous to those for the closure mode:

```
(TRIVIAL-DELAY_C exp) ⇒ (CONS '#T exp)          (1)
(DELAY_C (FORCE_C exp)) ⇒ exp
```

Note, however, that `(TRIVIAL-DELAY`$_C$ `exp)` does not delay evaluation of `exp` (as opposed to `(TRIVIAL-DELAY`$_{CL}$ `exp)`, which delayed `exp` but did not cache its value once evaluated), and so may be used with constant expressions but not with expressions that are evaluated at most once.

Now the disadvantages of the closure mode have gone away: accessing a variable no longer requires an *unknown* function call (`IF`, `CAR`, and `CDR` are known functions) if the variable has already been evaluated, and the work of status-checking and caching the computed value is now done in `FORCE`.

To see how we can take advantage of order of evaluation information, suppose an occurrence of a bound variable x is *never* the first to be demanded, that is, we know that x has been evaluated before this demand. Then when we force it we no longer need to check whether it has been evaluated, and can simply return its value directly:[1]

```
(FORCE_C exp) ⇒ (cdr exp)
```

What if we know that a given occurrence of x is *always the first* to be demanded; that is, it will always force evaluation of the delayed expression? It seems that the following optimization is possible:

```
(FORCE_C exp) ⇒ (let ((v ((cdr exp)) ))          (2)
                    (set! (cdr exp) v)
                    (set! (car exp) '#T)
                    v))
```

[1]Simple (local) versions of this optimization have been used in other compilers, for example the Lazy ML compiler [24].

Unfortunately, this optimization is not always safe when combined with the other optimizations discussed above (1). To see why, consider the function f defined earlier:

$$f(x, y) = \text{if } y \text{ then } 0 \text{ else } x + g(x)$$

Suppose that + evaluates its arguments left-to-right — first x is evaluated, then (because we optimize (DELAY (FORCE x)) ⇒ x) g is called with a thunk which has already been evaluated, that is, whose car is '#T. But if g uses the new optimization (2) for the first access of its argument, then it will try to "call" the cdr of the thunk, which is a value! The interaction arises because g assumes that its arguments are unevaluated, but because of the optimizations at (1) this may not be the case. Of course, we could do a global analysis to determine which functions always get their arguments unevaluated, and use this optimization in those functions only; this possibility should be weighed against (or possibly combined with) the next mode.

4.3.6 The Optimized Cell Mode (*CO*)

This mode is just like cell mode except that a function's arguments are guaranteed to be unevaluated on entry to that function. That is,

$$\begin{aligned} \texttt{DELAY}_{CO} &= \texttt{DELAY}_{C} \\ \texttt{FORCE}_{CO} &= \texttt{FORCE}_{C} \end{aligned}$$

The difference is that both of the optimizations described for the first two modes are disallowed:

$$\texttt{(DELAY}_{CO}\ \texttt{(FORCE}_{CO}\ \texttt{exp))} \not\Rightarrow \texttt{exp}$$
$$\texttt{(TRIVIAL-DELAY}_{CO}\ \texttt{exp)} \not\Rightarrow \texttt{(DELAY}_{CO}\ \texttt{exp)}$$

However, the opportunities for using path information increase greatly, and may outweigh the loss of these two optimizations.

There are three basic pieces of information that are useful in optimizing FORCE

for a particular occurrence of a variable x. This information, which is provided by path analysis, can be described as determining for certain that x:

- Has previously been evaluated (i.e., is this never the first occurrence of x?).
- Has never been evaluated (i.e., is this always the first occurrence of x?).
- Will never be used again (i.e., is this always the last occurrence of x?).

If one of the first two conditions holds the status check may be eliminated, and if the third condition holds the computed value need not be cached. The following table shows how $\texttt{FORCE}_{CO}$ may be optimized under the resulting six combinations of compile-time data:

	Evaluated?	Unevaluated?	Unknown

Last?	x has already been evaluated:
	`(cdr x)`
		This is the only demand for x:
		`((cdr x))`
			x will never be demanded again:
			`(if (car x)`
			`  (cdr x)`
			`  ((cdr x)))`

Unknown	x has already been evaluated:
	`(cdr x)`
		x has not yet been evaluated:
		`(let ((v ((cdr x))))`
		`  (set! (cdr x) v)`
		`  (set! (car x) '#T)`
		`  v)`
			No information, no optimization possible.

4.3.7 Applying Path Analysis at Code Generation

Given the models above, path analysis is easily applied to thunk optimization. After occurrence paths are computed for each function, the conditions *evaluated?*,

unevaluated?, and *last?* are determined for each variable occurrence by examining every path that contains that occurrence. The compiler then applies as much of this information as is appropriate for the mode it has chosen.[2] Our compiler uses cell mode, and the thunk benchmarks in Chapter 6 are all done using this mode. A comprehensive study of the relative practical advantages of cell mode and optimized cell mode could prove interesting, but would require an in-depth study of how arguments are typically used in functional programs and is beyond the scope of this thesis.

We have not discussed the details of code generation, but the compiler must take great care to ensure that a function's arguments are always passed in the mode and state of evaluation in which they are expected. Additional issues arise when higher-order constructs are considered. In this case simply matching the modes is not enough; their higher-order behaviors must match as well. Details of these and other issues in code generation may be found in [5].

[2]Of course, path analysis could be notified of this mode *a priori* and compute only the appropriate information.

Chapter 5

Implementation Issues

In the last three chapters we presented a theoretical model for path analysis and showed how other analyses could be derived from it. In this chapter we discuss some of the issues that arise in implementing path analysis, update analysis and thunk analysis, and suggest ways in which our implementation could be improved.

5.1 Implementating Path Analysis

5.1.1 Representing and Manipulating Paths

The first issue in implementating path analysis is how to represent a path. We represent every path p in two ways: as a vector v in which $v[i]$ contains p's i^{th} element, and as a table t indexed by path elements such that t's entry for the i^{th} path element is i. Although redundant, this dual representation makes it efficient to check for an element's membership and location in a path (via the table) while maintaining easy access to the ordered path (via the vector). We found that we used both representations often and that there was relatively little cost in maintaining them, so we consider the space and time overhead worthwhile. Sets of paths are represented as lists; this representation is discussed in more detail below.

The next issue is how to compare paths. Path elements are represented by

symbols, so two path elements can be compared efficiently via pointer comparison. Two paths can be compared fairly efficiently (time linear in the number of path elements) by walking down the vector representation and comparing elements. However, comparing *sets* of paths is expensive, and set manipulations such as *union* must be performed often. Unioning two unordered sets of size n can take time $O(n^2)$, and with each comparison taking time linear in the number of path elements, we found that path unions accounted for much of the runtime of path analysis.

The expense of set manipulation prompted our decision to make all alike paths *identical*, that is, to represent them by the same object. When a new path is created, it is entered into a hash table containing all existing paths; if the same path already exists the new path is identified with it, otherwise it is given a unique identifier and stored in the table. Using this technique, paths can be compared in constant time by their identifiers, and sets of paths can be represented by ordered lists or bit-vectors and compared in linear time. (Although the bit-vector representation may be slightly more efficient, factors such as widely varying set size led us to use ordered lists.) The expense is thus shifted from path-manipulation time to path-creation time; although comparing paths and sets of paths is now efficient, creating a new path requires that it be hashed, compared with other paths it collides with in the hash table, and finally assigned a unique identifier. Still, this method produced a notable speedup in the runtime required by path analysis.

5.1.2 Interactions with Other Analyses

It is important that we not lose sight of the fact that path analysis is an *operational* interpretation of a program. In an optimizing compiler in which many such operational interpretations are going on, care must be taken that they do not trip over one another. For example, although strictness analysis is subsumed by path analysis, the use of strictness information can also *affect* path analysis! Path analy-

sis assumes that data dependencies (and possibly orderings on primitive operators) supply the only ordering information, but if strictness analysis determines that a function f is strict in k of its n arguments, the compiler may choose to evaluate those k arguments *before* the function call, completely reshaping the order of evaluation information being inferred by path analysis. Consider the following function:

$$f(x, y, z) = \text{if } x \text{ then } y \text{ else if } z \text{ then } y + 1 \text{ else } y + 2$$

In the absence of strictness analysis, the paths through f are $\{\langle x, y\rangle, \langle x, z, y\rangle\}$. However, if after determining that f is strict in x and y (but not z) we decide to evaluate both x and y before the call, these paths become $\{\langle x, y\rangle, \langle x, y, z\rangle\}$.

Fortunately, the results of path analysis cannot affect strictness analysis, so strictness analysis can be performed first, and its information used by path analysis. This is a straightforward modification that we have incorporated into our implementation, but it should be noted that in doing so it is desirable to enforce an ordering on the evaluation of a function's strict arguments, just as we enforce an ordering on the evaluation of the arguments to a strict primitive function. Many compilers leave such an ordering to be decided by issues such as register allocation, but to do so in this case would vastly increase the complexity of path analysis.

5.1.3 Choosing an Ordering on Primitives

When path analysis was introduced in Chapter 2, we assumed that arguments to strict binary operators such as + were evaluated from left to right. This assumption was made to simplify the presentation of path analysis, but in fact there are several issues involved in choosing an ordering on strict functions. These issues are discussed in this section.

Strict Binary Functions

First, it should be clear that right-to-left could trivially replace left-to-right as the standard ordering, as left-to-right was an arbitrary choice to begin with.[1] Both choices have the advantage of fixing the ordering at compile-time, so that no additional analysis or decision-making is required at runtime. Furthermore, choosing a single ordering early eliminates the need to maintain the information associated with other possible orderings, which can be expensive. The disadvantage of an arbitrary fixed ordering is that it is not flexible. There are cases in which the ability to manipulate order of evaluation either at compile-time or at runtime makes a significant impact on a program's efficiency, but with the arbitrary fixed ordering this ability is lost.

One alternative is to do a compile-time analysis to determine an evaluation order for each operator, or for each *occurrence* of an operator, that is optimized with respect to some criteria. For example, if the goal is to do as much destructive aggregate updating as possible, and we wish to optimize evaluation order for this, an operator's arguments can be examined to determine where aggregates are updated and used, and to force the uses to occur before the updates. For example, consider the following function:

$$f(a, i, j, x) = sel(upd(a, j, x), i) + sel(a, i)$$

Clearly, a right-to-left ordering on the + inside of f will permit a to be updated destructively (assuming the global circumstances are favorable as well), while a left-to-right ordering will not. Furthermore, this can be determined at compile-time. Unfortunately, in the general case it is not possible to determine an optimal ordering at compile-time for the following reasons:

[1]Since most strict mathematical operators are binary (or unary, which is uninteresting from an ordering standpoint), this section discusses binary operators. However, it should be noted that the arguments in this section generalize directly to strict n-ary functions, for which $n!$ possible fixed evaluation orderings exist.

- In order to optimize *computation* time, the critical issue is not the number of lexical occurrences of *upd* but the number of *calls* to *upd* that can be done destructively. Consider the following example:

$$\begin{array}{lcl} f(a,b,i) & = & g(a,b,i) + h(a,b,i) \\ g(x,y,i) & = & \text{if } i = 0 \text{ then } x \text{ else } g(upd_1(x,i,sel(y,i)),y,i-1)\ ; \\ h(x,y,i) & = & \text{if } i = 100 \text{ then } x \text{ else } h(upd_2(x,i,i),upd_3(y,i,i),i+1)\ ; \end{array}$$

  A static analysis would at best determine the following:

  - If the + in f evaluates its arguments from left to right, upd_1 cannot be done destructively, but upd_2 and upd_3 can.
  - If the + in f evaluates its arguments from right to left, upd_2 and upd_3 cannot be done destructively, but upd_1 can.

  The probable conclusion would be to evaluate the + from left to right, thereby maximizing the number of lexical occurrences of *upd* that can be converted to destructive update. However, since upd_1, upd_2, and upd_3 all occur in recursive functions, there is no way to tell how often each will actually be called. If f is always called with $66 < i < 100$, the left-to-right decision is suboptimal.

- While an optimal runtime ordering can rarely be determined at compile-time, it is tempting to at least find an optimal static ordering, that is, one in which the most occurrences of *upd* may be done destructively. Unfortunately, even an optimal static ordering is intractable, because in general the ordering on strict operators depends on the ordering on arguments to functions, which in turn depends on the ordering on the strict operators! Thus the only possibility is to consider *all* possible combinations of orderings on strict operators and then determine which combinations provide the most static potential for destructive updating. While computable, this approach is impractical for most systems.

Although it may be impractical to compute a truly optimal ordering even in the static sense, some straightforward heuristics may improve performance. The simplest approach is a "0/1" analysis, in which for each expression e a boolean value $upds(e)$ is computed, where $upds(e)$ is $true$ if the computation of e might require a call to upd, $false$ otherwise. Then if the i^{th} strict operator op_i appears in the expression $e_1\ op_i\ e_2$, it will evaluate its arguments from left to right if $upds(e_2)$ is $false$, from right to left otherwise. A slightly more sophisticated analysis might count the lexical occurrences of upd in e_1 and e_2 and evaluate first the expression with the most occurrences of upd. An even more sophisticated analysis might determine what *aggregates* might be arguments to each upd in each of e_1 and e_2, and which might be used in each expression, and consider only the occurrences of upd that would be affected by the ordering on the op_i. This still is not a complete analysis because some of the $upds$ that appears to be affected by the ordering of op_i might be unsafe anyway because of a computation that occurs *after* the call to op_i. It is *this* analysis, the determination of which occurrences of upd cannot be done destructively regardless of the ordering on op_i, that requires full order of evaluation information, making a complete static analysis intractable.

Although we implemented the 0/1 analysis described above, we have found that for our purposes a fixed ordering is sufficient. It would be instructive to study the effect of the analyses described above on a variety of real programs, but such a study is beyond the scope of this thesis.

Other Strict Functions

The ordering issue is also interesting for primitives besides strict binary operators. Consider the *swap* function below:

$$swap(a, i, j) = upd_1(upd_2(a, i, sel(a, j)), j, sel(a, i))$$

Swap takes an array a and two integers i and j and returns a new array in which the values of $a[i]$ and $a[j]$ have been interchanged. The interesting point about *swap* is that upd_2 can be done destructively only if upd_1 evaluates its last two arguments before its first argument. A little thought about *upd* suggests that this will often be the case, since its first argument is an array, which could easily be produced by a call to *upd*, and its other two arguments are an integer and an arbitrary value, whose computations seem less likely to include an update. Thus it makes sense to put the arguments most likely to perform updates last, which in this case means *upd*'s second and third arguments should be evaluated before its first. Of course, a counterexample in which the opposite ordering would do better is easily constructed, but we speculate that such counterexamples occur infrequently in practice, and we fix a right-to-left ordering on *upd*'s arguments in our analysis. A similar argument applies to the arguments to *sel*, and we evaluate them from right to left as well.

It is interesting to note that *swap* requires a sort of "special treatment" in imperative languages as well. The standard swap code in Pascal looks like this:

$$\begin{array}{ll} temp & := a[i]; \\ a[i] & := a[j]; \\ a[j] & := temp; \end{array}$$

Element $a[i]$ is copied into *temp* so that $a[i]$ may be overwritten before its value is requested. Thus in imperative languages the temporary storage must be used explicitly, while in functional languages it is implicit, as an argument to the outer call to *upd*. Furthermore, if the arguments to *upd* were evaluated in such an order that the inner update could not be done destructively, a *trailer* would be required to hold the shadowed value of $a[i]$; this trailer represents exactly the storage that is used by *temp* in the Pascal program.

5.1.4 Higher-Order Constructs

Our implementation of path analysis does little inferencing on higher-order constructs. Any time a function becomes *anonymous*, by being passed into a function, returned from a function, or stored inside a list or array, the path information about that function is lost. Although theoretically path analysis extends nicely to the higher-order case, the higher-order information is very expensive to compute. Using the higher-order analysis described in Section 2.5, a *path pair* describing the order of evaluation properties of an expression contains a path *and* a higher-order behavior for that expression, where the higher-order behavior is a function of one argument that describes how the expression behaves when applied. To compare two paths, both parts of the path pairs must be compared, which means comparing *functions* over the powerdomain of path pairs. Functions are notoriously difficult to compare; a similar situation arises when strictness pairs are used for higher-order strictness analysis, and in the worst case it is resolved by *enumerating* both functions over their domains.[36] Since first-order strictness information is contained in a two-element domain (where the elements represent termination and non-termination), such an enumeration is feasible even if the higher-order behavior of the functions is several levels deep. Unfortunately, the size and complexity of the $Path$ domain renders the enumeration trick impractical here, and in the absence of an efficient mechanism for comparing functions a full higher-order analysis becomes intractable.

Although no real higher-order analysis is performed, our implementation of path analysis does not abandon higher-order programs entirely. The goal is to analyze the first-order portions of the program as usual, isolating the effect of the higher-order constructs as much as possible. The simplest technique is to introduce into the path domain a new element $\top_p$ that is stronger than any other element of $Path$. Intuitively, $\top_p$ represents a path that contains no useful information but cannot be

improved. In the path-append operator ":", $\top_p$ now dominates; that is,

$$\begin{array}{rrcl} \forall p, & \top_p : p & = & p : \top_p = \top_p \\ \forall p \neq \top_p, & \bot_p : p & = & p : \bot_p = \bot_p. \end{array}$$

The main advantage of this approach is efficiency; $\top_p$ is a concise way to represent an unimprovable path. The disadvantage is information loss; in a sense, we are giving up too much too early. Consider the following example, which uses the curried notation of the higher-order syntax:

$$f\ g\ x\ y = \text{if } g\ x\ y \text{ then } x + y \text{ else } x$$

In this example, the call to unknown function g returns $\{\top_p\}$, which dominates both possible paths through the conditional, yielding $\top_p$ as the only possible path through g. This is unfortunate, as we would like to preserve the information we do have about the paths through f — for example, that x must be evaluated in either path — while still accounting for the uncertainty introduced by the call to g.

The opposite extreme is to consider all possible paths through g, and incorporate them into the paths through f. This is analogous to the approach taken in strictness analysis when no information is available, where "no information" amounts to assuming that the function is not strict in any argument. The equivalent for path analysis, however, is to enumerate the set of all possible paths, which for an n-ary function contains $n! + (n-1)! + ... + 1$ elements, clearly an expensive alternative.

The cost of this approach can be greatly reduced if we are willing to modify the path structure by letting a single path element point to a *set* of elements, any number of which could be evaluated in any order at this point in the path. In the above example, this approach would give the following paths for f:

$$\{\langle\{x,y\},x,y\rangle, \langle\{x,y\},x\rangle\}$$

(Note that appearing in a "set" path element does not prohibit an identifier from appearing elsewhere in that same path.) Applications of path analysis must now

interpret these "set elements" in the most conservative way: to strictness analysis, the answer to $x \in \{x, y\}$ is $false$; to update analysis, the answer is $true$; and to thunk analysis, it depends on which question is being asked.

We actually use a variation of the technique just described in which we enumerate a *subset* of the possible paths through an unknown function that is safe for our applications, and incorporate this subset directly into the paths through f. For an unknown n-ary function g, we create a set containing the empty path, the bottom path, and n n-element paths, each of which starts with a different argument. Then every argument appears before every other argument in some path, and the possibilities of no arguments being evaluated and of non-termination are explicitly included. This is a safe approximation for strictness analysis (because of the empty path), path analysis (because every argument is seen to be evaluated before every other argument), and thunk analysis (because every argument is seen to be evaluated before *and after* every other argument). It is clearly more expensive than the "set" approach described above, but has the advantage of keeping the structure of paths intact and simplifying their applications.

5.1.5 Nested Equation Groups

The syntax of the first-order language defined in Chapter 1 is *flat*, that is, it does not permit equation groups to be nested and it assumes that all functions are named. This is a fully general model, since any functional program can be *lambda-lifted* [23] so that it contains only combinators, but we have found that in practice it is often convenient to deal with nested programs directly.[2] This is not difficult, but several new issues arise.

[2]This is particularly true since a nested ALFL program generates a nested T program, and T handles nested environments efficiently.

New Syntax and Semantics

Our syntax now includes equation groups as follows:

c	$\in$	Con	constants
x	$\in$	Bv	bound variables
p	$\in$	Pf	primitive functions
f	$\in$	Fv	function variables
eg	$\in$	$EqGp$	equation groups, where $eg = \{f_i(x_1, ..., x_n) = e_i;$ result $e\}$
e	$\in$	Exp	expressions, where $e = c \mid x \mid p(e_1, ..., e_n) \mid f(e_1, ..., e_n) \mid eg$
pr	$\in$	$Prog$	programs, where $pr = \{f_i(x_1, ..., x_n) = e_i\}$

A program can now be seen as an equation group, but to preserve the meaning of a program as an environment we have assumed that it contains no result clause.

While the details of the new semantics are not important, it should be noted that the meaning of an equation group nested at level n is the meaning of its result expression in an environment that attaches meaning to all functions defined at level n or less, and that the meanings of the functions in an equation group are defined in the same way as the meanings of the functions in a program.

Non-local variables

Non-local variables must be treated carefully. Consider the following example:

$$\begin{array}{l} \{\ g(a,b) = \ \{\quad f(x) = a + x + a; \\ \qquad\qquad\qquad\quad \text{result } f(b) + f(a)\}; \\ \quad \text{result } g(1,2)\} \end{array}$$

The path through g must reflect the use of a in f, since a is actually evaluated before b. This implies that f must *export* its non-local variables along with its path, so that they are included in the path of any expression that calls f. Thus the paths for these functions are as follows (the occurrences of a have been numbered left-to-right,top-to-bottom, so a_1 and a_2 appear in f and a_3 appears in the *result*

clause):

$$\begin{aligned} f: &\ \{\langle a_1, x, a_2\rangle\} \\ g: &\ \{\langle a, b\rangle\} \\ f': &\ \{\langle a_1, x, a_2\rangle\} \\ g': &\ \{\langle a_1, b_1, a_2, a_1, a_3, a_2\rangle\} \end{aligned}$$

Even in basic paths, non-local variables are treated differently from local variables in that more than one occurrence of each non-local variable may appear in a path. This does not affect the local meaning of the path, which derives only from the bound variables occurring in it, but is essential for the path to be able to export the information about its free variables to functions that use it. In occurrence paths, nothing changes except that *a single occurrence of a variable may appear more than once in a path.* For example, a_1 and a_2 each appear twice in the path through g'. This must be possible, since occurrences represent textual appearances and multiple calls to a function that refers to a non-local variable may result in multiple demands for the value of a single textual occurrence of a variable. Note that paths are still guaranteed to be finite in length and number, since the number of appearances of a given occurrence is bounded by the number of textual calls to the function in which it appears.

From an efficiency standpoint, it appears that programs with nested equation groups create *longer* paths, which are more expensive to maintain and manipulate, than their lambda-lifted counterparts. While this is true, it must be balanced by the advantage obtained by performing *dependency analysis* on the equation groups. Using dependency analysis, non-mutually recursive equations are separated into distinct equation groups, greatly facilitating the fixpoint computation. The ALFL compiler performs dependency analysis, and in our experience its benefits outweigh the additional path sizes.

Constants

A similar issue arises in the use of named constants. Consider the following example:

$$\begin{array}{l}\{f(x) = \{\ \ a\ =\ x+1; \\ \qquad\qquad\ g(y) =\ a+y+a; \\ \qquad\qquad\ \text{result } g(3)\}; \\ \ \text{result } f(2)\}\end{array}$$

When g is called in f's result clause, is x's value demanded once or twice? In other words, is a shared, or is the expression $x+1$ textually substituted for it everywhere? This is an *operational* issue that has no effect on the result of the program, but may affect the ways in which it can be optimized. We assume that a is shared, and in fact transform the program so that all such sharing is moved to bound variables. The transformed program becomes:

$$\begin{array}{l}\{f(x) = (\lambda\ a.\{\ \ g(y) =\ a+y+a; \\ \qquad\qquad\qquad\ \text{result } g(3)\})(x+1); \\ \ \text{result } f(2)\}\end{array}$$

Although we have not discussed how we treat λ-expressions, they pose no problems since they are simply unnamed functions.

5.1.6 Symbolic Analysis

Like many static analyses, path analysis makes no attempt to do a "logical trace" of conditionals. In the expression $(p \rightarrow c_1, a_1) + (p \rightarrow c_2, a_2)$ the two conditionals share a predicate, so the only possible paths are $\langle p, c_1, c_2\rangle$ and $\langle p, a_1, a_2\rangle$. However, path analysis will also find the "impossible" combinations $\langle p, a_1, c_2\rangle$ and $\langle p, c_1, a_2\rangle$, as it will not notice that both conditionals must branch the same way. An analysis to detect this situation could be layered on top of path analysis.

5.2 Implementing Update Analysis

Update analysis presents its own interesting issues because it is inherently a *collecting* analysis. That is, we cannot even derive *safe*, much less accurate, information

from a function f without knowing everywhere that f is applied. This leads to several implementation issues that do not arise for path analysis.

5.2.1 Higher-Order Constructs

The discussion of higher-order constructs in the last section is insufficient for update analysis. Consider the example again:

$$f\ g\ x\ y = \text{if } g\ x\ y \text{ then } x + y \text{ else } x$$

Suppose a function h is passed in for g, and h updates one of its arguments. That update cannot be done destructively, but we cannot know that without knowing that h is applied inside of f. (If h were applied directly inside of f, its occurrence of *upd* would be exported into f's paths, where the update conflict would appear.) An analysis could be done to determine a superset of the functions in which a function that becomes anonymous could be used, but without a full higher-order treatment such an analysis would be very weak. Our solution is to identify those functions that might become anonymous, an easy static analysis, and disallow destructive updating inside of them.

5.2.2 Index Analysis

We do no index analysis in conjunction with path analysis. That is, we make no attempt to discover cases where a function updates an array's i^{th} location and later accesses its j^{th} element, and $\forall i, j, i \neq j$. Such an analysis could be layered on top of path analysis, and could provide substantial benefit. Although there has been some related work in optimizing functional programs[15], most work in this area has been in the field of numerical analysis.

5.3 Implementing Thunk Analysis

Once occurrence paths have been computed, thunk analysis is straightforward. For every bound variable occurrence, the paths through the function in which that variable is bound are examined. For each path the questions *evaluated?*, *unevaluated?*, and *last?* are answered, corresponding to whether or not this is the first and/or last occurrence of this bound variable in this path, and then an AND operation is performed on the answers to each question across all paths. Thus only questions that are answered *true* for all paths are answered *true* for the variable occurrence. The bottom path answers *true* to all questions.

Using our treatment of higher-order constructs described in Section 5.1.4, thunk analysis can operate directly on the resulting paths without modification. The other models described in that section do require slight changes to thunk analysis. If $\top_p$ is used, it should answer *false* to *evaluated?*, *unevaluated?*, and *last?* for all paths. If set notation is used to clump together variables about which we have no ordering (or even evaluation) information, the set elements must be interpreted in the most conservative way. That is, if x_{ik} is contained in a set element appearing before x_{ij} in some path, then both *evaluated?* and *unevaluated?* are false for x_{ij}. Similarly, if x_{ik} occurs in a set element appearing after x_{ij} in some path, *last?* must be false for x_{ij}.

Nested equation groups also present no problems for thunk analysis, since all information about a bound variable occurrence x_{ij} is contained in the path through the function f in which x_i is bound.

Chapter 6

Benchmarks

In this chapter we present and discuss benchmarks for programs that are optimized using path analysis, and for path analysis itself. Our benchmarks are for programs in ALFL [17], a functional language developed at Yale, and were run on a MacintoshII with 13 megabytes RAM. The ALFL programs were translated into T [26] and then submitted to Orbit, the T compiler. Times shown are for compiled T code, using version 3.1 of T with 8 megabyte heaps. Since ALFL translates into T, the ALFL compiler can at best generate optimal T code so that the compiled ALFL program runs as fast as the hand-coded T program for the same problem. Note that T uses applicative-order evaluation, and that arrays are non-functional and are implemented efficiently through destructive udpating. Listings of the programs we used for benchmarks are included in Appendix B.

6.1 Update Analysis

This section discusses the speedups gained through update analysis. Table 6.1 presents benchmarks for the following programs:

- *quicksort:* Hoare's quicksort.
- *bubsort:* Bubblesort.

- *tridiag:* Tridiagonal factorization.
- *init:* Vector initialization.
- *matinit:* Matrix initialization.
- *matmult:* Matrix multiplication.

The size of the structures manipulated by each of the programs is noted in the table. Note that vector size does not affect update analysis; large vectors were used for the smaller programs to bring run times out of the noise level. The 1000-element vector was added for *init* because it allowed the copying strategy to be benchmarked in a function where updating dominated the runtime. In addition to update analysis, strictness analysis, termination analysis, and uncurrying were performed on all programs. For each program, the table gives the cpu time used by the hand-coded T program using iteration and destructive operations whenever possible; for the ALFL program using update analysis; for the ALFL program using the trailers implementation; and when possible, for the ALFL program using the copying implementation. In each of these programs update analysis was able to determine that all updates could be done destructively.

In *quicksort* update analysis resulted in optimal performance - the time for the compiled ALFL code is the same as the time for the hand-coded T version. Using trailers, however, produced a three-fold slowdown, which would be even worse if adjusted to account for eventual garbage collection of the additional memory required. Quicksort's functions are all strict in their arguments and its computations are vector-intensive, so the effects of more or less efficient vector operations are quite pronounced.

Bubsort differs from quicksort in two ways: its functions are strict in fewer arguments, and it does more selection relative to the amount of updating that it does. The strictness issue is reflected in the difference between the time for the

BENCHMARK	COPYING	TRAILERS	UPD ANALYSIS	T
QUICKSORT (10,000 elements)	—	2.60	0.73	0.73
BUBSORT (100 elements)	3.80	3.45	0.20	0.10
BUBSORT (200 elements)	—	12.10	0.56	0.43
INIT (1000 elements)	2.6	0.124	0.004	0.002
INIT (10,000 elements)	—	1.35	0.04	0.02
TRIDIAG (1000 elements)	—	1.88	0.88	0.88
MATMULT ((30×30)×(30×30))	3.4	4.15	1.15	—
MATINIT (30×30)	2.48	0.18	0.06	0.02

Table 6.1: Benchmarks for update analysis on ALFL programs

ALFL program with destructive updates and the time for the T program. The more interesting point, however, is how close the trailer and copying times are. In the trailer implementation, access and update both carry penalties, while in the copying implementation only update is penalized. Thus the additional overhead of copying is to some extent compensated for by faster access. Of course, part of the picture is missing here; the copying implementation uses much more memory than the trailer implementation, and for a 200-element array the time for the trailer implementation increases in about the expected proportion, while the copying implementation cannot complete execution without garbage collecting (twice!). The destructive implementation, of course, is far superior to either.

Init is interesting because it is not strict in its last argument and it is extremely update-intensive. The non-strict argument accounts entirely for the difference between the T runtime of .02 and the optimized ALFL runtime of .04 on the 10,000-element array. However, this difference is swamped by the jump to 1.35 that occurs when trailers are used. The slowdown is so great because *init* does almost nothing except update arrays, so inefficiency in an array operation has a very strong effect on the overall runtime. Running *init* on a 1000-element array produced times uncomfortably close to the noise level (although the ratios remained very close to those for the 10,000-element array), but this was the only update-intensive example on which we were able to benchmark the copying implementation.

Tridiag can be fully strictified, but is interesting because it relies heavily on floating-point operations. T uses a *consing* floating-point implementation, requiring two longwords for each floating-point operation. The overhead thus introduced is substantial, and the effect of an inefficient array implementation is muffled by the floating-point inefficiencies. However, update analysis still produces optimal performance, and using trailers results in a performance degradation of over a factor of two, so the optimization is still significant.

Matmult shows significant speedup from update analysis. The interesting point here is that because selections greatly outnumber updates, trailers are actually *slower* than a copying implementation!

Matinit shows slightly better speedup than *matmult* largely because it performs no other interesting runtime function besides matrix operations, so the effect of improved performance on these operations is more pronounced. Nevertheless, its speedup is less than that of *init* because of the greater cost of the matrix operations and the overall greater complexity of the program.

6.1.1 Conclusions

In a function whose dominant costs stem from array manipulation, update analysis can produce speedups of one to two orders of magnitude; in a function with high overhead from sources such as non-strict functions or expensive runtime operations, particularly if the number of array manipulations is relatively small, the effect may be much smaller. The speedup is significant, however, for all of these array-based functions. The speedup produced by trailers can also be significant, but is much less than that of update analysis, and as the *matmult* benchmark demonstrates, the trailer representation will actually lose to copying when array selections greatly outnumber array updates.

6.2 Thunk Analysis

All of our thunk analysis was done using the *cell mode* described in Section 4.3. Our examples include the following programs:

- *thunks:* A small recursive program that performs simple arithmetic function computations.
- *fac:* Tail-recursive factorial with floating-point multiplication.

- *sumn:* Factorial using addition instead of multiplication.
- *tally:* A recursive function that treats array elements as positive or negative and tries to make them sum to a given total.
- *matmult:* The matrix multiplication program introduced above.

These are small programs, but are designed to show where thunk analysis is and is not effective. Recall that when combined with strictness analysis, thunk analysis optimizes only non-strict arguments since strict arguments do not cause the creation of thunks. Furthermore, the extent to which even non-strict arguments can be optimized depends not only on the order of evaluation information that can be inferred for them, but also on whether or not they are used in a strict context in the program. For example, consider *init* once again:

$$init(a, i, x) = \text{if } i = 0 \text{ then } a \text{ else } init(upd(a, i, x), i - 1, x)$$

Using paths based on the information that *init* is strict in a and i but not in x, thunk analysis can infer that the value of the second lexical occurrence of x (the third argument in the call to init) will already be computed when it is demanded. However, since *init* expects its third argument delayed, x is not forced at the function call and the information on how to optimize that force is wasted. Often such an optimization is possible, however, as shown by the programs below.

Table 6.2 shows the runtimes for several programs with and without thunk analysis and the associated optimizations. Again, except as noted, strictness analysis, uncurrying, and termination analysis were also performed on all functions.

Thunks shows more than a factor of two speedup because it has very few costs except those associated with the delay and force of its one lazy argument, and the force can be optimized with information derived from thunk analysis. Thus this is

BENCHMARK	unoptimized	thunk opt
THUNKS	0.17	0.07
FAC (n=50)	0.17	0.15
SUMN (n=10000)	0.33	0.25
TALLY (n=13)	0.29	0.25
MATMULT (30 × 30)	1.19	1.19

Table 6.2: Benchmarks for thunk analysis on ALFL programs

in a sense a best case; in fact, the actual speedup from a general FORCE to FORCE-EVALD, the optimized version for an already evaluated thunk, is a factor of three, so the factor of two+ speedup exhibited by *thunks* is near optimal.

Fac is a small program on which we did not perform strictness analysis to see how the overhead of all-lazy arguments affects the speedup produced by thunk analysis. Fac has one optimizable thunk and shows about 10% speedup; while the effect of the thunk optimization is to some extent muffled by the totally lazy evaluation, it is further affected by our use of floating-point multiplication.[1] This speedup should be compared with that of *sumn* below, in which the floating-point expense is removed.

Sumn is simply *fac* (again without strictness analysis) with the floating-point multiplication replaced by a fixed-point addition. As expected, the decreased overhead improves the percentage speedup.

Tally is a doubly-recursive function that takes an array of integers, lower and upper bounds for the array, and a total value, and tries to make the elements of

[1]Fixed-point multiplication was unable to produce a value large enough to generate interesting runtimes.

the array sum to the total by allowing each element to be treated as positive or negative. It contains one optimizable thunk, and our numbers are for inputs for which the values in the array do not tally to the total, so all possible recursive calls are taken. A 14% speedup is shown.

Matmult contains two optimizable thunks, but exhibits no speedup with the thunk optimization. The effect of the optimizations is being swamped by the overhead of many multiplications and matrix manipulations, and does not contribute significantly to the total runtime.

6.2.1 Conclusions

While the effect of thunk analysis is less dramatic than that of update analysis, it can have a definite impact on runtimes. It is difficult to assess how often the situation in *matmult* will occur; clearly, it depends on the number of optimizable thunks and on how often the values of these thunks are demanded, but because the evaluation status of an argument must match the status expected by the function that calls it, the interactions between strict and non-strict arguments are somewhat non-intuitive. However, the numbers in the next section show that thunk analysis is inexpensive once paths have been computed, so it is worthwhile even for programs where the speedup it produces is small.

6.3 Analysis Time

The final point of interest is *analysis time*: How long does it take path analysis, and then update analysis and thunk analysis, to infer the information required for the optimizations discussed above? As discussed in Section 2.3.3, the worst-case complexity of path analysis is at least exponential in the number of arguments to a function. However, this gives little practical information about how long the analysis

will take; some worst-case exponential algorithms are quite speedy in practice, while some linear algorithms with large constants are very slow.

Table 6.3 gives cpu times in seconds for *preparation, path analysis, update analysis* and *thunk analysis* for six programs. *Preparation* includes parsing, dependency analysis and some node initializations within the parse tree. *Path analysis* finds all *update* paths, that is, its paths contain update elements, and it does aggregate propagation at the same time. While the analysis could be modified to compute only "ordinary" paths, the additional cost of the update information is small so we compute it automatically. Both basic paths and occurrence paths are computed in this stage, but this does not include the time required to get the strictness information that can be incorporated into path analysis. *Update analysis* takes the paths found by path analysis and the program tree and determines which updates may not be done destructively. This includes finding the transitive closure discussed in Chapter 3 and the additional paths it implies. *Thunk analysis* takes the paths found by path analysis and for each bound variable occurrence determines whether it is guaranteed to be the first, not first, or last occurrence of that bound variable to be demanded.

The programs are listed roughly in order of size, ranging from two lines for *init* to about 15 lines for *qsort*. The first observation is that path analysis is far more expensive than preparation, update analysis, or thunk analysis, so we will focus our discussion on it. The next observation is that our times are all for small programs, and there is a reason for this: as the size and complexity of a program's functions grow, path analysis becomes relatively much more expensive. This is demonstrated to some extent by the times for *matmult* and *qsort*. Although they differ in length by only a couple of lines, their analysis times differ by a factor of three; we speculate that this is because the functions defined by *qsort* are more mutually recursive than those in *matmult* and generate more paths. For larger programs this effect is much more pronounced. For example, a program to perform lu-decomposition that was

only about 40 lines long ground path analysis to a near halt. Interestingly enough, the ALFL compiler's standard strictness analysis ran very slowly on precisely the same program! For both analyses, it was not the size of the program that caused problems but its structure of deeply nested function definitions and highly recursive equation groups; a 40-line (or much larger) program with a simpler structure and few recursive dependencies would yield to either analysis quite easily. Although the complexity of both strictness analysis and path analysis grows not with program size but with the number of arguments to a function, it seems that some programs push that complexity closer to its theoretical worst-case than others; in particular, we speculate that large programs do this much more often than small programs. Thus we must conclude that like many interprocedural analyses, path analysis as described and implemented in this thesis is not a practical tool for large programs.

BENCHMARK	prep	path	upd	thunk
INIT	0.3	0.8	0.1	0.03
MATINIT	0.4	0.4	0.4	0.03
TRIDIAG	1.2	3.8	1.1	0.13
MATMULT	1.3	5.0	1.1	0.32
BUBSORT	1.4	9.1	4.9	0.22
QSORT	1.3	13.9	0.9	1.1

Table 6.3: Runtimes of path, update, and thunk analysis on ALFL programs (sec)

Chapter 7

Other Models of Order of Evaluation

7.1 Order of Evaluation in a Parallel System

7.1.1 The Sequential Nature of Path Analysis

Path semantics describes the order of evaluation properties of a lazy functional program in a sequential system. As we discussed briefly in Chapter 2, the sequential model is enforced by the assumption that the arguments to a strict binary operator, for example +, are evaluated from left to right. As we pointed out in that discussion, the general model also considers an ordering from right to left, but under no circumstances can the evaluations be *interleaved*, as would be possible (and quite natural) in a parallel system.[1] Consider the following program:

$$f(a, b, c) = g(a, b) + c;$$

$$g(x, y) = \text{if } x \text{ then } y \text{ else } y + 1 \ ;$$

In a sequential system the two possible paths through f are $\langle a, b, c\rangle$ and $\langle c, a, b\rangle$; in a parallel system, however, there is an additional possibility: $\langle a, c, b\rangle$.

[1] Of course, such interleaving is possible in a sequential system as well, but since doing so offers no advantage we have not considered this possibility.

To see the problems this can cause, consider a slightly modified example:

$$f(a,b) = g(a,b) + g(b,a);$$

$$g(x,y) = \text{if } h(upd(x,i,j)) \text{ then } y \text{ else } y+1 \ ;$$

Here h is some unspecified function that is strict in its argument, and i and j are global variables that are not of interest. Assuming that the arguments to + could be evaluated in *either* order, the possible paths for f and g are shown below:

g : $\{\langle x, (upd_1, x), y\rangle\}$
f : $\{\langle a, (upd_1, a), b, (upd_1, b)\rangle, \langle b, (upd_1, b), a, (upd_1, a)\rangle\}$
f' : $\{\langle a_1, (upd_1, a_1), b_1, b_2, (upd_1, b_2), a_2\rangle, \langle b_2, (upd_1, b_2), a_2, a_1, (upd_1, a_1), b_1\rangle\}$

The paths associated with f' are the occurrence paths through f where distinct occurrences of f's variables are numbered statically from left to right.

It it clear from f's occurrence paths that if the arguments to + are evaluated from left to right, a can be updated destructively but b cannot be, and if they are evaluated right to left b can be updated destructively but a cannot be. Under no circumstances can both updates be done destructively, but it is always the case that one of the two can be. Of course, without knowing which ordering will be chosen for +, neither destructive update is safe, but typically an ordering would be fixed in advance or chosen on the basis of a compile-time analysis.

Now consider the case where the arguments to +, and hence the two calls to g, are evaluated in parallel. Let g^1 represent the call $g(a,b)$, and g^2 represent the call $g(b,a)$. Suppose g^1 updates a, then g^2 updates b, then g^1 uses b, then g^2 uses a. This produces the following path:

$$\langle a_1, (upd_1, a_1), b_2, (upd_1, b_2), b_1, a_2\rangle$$

In this path, *neither* update can be done destructively, a possibility that did not arise in the sequential case. Thus applying the sequential analysis to a parallel system can give *unsafe* results, and thus is not acceptable.

The reader may notice that while the parallel scenario presented here is worse than either of the choices in the sequential case, it is no worse than *both* of the sequential choices. That is, if in the sequential model we were not to choose an ordering but to assume that either ordering could occur, we would conclude that neither update could be done destructively, just as we did in the parallel case. In fact, this will hold in general. While assuming all possible orderings on strict operators does not accurately model order of evaluation in a parallel system, it does provide safe results when used as the basis for an aggregate update analysis. We call this model *extended update analysis.* The semantics for extended update analysis differs from that of regular update analysis only in the strict primitive functions, which change as follows:

$$
\begin{aligned}
\hat{\mathcal{U}}_k[\![+]\!] &= \lambda s.\{\ (none, \langle x^p : y^p \rangle), (none, \langle y^p : x^p \rangle) \mid (x, y) \in s\} \\
\hat{\mathcal{U}}_k[\![IF]\!] &= \lambda s.\{\ (c^a, \langle p^p : c^p \rangle), (a^a, \langle p^p : a^p \rangle) \mid (p, c, a) \in s\} \\
\hat{\mathcal{U}}_k[\![UPD_j]\!] &= \lambda s.\{\ (none, \langle x^p : i^p : a^p : \langle (j, a^a) \rangle \rangle \\
&\qquad (none, \langle x^p : a^p : \langle (j, a^a) \rangle : i^p \rangle \\
&\qquad (none, \langle i^p : a^p : \langle (j, a^a) \rangle : x^p \rangle \\
&\qquad (none, \langle i^p : x^p : a^p : \langle (j, a^a) \rangle \rangle \\
&\qquad (none, \langle a^p : \langle (j, a^a) \rangle : i^p : x^p \rangle \\
&\qquad (none, \langle a^p : \langle (j, a^a) \rangle : x^p : i^p \rangle \mid (a, i, x) \in s\}
\end{aligned}
$$

That this model guarantees safe results is stated in the theorem below:

Theorem 6 *Let* extended path analysis *be a path analysis in which no assumption is made about the order in which strict primitive operators evaluate their arguments. Then if the i^{th} lexical occurrence of upd (upd_i) cannot be done destructively in a parallel system, there exists at least one path in the set of paths found with extended path analysis in which upd_i cannot be done destructively.*

Proof: For upd_i to be unsafe as a destructive operator, two conditions must hold:

1. At some point in the computation, upd_i must update some aggregate a, and
2. At some later point in the computation, a must be used again.

Suppose that these conditions are satisfied during the evaluation of $op(e_1, ..., e_n)$, where op is a strict binary operator, and $e_1 ... e_n$ are arbitrary expressions. One of two sequences of events must occur:

1. Both the update of a and the subsequent use of a occur during the evaluation of e_i.

2. a is updated during the evaluation of e_i and used during the evaluation of $e_j, j \neq i$.

Clearly, sequence (1) can occur in either a sequential or a parallel system, as it does not involve op or e_2. Sequence (2) can occur in a parallel system only if the portion of e_1 containing the update of a occurs before the portion of e_2 containing a's use. This will occur in a sequential system if e_1 is evaluated before e_2, which is guaranteed to be included in the orderings considered by extended path analysis.□

7.1.2 Parallel Path Analysis

The easiest way to derive a parallel model from path analysis is to consider all possible interleavings of evaluation orders at strict operators. For example, if $+_p$ is the path interpretation of $+$, this means that

$$\begin{aligned}\langle x_1, x_2\rangle +_p \langle y_1, y_2\rangle = \{ \quad & \langle x_1, x_2, y_1, y_2\rangle \\ & \langle x_1, y_1, x_2, y_2\rangle \\ & \langle x_1, y_1, y_2, x_2\rangle \\ & \langle y_1, x_1, x_2, y_2\rangle \\ & \langle y_1, x_1, y_2, x_2\rangle \\ & \langle y_1, y_2, x_1, x_2\rangle\}\end{aligned}$$

In general, the number of paths generated in this way is bounded from below by 2^k where k is the number of elements in the shorter of the two path arguments to $+_p$ — a clear source of "path explosion."

7.2 An Alternative Sequential Model

The path model described in Chapter 3 is very general in that it computes *complete* order of evaluation information. However, for some applications less complete information may be sufficient. Consider the thunk optimization in which the i^{th} occurrence of a bound variable x may be accessed directly if we know that some j^{th} occurrence of x will definitely have been evaluated first. This requires much less information than is computed by path analysis, and if this were the only use for path analysis in a given application, it could be computed more directly and less expensively.

In this section we present four non-standard interpretations for a program pr, $\mathcal{B}_p$, $\mathcal{MB}_p$, $\mathcal{A}_p$, and $\mathcal{MA}_p$, each of which returns an environment which, when applied to an n-ary function f, an index i, and a set of elements of V to be bound to each bound variable occurrence, describes the behavior of x_i as follows:

1. $(B_p[\![pr]\!])[\![f]\!](i, s_1, ..., s_n)$ (*Before*): Returns the elements of the s_k that *must* be used before x_i is evaluated.

2. $(\mathcal{MB}_p[\![pr]\!])[\![f]\!](i, s_1, ..., s_n)$ (*Maybe Before*): Returns the elements of the s_k that *might* be used before x_i is evaluated.

3. $(\mathcal{A}_p[\![pr]\!])[\![f]\!](i, s_1, ..., s_n)$ (*After*): Returns the elements of the s_k that *must* be used before x_i is evaluated.

4. $(\mathcal{MA}_p[\![pr]\!])[\![f]\!](i, s_1, ..., s_n)$ (*Maybe After*): Returns the elements of the s_k that *might* be used after x_i is evaluated.

It is difficult to state a precise relationship between these analyses and path analysis. Although it is clear from the description in the next section that the information they contain could be derived from path analysis, it is not clear how much less

information they compute. It is also not clear how their complexities are related to that of path analysis; although each of these analyses is exponential in the worst case, the domains of bound variables on which they operate are so much simpler than the domains of paths that in practice they are much easier to compute. In short, what we are presenting here is simply a less general model that has an *intuitive* relationship to path analysis, but not necessarily a formal relationship.

Although the analyses described below are quite different from path analysis, it is helpful to introduce them by relating them to the intuitive notion of paths. In the remainder of this section we will use the term *path* as it was introduced in Section 2.1, but without implying any particular relationship to path semantics and path analysis.

7.2.1 Intuitive Description

Recall that a *path* through a function $f(x_1, ..., x_n)$ is an ordering on the evaluations of the x_i. A path can be represented as a sequence, for example $\langle x_i, ..., x_j, ..., x_k \rangle$; in this path, x_i is evaluated before x_j, and x_i and x_j are both evaluated before x_k. A path is not required to contain all of the x_i, since with lazy evaluation some of f's arguments might never be evaluated.

Now define the relations $\prec_p$ and $\succ_p$ and their negations as follows:

$$\begin{aligned}
x_j \prec_p x_i &\stackrel{\text{def}}{\equiv} x_j \text{ appears before } x_i \text{ in path } p \\
x_j \not\prec_p x_i &\stackrel{\text{def}}{\equiv} x_j \text{ does not appear before } x_i \text{ in path } p \\
x_j \succ_p x_i &\stackrel{\text{def}}{\equiv} x_j \text{ appears after } x_i \text{ in path } p \\
x_j \not\succ_p x_i &\stackrel{\text{def}}{\equiv} x_j \text{ does not appear after } x_i \text{ in path } p
\end{aligned}$$

For now, assume that each bound variable in f is used at most once, that is, that there is no sharing. Also assume that there is a single path p associated with f. We can now define a function $B_p[\![x_i]\!]$ that defines the set of things that appear before x_i in p, and a function $A_p[\![x_i]\!]$ that defines the set of things that appear after x_i in p:

$$\begin{aligned} x_j \in B_p[\![x_i]\!] &\Leftrightarrow x_j \prec_p x_i \\ x_j \in A_p[\![x_i]\!] &\Leftrightarrow x_j \succ_p x_i \end{aligned}$$

While this is useful if there is only one possible path through f, there are typically *many* possible paths, and at compile-time they all must be considered equally likely. Let P be the set of these possible paths, and let P_i be the largest subset of P such that all p in P_i contain x_i. If we now wish to talk about things that will *definitely* be evaluated before and after x_i, we have to modify our definitions for B and A:

$$\begin{aligned} x_j \in B[\![x_i]\!] &\Leftrightarrow (\forall p \in P_i)\ x_j \prec_p x_i \\ x_j \in A[\![x_i]\!] &\Leftrightarrow (\forall p \in P_i)\ x_j \succ_p x_i \\ x_j \in NB[\![x_i]\!] &\Leftrightarrow (\forall p \in P_i)\ x_j \not\prec_p x_i \\ x_j \in NA[\![x_i]\!] &\Leftrightarrow (\forall p \in P_i)\ x_j \not\succ_p x_i \end{aligned}$$

By restricting our set of paths to P_i we are *assuming* that x_i will be evaluated. This assumption holds throughout this section, but in the future we will abbreviate $\forall p \in P_i$ and $\exists p \in P_i$ by $\forall p$ and $\exists p$.

Note that we have dropped the subscripts on B and A, since they are now defined over all possible paths. We have also defined two other sets, NB (not before) and NA (not after), which are *not* complements of B and A.

We can talk about things that happen in some paths but not in others as things that *might* happen; $MB[\![x_i]\!]$ defines the set of things that might be evaluated before x_i, and $MA[\![x_i]\!]$ defines the set of things that might be evaluated after x_i:

$$\begin{aligned} x_j \in MB[\![x_i]\!] &\Leftrightarrow (\exists p)\ x_j \prec_p x_i \\ x_j \in MA[\![x_i]\!] &\Leftrightarrow (\exists p)\ x_j \succ_p x_i \end{aligned}$$

Note that $MB[\![x_i]\!]$ and $NB[\![x_i]\!]$ are complementary, as are $MA[\![x_i]\!]$ and $NA[\![x_i]\!]$. This will be useful later, because it is often easier to directly compute the set of all things that might occur than it is to compute the set of all things that cannot occur.

We can now lift the restriction that no formal parameter can be shared. We will termporarily retain the assumption that x_i is not shared, but any other x_j may have

k occurrences, denoted $x_{j1}...x_{jk}$, and a path is now a sequence of occurrences (we will denote the single occurrence of x_i by x_{i1}). This allows us to distinguish between x_j being *used* before (or after) x_i is evaluated and x_j being *evaluated* before (or after) x_i is evaluated. In terms of occurrences, the latter relationship amounts to knowing if the occurrence of x_j that is demanded first appears in the path before (or after) the first (and under our current assumptions only) occurrence of x_i. In the formulas below, $B_u, A_u, NB_u \text{and} NA_u$ make up the "use model", while $B_e, A_e, NB_e \text{and} NA_e$ make up the "evaluation model".

$$\begin{aligned}
x_j \in B_u[\![x_{i1}]\!] &\Leftrightarrow (\forall p)(\exists k)\ x_{jk} \prec_p x_{i1} \\
x_j \in A_u[\![x_{i1}]\!] &\Leftrightarrow (\forall p)(\exists k)\ x_{jk} \succ_p x_{i1} \\
x_j \in NB_u[\![x_{i1}]\!] &\Leftrightarrow (\forall k, p)\ x_{jk} \not\prec_p x_{i1} \\
x_j \in NA_u[\![x_{i1}]\!] &\Leftrightarrow (\forall k, p)\ x_{jk} \not\succ_p x_{i1}
\end{aligned}$$

$$\begin{aligned}
x_j \in B_e[\![x_{i1}]\!] &\Leftrightarrow (\forall p)(\exists k)\ x_{jk} \prec_p x_{i1} \\
x_j \in A_e[\![x_{i1}]\!] &\Leftrightarrow (\forall m, p)(\exists k)\ x_{jk} \succ_p x_{i1} \ \wedge\ x_{jm} \not\prec_p x_{i1} \\
x_j \in NB_e[\![x_{i1}]\!] &\Leftrightarrow (\forall k, p)\ x_{jk} \not\prec_p x_{i1} \\
x_j \in NA_e[\![x_{i1}]\!] &\Leftrightarrow (\forall k, p)\ (\exists m) x_{jk} \not\succ_p x_{i1} \ \vee\ x_{jm} \prec_p x_{i1}
\end{aligned}$$

From these definitions we can see that:

$$\begin{aligned}
B_e[\![x_{i1}]\!] &\equiv B_u[\![x_{i1}]\!] \\
A_e[\![x_{i1}]\!] &\equiv A_u[\![x_{i1}]\!] \cap NB_u[\![x_{i1}]\!] \\
NB_e[\![x_{i1}]\!] &\equiv NB_u[\![x_{i1}]\!] \\
NA_e[\![x_{i1}]\!] &\equiv NA_u[\![x_{i1}]\!] \cup B_u[\![x_{i1}]\!]
\end{aligned}$$

Thus the *evaluation* of x_j can be described in terms of the *uses* of x_j. This is important because the computational models developed in Section 7.2.2 model "use" much more naturally than "evaluation". Furthermore, recall that in the "uses" model we can substitute the complement of $MB[x_i]$ for $NB[x_i]$, further facilitating the computation. Thus to effectively compute the set of x_j that are [are not] evaluated before [after] x_i is evaluated, we need only to compute the set of x_j that are [might be] used before [after] x_i is evaluated. From this point on we will discuss only these four sets, since all others can be derived from them.

We still have one restriction, which is that x_i occurs only once. To lift this restriction, suppose that each argument x_i to function f has an arbitrary number

of occurrences, and we want to know what will [might be] used before [after] x_i is evaluated. Since it is the *evaluation* of x_i that we are interested in, for any given path p we only care about the first occurrence of x_i to appear in p. Of course, this may vary from one path to another, so for path p we refer to the first occurrence of x_i in that path as $x^p_{i\#}$.

Consider the case of Before. $x_j \in B_u[x_i]$ means that $(\forall p)(\exists m)x_{jm} \prec_p x^p_{i\#}$. But $(\forall k)x^p_{i\#} \prec_p x_{ik}$, so we are really just stating that

$$x_j \in B_u[\![x_i]\!] \Leftrightarrow (\forall p, k)(\exists m)x_{jm} \prec_p x_{ik}$$

Similar reasoning yields the following equations for the other sets:

$$\begin{aligned} x_j \in MB_u[\![x_i]\!] &\Leftrightarrow (\exists p, m)(\forall k)x_{jm} \prec_p x_{ik} \\ x_j \in A_u[\![x_i]\!] &\Leftrightarrow (\forall p)(\exists k, m)x_{jm} \succ_p x_{ik} \\ x_j \in MA_u[\![x_i]\!] &\Leftrightarrow (\exists p, k, m)x_{jm} \succ_p x_{ik} \end{aligned}$$

It is clear from the above definitions that the behavior of x_i depends on the behavior of the x_{ik}. For example, $B_u[x_i]$ may be computed by computing $B_u[x_{ik}]$ for each path p. If $B_{u_p}[x_{ik}]$ denotes $B_u[x_{ik}]$ for a single path p, and similarly for the other analyses, then

$$\begin{aligned} B_u[\![x_i]\!] &\equiv \cap_{p,k} B_{u_p}[\![x_{ik}]\!] \\ MB_u[\![x_i]\!] &\equiv \cup_p \cap_k B_{u_p}[\![x_{ik}]\!] \\ A_u[\![x_i]\!] &\equiv \cap_p \cup_k A_{u_p}[\![x_{ik}]\!] \\ MA_u[\![x_i]\!] &\equiv \cup_{p,k} A_{u_p}[\![x_{ik}]\!] \end{aligned}$$

In Section 7.2.2 we compute precisely the sets described by these equations, but our method is somewhat different from that implied in the above discussion. Instead of computing all paths through a program and for each path computing the properties of the x_{ik} that occur on that path, we start with an x_{ik} and compute a single property that holds for it over all paths. We still need to combine our results for occurrences to obtain a result for a bound variable, but it is done in a slightly

different way. The approach taken and its relationship to the problem description appears after each set of semantic equations.

Notational Conventions

To facilitate our presentation in the next section we adopt the following notational conventions.

First, we introduce *labeling* in order to handle backward flow properly. Every expression has a unique label. Labelled expressions have the form $l : e$, and we define *expr* and *label* by $expr(l) = e$, and $label(e) = l$. Thus an expression e may be referred to with or without its label, or solely by its label, without loss of information.

Second, we assume no sharing of expressions other than formal parameters; this results in no loss of generality, since any subexpression can be parameterized. As in the last section we must be able to distinguish between *occurrences* of bound variables; we refer to the j occurrences of a bound variable x_i as $x_{i1} \ldots x_{ij}$, where each x_{ik} is simply a labelled expression. Note that x_i is *not* an expression.

Finally, we introduce the notion of a *context*, as intuitively described earlier. We associate with every program P a function $context_P$ that gives the context of every subexpression in P. For example, consider the expression $E = [\![f(e_1, \ldots, e_i, \ldots, e_n)]\!]$ in program P (from now on we shall enclose syntactic objects in double brackets). Then $context_P[\![e_i]\!] = E$. Note that $context_P[\![x_{ij}]\!]$ is well-defined, but $context_P[\![x_i]\!]$ is not, since x_i is not an expression. For simplicity we refrain from a formal definition of *context* with respect to a given program, but its construction should be obvious.

7.2.2 Non-Standard Semantics

The non-standard semantics in this section are for programs in the "generic" first-order lazy functional language whose syntax and semantics were given in Section

1.5.

Preliminaries

We define four non-standard interpretations for a program $pr = [\![\{f_i(x_1, \ldots, x_n) = body_i\}]\!]$, $\mathcal{B}_p[\![pr]\!]$, $\mathcal{MB}_p[\![pr]\!]$, $\mathcal{A}_p[\![pr]\!]$, and $\mathcal{MA}_p[\![pr]\!]$. Each of these returns an environment which, when applied to a function f, an index j of an argument to f, and a set of elements of V to be bound to each bound variable occurrence, returns the set of things in V that will [might] be evaluated before [after] x_j is evaluated.

These semantic descriptions make use of the following auxiliary functions: Given some expression $e = f(e_1, \ldots, e_n)$:

$\mathcal{N}[\![e]\!]$= those e_i that will definitely be evaluated in evaluating e.
$\mathcal{D}[\![e]\!]$ = those e_i that might be evaluated in evaluating e.

$\mathcal{N}$ is simply first-order strictness analysis for expressions, and is adapted from the development in [21]. Note that both $\mathcal{N}[\![e]\!]$ and $\mathcal{D}[\![e]\!]$ concern expressions that are (or might be) evaluated *during* the evaluation of e, in contrast to the previous questions of what is evaluated *before* and *after* e's evaluation.

The following standard semantic domains are used throughout our analysis:

V, the set of variables of interest
Sv, the powerset of V
I, the set of indices of functional arguments
$Sfun = Sv^n \rightarrow Sv$, the space of functions that map sets of variables of interest to other sets
$Env = Fv \rightarrow Sfun$, the space of function environments
$Bve = Bv \rightarrow Sv$, the space of bound variable environemnts

For each interpretation, the function environment is an element of Env, and the bound variable environment is an element of Bve, thus:

$$\begin{array}{rcl} bve & \in & Bve \\ nenv, nenv', benv, & & \\ mbenv, aenv, maenv & \in & Env \end{array}$$

For each interpretation we also define a set of semantic functions. They are listed below with their types:

$$\begin{array}{lcl} \mathcal{K}'_n, \mathcal{K}_b, \mathcal{K}_{mb}, \mathcal{K}_a, \mathcal{K}_{ma} & : & Pf \rightarrow I \rightarrow (Sfun + Sv) \\ \mathcal{N}', \mathcal{B}, \mathcal{MB}, \mathcal{A}, \mathcal{MA} & : & Exp \rightarrow Bve \rightarrow Env \rightarrow Sv \\ \mathcal{N}'_p, \mathcal{B}_p, \mathcal{MB}_p, \mathcal{A}_p, \mathcal{MA}_p & : & Prog \rightarrow Senv \end{array}$$

First-Order Strictness ($\mathcal{N}$)

Our treatment of strictness analysis is taken almost directly from [21]. We omit the details, since our analysis differs only at the formal parameter, where we define N to operate on *occurrences* of bound variables, instead on the bound variables themselves. Thus $\mathcal{N}[\![x_{ij}]\!] bve\ nenv = bve[\![x_{ij}]\!]$, while $\mathcal{N}[\![x_i]\!] bve\ nenv$ is undefined. The standard strictness analysis may be trivially obtained from our version simply by requiring that all occurrences of a single bound variable behave identically.

As in [21], we also define $\mathcal{N}_p[\![pr]\!] = nenv$, where the environment $nenv$ is the least fixed point of the set of "strictness equations" for the functions in program pr. We use $nenv$ freely in our analyses.

Possible Definitions ($\mathcal{D}$)

$\mathcal{D}[\![e]\!]$ is used instead of strictness when we want to know what expressions *might* be evaluated in evaluating e, rather than the ones that are sure to be. $\mathcal{D}[\![e]\!]$ essentially traverses the subtree rooted at e and returns the values in bve of the occurrences of formal parameters found at the leaves.

$\mathcal{D} : Exp \rightarrow Bve \rightarrow Sv$

$\mathcal{D}[\![c]\!]bve = \{\}$

$\mathcal{D}[\![x_{ij}]\!]bve = bve[\![x_i]\!]$

$\mathcal{D}[\![p(e_1, \ldots, e_n)]\!]bve = \mathcal{D}[\![e_1]\!]bve \cup \ldots \cup \mathcal{D}[\![e_n]\!]bve$

$\mathcal{D}[\![f(e_1, \ldots, e_n)]\!]bve = \mathcal{D}[\![e_1]\!]bve \cup \ldots \cup \mathcal{D}[\![e_n]\!]bve$

Before ($\mathcal{B}$)

Intuitively, $\mathcal{B}$ takes an expression e and environments *bve* and *benv*, and returns a set of elements of V that must be used before e is evaluated. Note that $\mathcal{K}_b$ takes a primitive function p and returns a function whose first argument is an *index* indicating which of p's arguments is being evaluated, where the arguments are numbered from left to right starting with 1. For example, since we assume no fixed ordering on the arguments to $+$, $\mathcal{K}_b[\![+]\!]$ returns a function that always returns $\{\}$. For the conditional, however, we know that the predicate is evaluated before the arms, so $\mathcal{K}_b[\![IF]\!](i, p, c, a)$ returns p when i is 2 or 3, otherwise $\{\}$.

$\mathcal{K}_b\ [\![+]\!] = \lambda(i, x, y).\ \{\}$

$\mathcal{K}_b\ [\![IF]\!] = \lambda(i, x, y, z).$ if $i = 1$ then $\{\}$
else if $(i = 2) \vee (i = 3)$ then x
else *error*

$\mathcal{B}[\![e_i]\!]bve\ benv =$ case $context[\![e_i]\!]$

$[\![]\!]$	:: $\{\}$ (top of function)
$l : p(e_1, \ldots, e_i, \ldots, e_n)$	:: $\mathcal{K}_b[\![p]\!](i, \mathcal{N}[\![e_1]\!]bve\ nenv, \ldots, \mathcal{N}[\![e_n]\!]bve\ nenv)$ $\cup \mathcal{B}[\![expr(l)]\!]bve\ benv$
$l : f(e_1, \ldots, e_i, \ldots, e_n)$	:: $benv[\![f]\!](i, \mathcal{N}[\![e_1]\!]bve\ nenv, \ldots, \mathcal{N}[\![e_n]\!]bve\ nenv)$ $\cup \mathcal{B}[\![expr(l)]\!]bve\ benv$

$\mathcal{B}_p[\![\{f_i(x_1,\ldots,x_n) = body_i\}]\!] = benv$ whererec

$$benv = [(\lambda(j, y_1, ..., y_n).\bigcap_k(\mathcal{B}[\![x_{jk}]\!]\ [\{\}/x_{j*}, y_m/x_{m*}, m \neq j]\ benv))/f_i] \quad (1)$$

Context and *nenv* (the strictness environment) have been dropped as arguments to $\mathcal{B}$; they are independent of this analysis, and have presumably been defined in some outer scope.

In our path description for $B[\![x_i]\!]$ in Section 7.2.1, we intersected over all bound variable occurrences and over all paths. Here, our function $\mathcal{B}[\![x_{ik}]\!]$ returns the things that *must* occur before x_{ik}, i.e., that occur before x_{ik} on all paths. (This is evident in the definitions for constant functions and by the fact that the forward flow is done with strictness, $\mathcal{N}$). Thus we have effectively done the intersection over paths, and it remains only to intersect over k; this is precisely the intersection that appears in (1).

Maybe Before ($\mathcal{MB}$)

Intuitively, $\mathcal{MB}$ takes an expression e and environments *bve* and *mbenv*, and returns a set of elements of V that might be used before e is evaluated in environment *bve*.

$$\mathcal{K}_{mb}[\![+]\!] = \lambda(i, x, y).\ \text{if } i = 1 \text{ then } y$$
$$\text{else if } i = 2 \text{ then } x$$
$$\text{else } error$$
$$\mathcal{K}_{mb}[\![IF]\!] = \lambda(i, x, y, z).\ \text{if } i = 1 \text{ then } \{\}$$
$$\text{else if } (i = 2) \vee (i = 3) \text{ then } x$$
$$\text{else } error$$

$$\mathcal{MB}[\![e_i]\!]bve\ mbenv = \text{case } context[\![e_i]\!]$$

$$\begin{array}{ll}
[\![]\!] & :: \{\} \\
l : (p(e_1, \ldots, e_i, \ldots, e_n)) & :: \mathcal{K}_{mb}[\![p]\!](i, \mathcal{D}[\![e_1]\!]bve, \ldots, \mathcal{D}[\![e_n]\!]bve) \\
 & \quad \cup \mathcal{MB}[\![expr(l)]\!]bve\ mbenv \\
l : (f(e_1, \ldots, e_i, \ldots, e_n)) & :: mbenv[\![f]\!](i, \mathcal{D}[\![e_1]\!]bve, \ldots, \mathcal{D}[\![e_n]\!]bve) \\
 & \quad \cup \mathcal{MB}[\![expr(l)]\!]bve\ mbenv
\end{array}$$

$$\mathcal{MB}_p[\![\{f_i(x_1, ..., x_n) = body_i\}]\!] = mbenv \text{ whererec}$$

$$mbenv = [(\lambda(j, y_1 \ldots y_n).\ \text{let } bve_1 = [\{\}/x_{j*}, y_m/x_{m*}, m \neq j]$$

$$bve_2 = [(\mathcal{MB}[\![x_{jk}]\!]\ bve_1\ mbenv)/x_{jk}, V/x_{m*}, m \neq j]$$

$$\text{in}\quad (\mathcal{N}'[\![body_i]\!]\ bve_2\ nenv') \cap (\bigcup_k (bve_2[\![x_{jk}]\!] \cap \mathcal{B}[\![x_{jk}]\!]bve_2\ benv)))/f_i] \quad (2)$$

where $\mathcal{N}'[\![body_i]\!]$ returns the intersection of the bound variable occurrences in which $body_i$ is strict, and is defined below:

$$\mathcal{K}'_n[\![IF]\!] = \lambda(x, y, z).\ x \cap (y \cup z)$$

$$\mathcal{K}'_n[\![+]\!] = \lambda(x, y).\ x \cap y$$

$$\mathcal{N}'[\![p(e_1, ..., e_n)]\!]\ bve\ nenv' = \mathcal{K}'_n[\![p]\!](\mathcal{N}'[\![e_1]\!]\ bve\ nenv', ..., \mathcal{N}'[\![e_n]\!]\ bve\ nenv')$$

$$\mathcal{N}'[\![f(e_1, ..., e_n)]\!]\ bve\ nenv' = nenv'[\![f]\!](\mathcal{N}'[\![e_1]\!]\ bve\ nenv', ..., \mathcal{N}'[\![e_n]\!]\ bve\ nenv')$$

$$\mathcal{N}'[\![x_{ij}]\!]\ bve\ nenv' = bve[\![x_{ij}]\!]$$

$$\mathcal{N}'[\![c]\!]\ bve\ nenv' = V$$

$$\mathcal{N}'[\![\{f_i(x_1, \ldots, x_n) = body_i\}]\!] = nenv' \text{ whererec}$$

$$nenv' = [(\lambda y_1 ... y_n.(\mathcal{N}'[\![body_i]\!]\ [y_j/x_{j*}]\ nenv'))/f_i]$$

In our path description for $MB[\![x_i]\!]$, we intersected over all x_{ik} in a path and then unioned over all paths. Here, $\mathcal{MB}[\![x_{ik}]\!]$ finds the things that might occur before x_{ik} on *any* path, so we have effectively done the union over all paths (again, this is

apparent in the definitions of $\mathcal{K}_{mb}$ and in the use of $\mathcal{D}$ for forward flow). However, we still need to combine the values for $\mathcal{MB}[\![x_{ik}]\!]$ to get a single value for x_i (note that simply intersecting $\mathcal{MB}[\![x_{ik}]\!]$ over k is not correct). If x_{ik} appears in every path (i.e., if f is *strict* in x_{ik}), then for x_j to appear before all occurrences of x_i in some path it must appear before x_{ik} in that path, and thus must be an element of $\mathcal{MB}[\![x_{ik}]\!]$. Thus it is clear that we need to intersect $\mathcal{MB}[\![x_{ik}]\!]$ over all strict x_{ik}, and this is precisely what $\mathcal{N}'$ does in the first half of the intersection in (2).

But a function will be strict in an occurrence of x_i only if it is strict in x_i, which we do not require, since we *assume* that x_i will be evaluated. If the function is not strict in x_i, then the set of things that might occur before x_i is simply the set of things that occur before $x^p_{i\#}$ in any path p. We do not know which x_{ik} serve as $x^p_{i\#}$ for some p, but we can determine this set by noting that if for some p, $x_{ik} = x^p_{i\#}$, then *no other occurrence of* x_i *definitely occurs before* x_{ik}, since x_{ik} occurs first in some path. Notice that in the second half of the intersection in (2) we intersect $\mathcal{MB}[\![x_{ik}]\!]$ with $\mathcal{MB}[\![x_{im}]\!]$ where x_{im} definitely occurs before x_{ik}. This is done in an environment (bve_2) in which if no such x_{im} is found, the set of all variables is returned, so the intersection has no effect. Otherwise, $\mathcal{MB}[\![x_{ik}]\!]$ is reduced to those things that might occur before occurrences of x_i that definitely occur before x_{ik}(!). Thus, the effect of the second half of the intersection in (2) is to union $\mathcal{MB}[\![x_{ik}]\!]$ over all k for which, for some p, $x_{ik} = x^p_{i\#}$.

In sum, we effectively *intersect* over those strict x_{ik} and *union* over those non-strict x_{ik} that might occur first, and then intersect these sets. It is interesting to note that if f is strict in x_i, then (2) is entirely determined by the first half of the intersection (since the second half will return a superset of the first half); similarly, if f is not strict in x_i, (2) is entirely determined by the second half of the intersection (the first half will return a superset of the second half).

After ($\mathcal{A}$)

Intuitively, $\mathcal{A}$ takes an expression e and environments *bve* and *aenv*, and returns a set of elements of V that must be used after e is evaluated.[2]

$$\mathcal{K}_a[\![+]\!] = \lambda(i,x,y).\ \{\}$$
$$\mathcal{K}_a[\![IF]\!] = \lambda(i,x,y,z).\ \text{if } i = 1 \text{ then } y \cap z$$
$$\text{else if } (i = 2) \vee (i = 3) \text{ then } \{\}$$
$$\text{else } error$$

$$\mathcal{A}[\![e_i]\!]bve\ aenv = \text{case } context[\![e_i]\!]$$
$$[\![]\!] \qquad :: \{\}$$
$$l : (p(e_1,\ldots,e_i,\ldots,e_n)) \qquad :: \mathcal{K}_a[\![p]\!](i, \mathcal{N}[\![e_1]\!]bve\ nenv,\ldots,\mathcal{N}[\![e_n]\!]bve\ nenv) \cup \mathcal{A}[\![expr(l)]\!]bve\ aenv$$
$$l : (f(e_1,\ldots,e_i,\ldots,e_n)) \qquad :: aenv[\![f]\!](i, \mathcal{N}[\![e_1]\!]bve\ nenv,\ldots,\mathcal{N}[\![e_n]\!]bve\ nenv) \cup \mathcal{A}[\![expr(l)]\!]bve\ aenv$$

$$\mathcal{A}_p[\![\{f_i(x_1,...,x_n) = body_i\}]\!] = aenv \text{ whererec}$$
$$aenv = [(\lambda(j,y_1,\ldots,y_n).\ \text{let } bve_1 = [\{\}/x_{j*}, y_m/x_{m*}, m \neq j]$$
$$bve_2 = [(\mathcal{A}[\![x_{jk}]\!]\ bve_1\ aenv)/x_{jk}, \{\}/x_{m*}, m \neq j]$$
$$\text{in}\quad (\mathcal{N}[\![body_i]\!]\ bve_2\ nenv) \cup (\bigcap_k (bve_2[\![x_{jk}]\!] \cup \mathcal{B}[\![x_{jk}]\!]bve_2\ benv)))/f_i] \qquad (3)$$

In our path description for $A[\![x_i]\!]$, we unioned over all x_{ik} in a path and then intersected over all paths. Here, $\mathcal{A}[\![x_i]\!]$ finds the things that will occur after x_i on all

[2]In the following analysis, note that we disregard the things that are used *during* the evaluation of x_i. To include them is trivial, since strictness ($\mathcal{N}$) computes the things that must be used during the evaluation of x_i, and defs ($\mathcal{D}$) computes the things that might be used. We can account for these sets in the translation from the "uses" model to the "evaluation" model, e.g., $A_e[\![x_i]\!] = A_u[\![x_i]\!] \cap NB_u[\![x_i]\!] \cap ND[\![x_i]\!]$, where ND is the complement of $\mathcal{D}$.

paths, so we have effectively intersected over all paths (as for $\mathcal{B}$, strictness is used for the forward flow). But as for $\mathcal{MB}$, we did this path intersection first, so what remains to be done is not a simple union. However, reasoning analogous to that used for $\mathcal{MB}$ applies here. If x_{ik} appears in every path, then anything that appears after it in all paths will definitely appear after x_i so we want to union $\mathcal{A}[\![x_{ik}]\!]$ over all strict x_{ik}. This is a simple application of strictness analysis, and accounts for the first half of the union in (3).

Again, strictness is not enough. If a function is not strict in x_i, then we can be sure that x_j appears after x_i only if it appears after *all* x_{ik} that might occur first. We determine the set of x_{ik} that might occur first in some path in the same way as before, by noting that their "before" set is empty, and then we intersect $\mathcal{A}[\![x_{ik}]\!]$ over that set. This makes up the second half of the union in (3).

In sum, we effectively *union* over the strict x_{ik} and *intersect* over those non-strict x_{ik} that might occur first, and then union these sets. Again notice that if f is strict in x_i, then (3) is entirely determined by the first half of the union (since the second half will return a subset of the first half), and if f is not strict in x_i (3) is determined by the second half of the union (since the first half will return a subset of the second half).

Maybe After ($\mathcal{MA}$)

Intuitively, $\mathcal{MA}$ takes an expression e and environments *bve* and *maenv*, and returns a set of elements of V that might be used after e is evaluated.

$$\mathcal{K}_{ma}[\![+]\!] = \lambda(i,x,y).\ \text{if } i = 1 \text{ then } y$$
$$\text{else} \quad \text{if } i = 2 \text{ then } x$$
$$\text{else} \quad \mathit{error}$$
$$\mathcal{K}_{ma}[\![IF]\!] = \lambda(i,x,y,z).\ \text{if } i = 1 \text{ then } y \cup z$$

$$\begin{array}{ll} \text{else} & \text{if } (i=2) \vee (i=3) \text{ then } \{\} \\ \text{else} & error \end{array}$$

$$\begin{array}{ll} \mathcal{MA}[\![e_i]\!] bve\ maenv = \text{case } context[\![e_i]\!] & \\ \quad [\![]\!] & :: \{\} \\ \quad l:(p(e_1,\ldots,e_i,\ldots,e_n)) & :: \mathcal{K}_{ma}[\![p]\!](i, \mathcal{D}[\![e_1]\!] bve, \ldots, \mathcal{D}[\![e_n]\!] bve) \\ & \qquad \cup \mathcal{MA}[\![expr(l)]\!] bve\ maenv \\ \quad l:(f(e_1,\ldots,e_i,\ldots,e_n)) & :: maenv[\![f]\!](i, \mathcal{D}[\![e_1]\!] bve, \ldots, \mathcal{D}[\![e_n]\!] bve) \\ & \qquad \cup \mathcal{MA}[\![expr(l)]\!] bve\ maenv \end{array}$$

$$\mathcal{MA}_p[\![\{f_i(x_1,\ldots,x_n) = body_i\}]\!] = maenv \text{ whererec}$$

$$maenv = [(\lambda(j, y_1, \ldots, y_n).\ \cup_k(\mathcal{MA}[\![x_{jk}]\!]\ [\{\}/x_{j*}, y_m/x_{m*}, m \neq j] maenv))/f_i] \qquad (4)$$

In our path description for MA, we unioned over all bound variable occurrences and over all paths. Here, our function $\mathcal{MA}[\![x_{ik}]\!]$ returns the set of things that *might* occur after x_{ik}, i.e., that occur after x_{ik} in some path. Thus we have effectively unioned over all paths, and it remains only to union over k; this is precisely the union that appears in (4).

7.2.3 Discussion

While the path model described in Section 7.2.1 differs substantially from the model described in this section, it is interesting to note that they contain precisely the same symmetries. In the path model, for B we intersect over both paths and occurrences; for MB, we intersect over occurrences and union over paths; for A, we union over occurrences and intersect over paths; and for MA, we union over both paths and occurrences. In this section, for $\mathcal{B}$ we intersect over all occurrences; for $\mathcal{MB}$, we intersect over strict occurrences and union over non-strict occurrences; for $\mathcal{A}$, we union over strict occurrences and intersect over non-strict occurrences; and

for $\mathcal{MA}$, we union over all occurrences. While these dualities are not surprising in either model, it is interesting that two quite different approaches yield results of such similar form.

Note that since all of our domains are finite, and we use only monotonic operators, each of the analyses described in this section is guaranteed to have a least fixpoint.

Chapter 8

Related Work, Conclusions, and Future Work

8.1 Related Work

Although we know of no other work that focuses on a general model for order of evaluation of expressions, the theoretical foundations on which our analyses are based and the problems to which we apply our analyses have been studied in other contexts.

8.1.1 Denotational Semantics and Abstract Interpretation

Denotational semantics has long been used as a means of attaching meaning to syntax. A brief history of its developement may be found in [35]; references include [35,33,2]. Abstract interpretation was introduced by Cousot and Cousot in [11], and was first applied to the problem of optimizing functional languages by Mycroft in [27]. Since then a variety of work has explored its use in the optimization of programming languages, including work on related issues such as collecting interpretations[20].

8.1.2 Destructive Aggregate Updating

Schmidt [34,32] discusses destructive aggregate updating in the context of denotational specifications. His work focuses on determining the serializability of the semantic store argument of a denotational definition, a property he terms *single-threadedness*, and implementing a single-threaded store argument as a global variable. However, the work in [34] relies on a call-by-value reduction scheme, and that in [32] relies on an even more restrictive scheme called *block* call-by-value.

Hudak [18] applies a static reference-counting model to destructive aggregate updating. He describes a non-standard reference-counting semantics for a first-order functional language with applicative-order evaluation, and gives a computable abstraction and a collecting interpretation of this abstraction through which compile-time information about liveness can be inferred. The generality of this model is unclear, as difficulties are encountered in attempts to extend it to handle lazy evaluation and higher-order constructs.

Gopinath [15] explores *targeting*, the reuse of storage space where copying would otherwise be necessary, in applicative-order functional languages. Besides the assumption of applicative-order evaluation, the emphasis of his work differs from ours in that he makes use of the properties of specific operators, particularly array operators, to do several forms of symbolic analysis, and concentrates on determining when strategies such as divide-and-conquer are guaranteed to use disjoint portions of an aggregate structure. He assumes that liveness properties have already been computed, although in a later work [16] he gives a specification for computing these properties.

Bruynooghe [6] discusses techniques for peforming a global analysis on logic programs through which inaccessible data structures are overwritten to gain efficiency. Typing and mode restrictions supplied by the user are used by the static analysis in attempting to determine when it is safe to overwrite a data structure. It is not

clear, however, exactly how the techniques described here would apply to functional languages, and whether or not they could handle lazy evaluation and higher-order functions.

Gopalakrishnan and Srivas [14] use a graph model to determine when destructive updating of abstract data types is possible for a restricted group of functional programs, and investigate the possibility of transforming an expression E that does not permit destructive updating into an equivalent expression E' that does. Their work is restricted, however, to applicative-order languages with recursion limited to iteration. No interprocedural analysis is performed, yielding a simple but ungeneral model.

Some languages dispose of incrementally updatable arrays entirely, instead offering *monolithic* structures that are initialized upon creation and cannot be updated later [4,25]. Although easy to implement, these monolithic structures lack flexibility and are unsuitable for many numerical algorithms. Arvind's I-structures [3] are a compromise, providing write-once incremental updating in which the array initialization is essentially spread out over time. I-structures can also be implemented efficiently without complex compile-time or run-time analysis and they do not restrict parallelism, but for a sequential system they lack the flexibility of general incrementally updatable structures.

8.1.3 Thunk Analysis

Although we know of no other interprocedural analysis aimed at optimizing thunks, similar local analyses are implemented in many compilers, including the Lazy ML compiler [24].

8.1.4 Strictness Analysis

Strictness analysis has been widely studied. Mycroft [27] first studied the transformation of call-by-need into call-by-value, and showed that abstract interpretation is a suitable framework for detecting when the transformation is safe. Clack and Peyton-Jones [10] give a practical introduction to strictness analysis, showing its costs and benefits in terms of speedup and analysis time. Burn, Hankin and Abramsky [8] and Hudak and Young [21] address higher-order strictness analysis for flat domains. Hughes [22] addresses strictness analysis for non-flat domains, and Wadler and Hughes approach strictness analysis using *contexts*, a technique very similar to the backward flow we use in the alternative model of order of evaluation discussed in Section 7.2.

8.2 Conclusions

This thesis set out to study methods for statically inferring order of evaluation information for lazy functional programs, and to investigate optimizations to which this information could be applied. We presented several models of order of evaluation but focused on first-order path analysis, with which we attempted to answer the following questions:

1. Can order of evaluation information be statically inferred for lazy functional programs?

2. If obtainable, is order of evaluation information useful for optimizing lazy functional programs?

The answer to the first question is "Yes, but it's expensive." In Chapter 4 we show that path analysis subsumes strictness analysis and therefore has a lower-bound complexity of 2^n, where n represents the number of arguments to a function. While

this is only a theoretical lower bound, the numbers in Section 6.3 demonstrate that path information is indeed expensive to compute; in fact, for large programs path analysis is not a practical tool. However, this result is less surprising in light of another recent result in this area: a full interprocedural strictness analysis with a limited higher-order analysis was recently determined to be impractical for large programs as well.[1] While disappointing, our "negative" result does not by any means discredit the path model of order of evaluation; it simply means that path analysis must serve as a *basis* from which we can abstract to generate more tractable analyses.

The answer to the second question is a definite "Yes". In Chapter 3, path analysis is formally extended to handle the aggregate update problem via *update analysis*. Update analysis contains only slightly more information than path analysis, but as the figures in Chapter 6 demonstrate, the conversions from trailer updating to destructive updating that it permits can result in runtime speedups of more than an order of magnitude. Although the effect of thunk analysis is less dramatic than that of update analysis, it is clear that for some programs it can produce significant speedups.

This work has some important theoretical results as well. *Non-standard semantics* are only beginning to be widely used as a basis for semantic analysis, and the developments of path semantics and update semantics and their abstractions are interesting in their own right. Update semantics is particularly intriguing because analyzing a function f requires information about *where f is called*, sometimes called a *collecting* analysis[20]. Update semantics incorporates a limited form of collecting *directly into the semantics* by way of "exporting" update information from a called function into the function that calls it. The fact that *any* form of interprocedural update analysis must do some sort of collecting sets it apart from

[1] This conclusion was reached by the functional programming group at Yale, and was based on the strictness analyzer described in [36].

other semantic analyses such as strictness analysis and path analysis, and invites further investigation.

Overall, we conclude that order of evaluation information is an important compile-time tool and that path semantics and path analysis are useful runtime and compile-time models for order of evaluation. Further work is required, however, to develop a practical tool for computing order of evaluation information. This and other areas open to further investigation are discussed in the next section.

8.3 Future Work

This work suggests extensions in several directions, both practical and theoretical.

1. *Improving the runtime performance of path analysis.*

   - *Alternative representations for paths.* Much of the runtime cost of path analysis comes from maintaining and manipulating linear paths. We chose such a linear representation because it is straightforward and closely reflects our path model, but an alternative runtime representation of paths may improve performance. In particular, a graphical representation similar to that discussed in [14] would be much more compact, but it is not clear how such a graphical representation would be manipulated in the context of an interprocedural analysis. This is a promising direction, however, which we expect to pursue.
   - *A more abstract path analysis.* Considering the expense of path analysis and the importance of its applications, it may be useful to derive a *more abstract* path analysis that trades precision for savings in computation time. While the direction such an abstraction should take is not clear, possibilities include limiting the interprocedural analysis and maintaining only a subset of a function's order of evaluation information.

As discussed in our conclusions above, the necessity of this sort of abstraction is currently being recognized in the context of other expensive compile-time analyses as well.

2. *Models for lazy evaluation.* Although path analysis provides information for optimizing lazy evaluation, the extent to which this information can be used depends on how lazy evaluation is implemented. This is a complex issue, as many aspects of optimization and code generation are affected by the representation of thunks and the interfaces used to manipulate expressions in various stages of evaluation. While this problem extends beyond the issue of order of evaluation, it is representative of the issues that arise when analyses are combined in "real" compilers.

3. *Theoretical models for semantic analysis.* Path analysis and update analysis raise several interesting theoretical issues. The powerdomain construction used in path analysis is straightforward, but as discussed in Section 2.3.5 a non-flat path domain would give rise to a general powerdomain construction whose semantics in the context of the path model is not clear. Such powerdomain problems often arise in semantic descriptions, and an investigation of the role of powerdomains in semantic analyses could be instructive. Update analysis proposes another area for future study: how *contextual information* can be described using denotational semantics, and the relationship between update analysis's "export" semantics and a true *collecting interpretation.*

Appendix A

Proof of Theorem 5

This proof involves a small trick in that we show that the paths derived using the *independent* attribute method yield the same analysis, instead of those derived from the *relational* attribute method that was presented in Chapter 3. We do this because the form of the independent attribute method more closely matches that of Hudak and Young's strictness analysis, making comparison of the two analyses easier. While the relational attribute method yields a more precise (smaller) set of paths (and thus might seem to give a *better* strictness analysis), it is easy to show that the two methods behave identically when the strictness function F (defined below) is applied to the resulting sets. (To see why, consider the extra paths generated by the independent method. Every such path must contain a path generated by the relational method, and thus performing the intersection in F with the sets derived from these extra paths can have no effect on the result.)

The independent method is very much like the relational method, except that bound variables are bound to *sets of paths*, instead to a single paths as in the relational method. The differences are apparent in the domains and equations shown below (only those equations that differ from the relational method are shown):

Domains

D,			the domain of path elements
$Path$,			the domain of paths
$P(Path)$,			the powerdomain of $Path$
$Pfun$	$=$	$\bigcup_{n=1}^{\infty}((P(Path))^n \rightarrow P(Path))$,	the function space mapping paths to paths
$Aenv$	$=$	$Fv \rightarrow Pfun$,	the function environment
Bve	$=$	$Bv \rightarrow P(Path)$,	the bound variable environment

Functions

$$\begin{aligned} \mathcal{A} &: Exp \rightarrow Bve \rightarrow Aenv \rightarrow P(Path) \\ \mathcal{A}_k &: Pf \rightarrow Pfun \\ \mathcal{A}_p &: Prog \rightarrow Penv \end{aligned}$$

$$\begin{aligned} \mathcal{A}[\![f(e_1,...,e_n)]\!]\ bve\ penv &= penv[\![f]\!](\mathcal{A}[\![e_1]\!]\ bve\ penv,...,.\mathcal{A}[\![e_n]\!]\ bve\ aenv) \\ \mathcal{A}[\![p(e_1,...,e_n)]\!]\ bve\ aenv &= \mathcal{A}_k[\![f]\!](\mathcal{A}[\![e_1]\!]\ bve\ aenv,...,.\mathcal{A}[\![e_n]\!]\ bve\ aenv) \\ \mathcal{A}_k[\![IF]\!] &= \lambda(p,c,a).p * (c \cup a) \\ \mathcal{A}_k[\![+]\!] &= \lambda(x,y).x * y \\ \mathcal{A}_p[\![\{f_i(x_1,...,x_n) = body_i\}]\!] &= aenv \text{ whererec} \\ aenv &= [(\lambda(y_1,...,y_n).\mathcal{A}[\![body_i]\!]\ [y_j/x_j]\ aenv)/f_i] \end{aligned}$$

The "cross-product" operator $*$ is defined in terms of the path-append operator "::" (defined in the relational semantics) as follows:

$$\{p_1,...,p_m\} * \{p_{m+1},...,p_n\} =$$

$$\{p_1 :: p_{m+1}, p_1 :: p_{m+2},..., p_1 :: p_n,..., p_m :: p_{m+1},..., p_m :: p_n\}$$

We now proceed with our proof.

Let $\perp_{Sv}$ be the bottom element in the strictness domain (following [21]), $\perp_p$ the bottom element in the path domain, and $\{\perp_p\}$ the bottom element in the powerdomain of paths (as in Section 3.2). Note that $\perp_{Sv}$ represents the set of all variables V, and $\{\perp_p\}$ represents the set containing only the non-terminating path. Define

$$senv_0 = [(\lambda(z_1,...,z_n).\perp_{Sv})/f_i]$$

$$aenv_0 = [(\lambda(z_1, ..., z_n).\{\bot_p\})/f_i]$$

$$senv_k = [(\lambda(z_1, ..., z_n).\mathcal{S}[\![e_i]\!][z_j/x_j]senv_{k-1})/f_i]$$

$$aenv_k = [(\lambda(z_1, ..., z_n).\mathcal{A}[\![e_i]\!][z_j/x_j]aenv_{k-1})/f_i]$$

Let

$$\begin{array}{rcl} p_i & \in & Path \\ y_i & \in & D \text{ (path elements)} \\ s_i & \in & P(V) \text{ (strictness sets)} \end{array}$$

To "translate" from paths to strictness sets, we define the function F as follows:

$$F\{p_1, ..., p_m\}(y_1, ..., y_n)(s_1, ..., s_n) = p'_1 \cap p'_2 \cap ... \cap p'_m$$

$$\text{where } p'_i = \bigcup\{s_j \mid y_j \in p_i\}$$

We assert that $\forall y_i \in D, y_i \in \bot_p$.

Let *senv* be the strictness environment as found by Hudak and Young, and $\mathcal{S}$ be the strictness function; also let *aenv* be the path environment, and $\mathcal{A}$ be the path function. Then we claim that if the u_i are strictness expressions composed of s_i, and the v_i are sets of paths that are composed of y_i,

$$x \in senv[\![f_i]\!](u_1, ..., u_n) \iff x \in F(aenv[\![f_i]\!](v_1, ..., v_n))(y_1, ..., y_n)(s_1, ..., s_n)$$

$$\text{where } u_j = F\ v_j\ (y_1, ..., y_n)(s_1, ..., s_n)$$

Let $\Psi(senv, aenv)$ be the above predicate. (In using *Psi* we will often drop the qualifications on u_j, but it is still implied.) Since $senv_*$ and $aenv_*$ are constructed from finite domains and monotonic operators, every chain $senv_0, senv_1, ...$, $aenv_0, aenv_1, ...$ is guaranteed to be of finite height, and thus the predicate is admissible. Now let $senv_*$ be the least fixed point of the strictness equations, and $aenv_*$ the least fixed point of the path equations, then we will use structural and fixpoint induction to show that $\Psi(senv_*, aenv_*)$ holds.

First, consider $\Psi(senv_0, aenv_0)$:

$$\Psi(senv_0, aenv_0) = x \in \bot_{Sv} \iff x \in F\{\bot_p\}(y_1, ..., y_n)(s_1, ..., s_n)$$

$\perp_{Sv}$ is defined to be V, the set of all strictness elements, and by definition $\forall i, y_i \in \perp_p$. Thus we have

$$\Psi(senv_0, aenv_0) = x \in V \iff x \in \bigcup_i s_i$$

which is true, since here V is defined to be precisely $\bigcup_i s_i$.

Now consider the case of $\Psi(senv_k, aenv_k)$:

$\Psi(senv_k, aenv_k) =$

$$x \in \mathcal{S}[\![e_i]\!]senv_{k-1}[u_j/x_j] \iff x \in F(\mathcal{A}[\![e_i]\!]aenv_{k-1}[v_j/x_j])(y_1, ..., y_n)(s_1, ..., s_n)$$

This requires structural induction on e_i. Let $a = senv_{k-1}[u_j/x_j]$, $b = aenv_{k-1}[v_j/x_j]$, $c = (y_1, ..., y_n)(s_1, ..., s_n)$.

1. e_i is a constant. Then $\Psi(senv_k, aenv_k)$ becomes

   $$x \in \{\} \iff x \in F\{\langle\rangle\}c$$

   which, applying F, becomes

   $$x \in \{\} \iff x \in \{\}$$

2. e_i is a bound variable x_j. Then $\Psi(senv_k, aenv_k)$ becomes

   $$x \in u_j \iff x \in F\ v_j\ c$$

   By the structural induction hypothesis,

   $$u_j = F\ v_j\ c$$

   so trivially the implication holds.

3. $e_i = f(e_1, ..., e_n)$. Then $\Psi(senv_k, aenv_k)$ becomes

   $$x \in senv_{k-1}[\![f]\!](\mathcal{S}[\![e_1]\!]a, ..., \mathcal{S}[\![e_n]\!]a)$$

$$\Longleftrightarrow$$

$$x \in F(aenv_{k-1}[\![f]\!](\mathcal{A}[\![e_1]\!]b, ..., \mathcal{A}[\![e_n]\!]b))c$$

By the structural induction hypothesis,

$$\mathcal{S}[\![e_j]\!]a = F(\mathcal{A}[\![e_j]\!]b)c$$

but then the fixpoint induction hypothesis immediately applies.

4. $e_i = IF(e_1, e_2, e_3)$. Then $\Psi(senv_k, aenv_k)$ becomes

$$x \in \mathcal{S}[\![IF(e_1, e_2, e_3)]\!]a \iff x \in F(\mathcal{A}[\![IF(e_1, e_2, e_3)]\!]b)c$$

Applying the definitions for $\mathcal{S}$ and $\mathcal{A}$, we get

$$x \in (\mathcal{S}[\![e_1]\!]a \cup (\mathcal{S}[\![e_2]\!]a \cap \mathcal{S}[\![e_3]\!]a)) \iff x \in (F(\mathcal{A}[\![e_1]\!]b \cup (\mathcal{A}[\![e_2]\!]b * \mathcal{A}[\![e_3]\!]b))c)$$

Let $Q_1 = \mathcal{S}[\![e_1]\!]a \cup (\mathcal{S}[\![e_2]\!]a \cap \mathcal{S}[\![e_3]\!]a)$, $Q_2 = \mathcal{A}[\![e_1]\!]b \cup (\mathcal{A}[\![e_2]\!]b * \mathcal{A}[\![e_3]\!]b)$. Thus we are trying to show that

$$(x \in Q_1 \iff x \in F\ Q_2\ c)$$

Note that by the definition of F,

$$x \in (F\ paths\ c) \iff (\forall p \in paths)(\exists y_j \in p)\ x \in s_j \qquad (A.1)$$

Clearly,

$$x \in Q_1 \iff (x \in \mathcal{S}[\![e_1]\!]a) \vee (x \in \mathcal{S}[\![e_2]\!]a \wedge x \in \mathcal{S}[\![e_3]\!]a) \qquad (A.2)$$

By $(A.1)$ and the definition of $*$, we can see that

$$x \in F\ Q_2\ c \iff ((\forall p \in \mathcal{A}[\![e_1]\!]b)(\exists y_j \in p)\ x \in s_j) \ \vee$$

$$(((\forall p \in \mathcal{A}[\![e_2]\!]b)(\exists y_j \in p)\ x \in s_j) \wedge ((\forall p \in \mathcal{A}[\![e_3]\!]b)(\exists y_j \in p)\ x \in s_j))$$

which, by $(A.1)$, implies that

$$x \in Q_2 \iff (x \in F(\mathcal{A}[\![e_1]\!]b)c) \vee ((x \in F(\mathcal{A}[\![e_2]\!]b)c) \wedge (x \in F(\mathcal{A}[\![e_3]\!]b)c)) \tag{A.3}$$

But by the structural induction hypothesis, we know that

$$\mathcal{S}[\![e_i]\!]a = F(\mathcal{A}[\![e_i]\!]b)(y_1, ..., y_n)(s_1, ..., s_n)$$

and so $(A.2)$ and $(A.3)$ are precisely equivalent and the implication holds.

The proofs of the other primitive functions take the same form, and their details are left to the reader.

Appendix B

Text of Benchmarks

QUICKSORT

```
quicksort vec n == qsort vec 1 n;
qsort vec left right ==
   { pivot == sel vec left;
     result (left >= right) -> vec,
            scanright vec left right;
     scanright v l r == (l=r) -> (finish (upd v l pivot) l),
                        (sel v r) >= pivot -> scanright v l (r-1),
                          scanleft  (upd v l (sel v r)) (l+1) r;
     scanleft  v l r == (l=r) -> finish (upd v l pivot) l,
                        ((sel v l) <= pivot) -> scanleft  v (l+1) r,
                         scanright (upd v r (sel v l)) l (r-1);
     finish v mid == qsort (qsort v left (mid-1)) (mid+1) right;
   };
```

BUBSORT

```
bsort a n == {result bsort2 a true;
              bsort2 a change == not(change) -> a,
                                 sortall a 0 0;
              sortall a i change ==
                (i=(n-1)) -> bsort2 a change,
                 ((sel a i) > (sel a (i+1))) ->
                      sortall (swap a i (i+1)) (i+1) true,
                  sortall a (i+1) change;
              swap a i j == upd (upd a i (sel a j)) j (sel a i)};
```

INIT

```
init a i x == (i=0) -> a, init (upd a i x) (i-1) x;
```

TRIDIAG

```
tridiag c a b n ==
  {tri1 c a b i == (i>n) -> scons a (scons b (scons c [])),
       tri2 (upd c (i-1) ((sel c (i-1)) / (sel b (i-1))))
            (upd a i ((sel a i) / (sel b (i-1))))
            b
            i;
   tri2 c a b i ==
```

```
    tri1 c
         a
         (upd b i ((sel b i)-((sel a i)*(sel b (i-1))*(sel c (i-1)))))
         (i+1);
   result tri1 c a b 2};
```

MAT_MULT

```
matmult a arows acols b brows bcols ==
       {result mult_matrices (mkm arows bcols zero_array) 0 0;
        mult_matrices array i j ==
           (i=arows) -> array,
              (j=bcols) -> mult_matrices array (i+1) 0,
                 mult_matrices (updm array i j (do_mult i j)) i (j+1);
        do_mult row col == {result do_mult2 0 0;
                            do_mult2 acc i == (i=acols) -> acc,
do_mult2 (acc + (selm a row i)*(selm b i col)) (i+1)}};
```

MATINIT

```
initm a i j x == {result  loop1 a i x;
                  loop1 a i  x == (i<0) -> a, loop2 a i j x;
                  loop2 a i j x == (j<0) ->loop1 a (i-1) x,
                                   loop2 (updm  a i j x) i (j-1) x};
```

THUNKS

```
 thunks a b c == (a=0) -> b, (c=0) -> b, test2 (a-1) (b+c) c;
```

FAC

```
 fac n acc == (n=0) -> acc, fac (n-1) (n*acc);
```

SUMN

```
 sumn n acc == (n=0) -> acc, sumn (n-1) (n+acc);
```

TALLY

```
 tally lo hi v total ==
   {result tally_up lo total;
    tally_up lo total == (lo=hi) -> (total=0),
                (tally_up (lo + 1) (total - (sel v lo)))  -> true,
                   (tally_up  (lo + 1) (total + (sel v lo)))};
```

Bibliography

[1] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] L. Allison. *A practical introduction to denotational semantics.* Cambridge University Press, Cambridge, 1986.

[3] Arvind, R.S. Nikhil, and K.P. Keshav. I-structures: data structures for parallel computing. In *Proceedings of the Workshop on Graph Reduction, Los Alamos, New Mexico*, February 1987.

[4] H. Barendregt and M. van Leeuwen. *Functional Programming and the Language TALE.* Technical Report, Mathematical Institute, Netherlands, 1985.

[5] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1:147–164, 1988.

[6] M. Bruynooghe. Compile time garbage collection or how to transform programs in an assignment-free language into code with assignments. In *Program Specification and Transformation*, pages 113–130, International Federation for Information Processing, 1987.

[7] G.L. Burn, C.L. Hankin, and S. Abramsky. *The theory and practice of strictness analysis for higher order functions.* Technical Report DoC 85/6, Imperial College of Science and Technology, Department of Computing, April 1985.

[8] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *LNCS 217: Programs as Data Objects*, pages 42–62, Springer-Verlag, 1985.

[9] A. Church. *The Calculi of Lambda-Conversion.* Volume 6 of *Annals of Mathematical Studies*, Princeton University Press, Princeton, New Jersey, 1951.

[10] C. Clack and S.L. Peyton Jones. Strictness analysis – a practical approach. In *Functional Programming Languages and Computer Architecture*, pages 35–49, Springer-Verlag LNCS 201, September 1985.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, ACM, 1977.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM Sym. on Prin. of Prog. Lang.*, pages 269–282, ACM, 1979.

[13] L. Damas and R. Milner. Principle type schemes for functional languages. In *9th ACM Sym. on Prin. of Prog. Lang.*, ACM, August 1982.

[14] G. Gopalakrishnan and M. Srivas. Implementing functional programs using mutable abstract data types.

[15] K. Gopinath. *Copy elimination in single assignment languages.* PhD thesis, Stanford University, 1988.

[16] K. Gopinath and J. Hennessy. Copy elimination in functional languages. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, January 1989.

[17] P. Hudak. *ALFL Reference Manual and Programmer's Guide.* Research Report YALEU/DCS/RR-322, Second Edition, Yale University, October 1984.

[18] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Symposium On Lisp and Functional Programming*, pages 351–363, ACM, August 1986.

[19] P. Hudak and R. Sundaresh. *On the Expressiveness of Purely Functional I/O Systems.* Technical Report YALEU/DCS/RR665, Yale University, Department of Computer Science, December 1988.

[20] P. Hudak and J. Young. Collecting interpretations of expressions (without powerdomains). In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 107–118, January 1988.

[21] P. Hudak and J. Young. Higher-order strictness analysis for untyped lambda calculus. In *12th ACM Symposium on Principles of Programming Languages*, pages 97–109, January 1986.

[22] J. Hughes. Strictness detection in non-flat domains. In *LNCS 217: Programs as Data Objects*, pages 42–62, Springer-Verlag, 1986.

[23] T. Johnsson. Lambda lifting: transforming programs to recursive equations.

[24] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 58–69, ACM, SIGPLAN Notices 19(6), June 1984.

[25] R. Keller. *FEL Programmer's Guide.* Technical Report, University of Utah, April 1983.

[26] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: an optimizing compiler for Scheme. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, ACM, June 1986. Published as SIGPLAN Notices Vol. 21, No. 7, July 1986.

[27] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, Univ. of Edinburgh, 1981.

[28] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the International Symposium on Programming*, pages 269–281, Springer-Verlag LNCS Vol. 83, 1980.

[29] A Mycroft and N. Jones. *A Relational Framework for Abstract Interpretation*, pages 156–171. Springer-Verlag, 1985.

[30] F. Nielson. *Abstract Interpretation Using Domain Theory.* PhD thesis, University of Edinburgh, October 1984.

[31] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[32] D. Schmidt. *Detecting Stack-Based Environments in Denotational Definitions.* Research Report TR-CS-86-3, Kansas State University, October 1986.

[33] D.A. Schmidt. *Denotational Semantics – A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, Mass., 1986.

[34] D.A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.

[35] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1977.

[36] J. Young. *Theory and Practice of Semantics-Directed Compiling for Functional Programming Languages.* PhD thesis, Yale University, Department of Computer Science, expected 1988.