On Expressing the Mutation of State in a Functional Programming Lanugage

Juan Carlos Guzman
Research Report YALEU/DCS/RR-962
May 1993

This work is partially supported by Darpa N00014-91-J-4043.

On Expressing the Mutation of State in a Functional Programming Language

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Juan Carlos Guzmán

May 1993

Abstract

On Expressing the Mutation of State in a Functional Programming Language

Juan Carlos Guzmán

Yale University

1993

A long standing problem in functional programming languages is how to efficiently manipulate large data structures since the underlying computational model adopted by these languages makes no provision for reutilization of data structures—a small change in a structure results in a completely new object. This might reflect the programmer's intention, but the original structure is usually no longer used after the modification occurs, and thus it can be directly reclaimed in allocating the new object with substantial savings in space and time complexity. However, there is no effective method for detecting precisely which data structures can be reused. The known heuristics are obscure. Even if a program is suitable for optimization, it is up to the compiler whether or not to perform it. But most importantly, the user is unable to *express* the fact that the structure must be reused.

The expressiveness of modern (i.e., higher-order, nonstrict, polymorphic) functional languages can be extended with the ability to destructively manipulate state without losing referential transparency. Such an expansion can be designed in a way that the resulting state semantics is easy to reason about and implies a direct and obvious implementation. Further, information about mutability properties can be

totally reconstructed statically, and this information can be fairly intuitive for the programmer to understand, and thus, a valuable aid in the development of programs that handle state more efficiently.

To that effect, I present Single-Threaded Lambda Calculus—an extension to Lambda Calculus with graphical rewrite rules, a notion of store, and mutators (i.e., *update*!, etc.) to express mutations of state. In addition, I present an extended type system that carries information about operational properties of programs, including mutability. The type of a program is statically decidable by a type reconstruction algorithm. All expressions may exhibit type polymorphism, and are inherently polymorphic in their higher-order operational properties. Further, Single-Threaded Lambda Calculus is confluent when restricted to well-typed programs.

Acknowledgements

I would specially like to thank my advisor Paul Hudak, who—among other things—guided me through graduate school, directed my thesis, *faced* earlier drafts of the dissertation, and was very supportive of my family and myself. I was very lucky indeed in finding such a good person as my advisor.

My readers Ken Yip and Uday Reddy also provided accurate suggestions on style and contents of the dissertation. I appreciate their efforts in making the reviewing process as smooth as possible.

I learned quite a bit of Philosophy of Computer Science by being a teaching assistant of Alan J. Perlis. I consider myself very fortunate to have met and learned from such a brilliant professor.

The "wrestling team" was always eager to debate over my ideas. Much of the current ideas were debugged in Friday morning wrestlings. The team included David Kranz, Ben Goldberg, Richard Kelsey, Adrienne Bloss, Jim Philbin, Steve Anderson, Jonathan Young, Rick Mohr, Zhijing Mou, Duke Briscoe, Amir Kishon, Siau-Cheng Khoo, Raman Sundaresh, Dan Rabin, Tom Blenko, Kung Chen, María Mercedes Guzmán, John Peterson, Charles Consel, and Paul Hudak.

I appreciate very much the help of Linda Joyce, both while at Yale, and after I left. She would always be at the other end of the line making sure the draft printed, Paul's got it, etc., etc., etc. In the world of telecommunications you can work on a remote computer, but you can't be certain on the quality of the remote printed copy!

I also acknowledge the continuing support of the Yale Computer Science Deptartment as a whole. Faculty and staff alike have always been in harmony and ready to help. Thanks to all.

My parents were the first ones to seriously convince me to pursue Ph.D. studies. Further, they always dissipated my intentions of quitting. They are, in fact, *guilty* of me writing this thesis (well, it's my fault, too).

María Mercedes, my wife, also *conspired* with my parents in encouraging me to

continue in the program. We were married under the *curse* of a then future thesis. The curse grew meaner with time. Still, María was always cooperative and extremely understanding of the curse, and helped in many ways to overcome it.

John and Marti Peterson were also very supportive. They would always provide a practical advice whenever I got frustrated with my thesis. I also remember with pleasure the hikes to the woods and their patience in teaching the basics of "rock climbing". Walter and Liz Welsh also made us feel at home. I am grateful of their kindness with my family and myself.

To all my family and friends who visited us during these long years. It was always comforting to have a familiar face around!

Intevep, S.A.[1] also provided incentives to finish. Benjamín Sagalovsky and Kathy Octavio were very kind and provided the needed support. Also, I used the computer facilities of Universidad Simón Bolívar. LDC people were specially helpful.

I was a scholarship recipient from Fundación Gran Mariscal de Ayacucho during part of my stay at Yale. This research was also supported in part by DARPA contract N00014-91-J-4043.

[1] Research and Technological Support Center of Petróleos de Venezuela, S.A,

Contents

List of Figures

Chapter 1

Introduction

In recent years various calculi, logics, and type systems that deal with notions of side-effects, sequencing, and related operational concerns have emerged. The most notable of these are Felleisen's λ-v-CS-calculus [Felleisen, 1988], which captures the essence of languages such as Scheme with side-effects and first-class continuations; Girard's Linear Logic [Girard, 1987], which captures the essence of linear computations; and Gifford's Effect Systems [Lucassen and Gifford, 1988], which capture within a type system notions of side effects and aliasing. My work shares a bit of all these in both motivation and technical detail, but differs in the following way:

My primary goal is to devise a method to express mutations to state in a modern (higher-order, polymorphic, non-strict) functional language, without sacrificing referential transparency, and with a simple, easy to reason about semantics. Although collectively these properties seem contradictory, I will describe a solution that I find quite satisfactory.

Aside from the fundamental property of referential transparency, the two key properties that I wish to maximize are *simplicity* and *expressiveness*. The system must be easy to use: expressing mutations to state should be natural, and the resulting behavior should be easy to reason about. I believe that λ-v-CS-calculus, for example, may be difficult for a programmer to reason about, and linear logic is too constraining

(and thus not natural). Effect systems are much closer to my goal, but the starting point there is an imperative language, whereas mine is functional, and thus the system falls short in meeting certain goals that I will describe shortly.

In this document:

1. I first describe λ_{st}, or *single-threaded lambda calculus*, whose graphical rewrite rules exhibit a certain degree of linearity properties (with respect to reduction strategies). Confluence can be proven for λ_{st} over all reduction strategies.

2. A class of *mutators* for expressing the mutation of state, is introduced, as well as the corresponding *δ-rules* which give the semantics of mutators within λ_{st}. We show that λ_{st} is then no longer confluent.

3. I extend λ_{st} with a *polymorphic type and liability system* which rejects all programs that may not be confluent. I show that the resulting calculus, *poly-*λ_{st}, is confluent yet able to express mutations of state in a natural way.

   This extended type system is the most interesting aspect of my development, and possesses the following properties:

   (a) It is polymorphic in both types and mutability properties of objects. (Indeed it is expressed as an extension of the Hindley-Milner type system.)

   (b) Type reconstruction[1] is decidable.

   (c) Non-destructive operations are permitted in contexts that allow destructive operations (but not vice versa), and thus a *subtype hierarchy* is induced, with coercions permitted between functions in the hierarchy.

The practicallity of my results should be obvious: Programming languages may be designed around poly-λ_{st} that share all of the desirable properties of modern functional languages such as Haskell [Hudak *et al.*, 1992], yet are strictly more expressive

[1] I adopt the term *type reconstruction* to denote the process of inferring types in the absence of explicit type annotations, to avoid confusion with the term *type inference* which is sometimes used in contexts where a certain amount of type information is syntactically available.

in their ability to express mutations of state. This promises to solve a long-standing open problem in programming language research: combining functional and imperative programming techniques within one consistent framework.

1.1 Computation in Functional Languages

Functional programming languages are based on Lambda Calculus—a term rewriting system that formalizes the meaning of *functions* [Barendregt, 1984]. This calculus has a very simple and elegant semantics based on *term* reduction. The operational semantics of functional languages, however, is that of *graph* reduction rather than term reduction. The graph reduction semantics used is 'compatible' with the term reduction semantics implied by the calculus, with the advantage that graph sharing results in shorter reduction sequences to normal form [Wadsworth, 1971]. A λ-expression is a term, and thus has no notion of sharing. Viewed as a graph, a λ-expression is a tree. However, applying graph reduction to λ-terms will eventually develop into shared graphs, with the advantage that since a shared graph corresponds to several terms, reduction of a graph node corresponds to several term reductions. Consider, for example, the λ-term

$$(\lambda x.\ x{*}x)\ ((\lambda y.\ y{+}y)\ 1)$$

Figures 1.1 and 1.2 show a possible term reduction path, and a possible graph reduction path for the above expression. It can be observed there that while the first term reduction duplicated the subterm $((\lambda y.\ y{+}y)\ 1)$, the graph reduction version did not duplicate the term. Instead, the first graph reduction induced sharing—e_1 and e_2 represent the *same* graph. Therefore, unlike in term reduction, reducing e_1 results in e_2 being also reduced. The practicality of the results of graph reduction are such that functional programmers think in terms of graph reduction when reasoning about operational properties of their programs such as sharing, and thus time and

$$
\begin{aligned}
& (\lambda x.\ x{*}x)\ ((\lambda y.\ y{+}y)\ 1) \\
\rightarrow\ & ((\lambda y.\ y{+}y)\ 1){*}((\lambda y.\ y{+}y)\ 1) \\
\rightarrow\ & (1{+}1){*}((\lambda y.\ y{+}y)\ 1) \\
\rightarrow\ & 2{*}((\lambda y.\ y{+}y)\ 1) \\
\rightarrow\ & 2{*}(1{+}1) \\
\rightarrow\ & 2{*}2 \\
\rightarrow\ & 4
\end{aligned}
$$

Figure 1.1: Term Reduction Steps for $(\lambda x.\ x{*}x)\ ((\lambda y.\ y{+}y)\ 1)$

space complexity. For example, the above λ-term, and

$$((\lambda y.\ y{+}y)\ 1){*}((\lambda y.\ y{+}y)\ 1)$$

have the same value—they both reduce to 4—but the former does so in fewer *graph* reductions. Figures 1.2 and 1.3 show a reduction path for each of these λ-terms.

Note that while e_1 and e_2 correspond to the same graph (Figure 1.2), and thus are reduced simultaneously, the reductions of e_1', and e_2' (Figure 1.3) are independent since they do not correspond to the *same* graph, but are simply *isomorphic*—sharing can only be obtained through (graph) reductions.

1.2 State Manipulation—The Problem

One of the 'advantages' of functional languages is that the notion of implicit store (a huge data structure that holds all the values of your program) is absent. All values used by a functional program have to be explicitly given; i.e., state is explicit.

Having no notion of sharing, lambda calculus also lacks the notion of store reusability. This is a disadvantage when manipulating large portions of state, since a small

$$
\begin{aligned}
& (\lambda x.\ x{*}x)\ \overbrace{((\lambda y.\ y{+}y)\ 1)}^{e} \\
\rightarrow\ & \overbrace{((\lambda y.\ y{+}y)\ 1)}^{e_1} * \overbrace{((\lambda y.\ y{+}y)\ 1)}^{e_2} \\
\rightarrow\ & (1{+}1){*}(1{+}1) \\
\rightarrow\ & 2{*}2 \\
\rightarrow\ & 4
\end{aligned}
$$

Figure 1.2: Graph Reduction Steps for $(\lambda x.\ x{*}x)\ ((\lambda y.\ y{+}y)\ 1)$

$$
\begin{aligned}
& \overbrace{((\lambda y.\ y{+}y)\ 1)}^{e_1'} * \overbrace{((\lambda y.\ y{+}y)\ 1)}^{e_2'} \\
\rightarrow\ & (1{+}1){*}((\lambda y.\ y{+}y)\ 1) \\
\rightarrow\ & 2{*}((\lambda y.\ y{+}y)\ 1) \\
\rightarrow\ & 2{*}(1{+}1) \\
\rightarrow\ & 2{*}2 \\
\rightarrow\ & 4
\end{aligned}
$$

Figure 1.3: Graph Reduction Steps for $((\lambda y.\ y{+}y)\ 1){*}((\lambda y.\ y{+}y)\ 1)$

modification of a large data structure may represent a duplication of the data. Efficient manipulation of large data structures has been a long standing problem in functional languages. Solutions have been proposed mainly along the lines of abstract interpretation, like the ones in [Hudak, 1986,Bloss, 1989].

The kind of large data structures that I will be considering throughtout this dissertation is arrays, along with a reduced set of primitives: a constructor `mkarray`—`(mkarray i x)` creates an array of `i` elements, all of them having the value `x`—an update operation `update`—`(update a i x)` is an array similar to `a` except that element `i` has value `x`—and an element lookup operation `lookup`—`(lookup a i)` returns the `i`th element of the array.

In general, the value of the array before the update remains available even after the operation is done. In the worst case, (part of) the updated array must be a copy of the original. This becomes very expensive both in additional time and space needed for the copies, and possibly in array access time when updates are used since they tend to be used for *every* element of the structure. Further, there are data structures, like files, or communication channels, that cannot be duplicated. In any case, it more often happens that the value of the structure before the update is in fact not used anymore after the update is performed, in which case *reusing* the now inaccessible data structure to hold the new updated value would make more sense. Clever implementations, of course, can improve the efficiency greatly, and indeed this has been a rather popular research topic in recent years. The most successful implementation will manage to perform all such updates by re-using the old array and performing the updates *destructively.*

But regardless of how successful such optimization techniques might be, the problem is that a user has no simple way to *reason* about the efficiency, unless one has perfect knowledge of the optimization methods being used in a particular implementation.

1.3 The Proposed Solution

The solution that I advocate is to allow the programmer to specify *explicitly* that a particular update is to be done destructively (just as one would in an imperative language), but at the same time not have to worry about destroying referential transparency. However, the destructive updates cannot be expressed in the lambda calculus, even with the graph reduction semantics.

Directly providing an operator `update!` similar to `update` but that would perform an update *in place* (modify the structure itself rather than making the modification in a copy) would result in a non-confluent calculus. For example, the value of the following program would depend on the reduction order:

```
nonconfluent i =
    let a = mkarray i 0
    let x = lookup (update! a i 1) i
    in (lookup a i) + x
```

It's value would be 2 if `update! a i` was evaluated before `lookup a i`, and 1 otherwise.

The previous example shows one instance where a destructive operation would result in a *referentially opaque* program. The value of (part of) `a` is allowed to change, and that value change is noticeable: `(lookup a i)` has access to the value of `a` *after it has been destructively updated.* In this thesis I am concerned only on destructive updates that result in referentially transparent programs. A necessary condition to ensure referential trasnparency is that arrays that are destructively updated are not shared at the time they are updated. To allow data reutilization while retaining referential transparency I will need five things:

- a paradigm where destructive operations can be expressed—λ_{st}-calculus,
- a destructive version of `update` (let's call it `update!`),

- a copy operation that creates an unshared copy of the array,
- a way to sequence destructive operations (I will use `let*`, similar to a `let` expressions but with a strict sequential semantics), and
- a type system to ensure that the programs are "safe."

As an example, consider a function that swaps two elements in an array. Using the non-destructive update, it could be written as:

```
swap a i j =
    let x = lookup a i
        y = lookup a j
    in update (update a i y) j x
```

In the proposed type system, simply changing the `update`'s to `update!`'s will result in an ill-typed program, because in general there is no guarantee that the lookup's will be done prior to the updates. However, if in addition, `let` is changed to `let*`, the following well-typed program is obtained:

```
swap! a i j =
      let* x = (lookup a i)
           y = (lookup a j)
      in update! (update! a i y) j x
```

This program is simple, easy to reason about, and, for arrays of immutable types, has the same efficiency as its imperative counterpart. Furthermore, my type system will infer the type of `swap!` to be:

$$Array\ \tau \xrightarrow{ws} Int \xrightarrow{rs} Int \xrightarrow{cs} Array\ \tau$$

The ws above the first arrow indicates that `swap` "writes" its first argument, the rs above the second arrow indicates that it only "reads" the second argument (but the argument is not part of the result), and the cs above the third arrow indicates that

the value of the third argument is "captured" as part of the result. The type variable τ represents any type. Using this information the type sytem will ensure that `swap` is only used in single-threaded contexts. For example, this expression will be ill-typed:

```
(swap! a i x, swap! a j y)
```

since `swap!` mutates its first argument and in this context must do so in two incompatible ways. On the other hand, this use is well-typed:

```
if (lookup a i x)
then (swap! a i j)
else (swap! a i k)
```

since the type system "knows about" conditionals.

1.4 Related Work

Girard's *linear logic* [Girard, 1987] is a constructive logic designed to capture linear computations: variables can be used only once; not more, not less. Its advantages are that space can always be reused, and in fact run-time garbage collection can be avoided entirely. Its main disadvantage is that using variables non-linearly amounts to explicit copying of its value, and it does not adequately distinguish the destructive accesses from non-destructive ones. I feel that the resulting programs are not at all natural, at least with regard to what we are used to in modern functional languages. Wadler [Wadler, 1989] has generalized Girard's system somewhat, but the fundamental limitations of linear logic remain.

Gifford and Lucassen's *polymorphic effect systems* [Lucassen and Gifford, 1988] are targeted for an imperative language such as Scheme that has a purely functional sublanguage; the goal is to distinguish levels of "imperativeness" within a program. There are important differences between my work and theirs, ranging from the nature of the language (mine is functional, theirs is imperative), to the ability to control where mutations happen (we control variables (objects), they control "regions" (heaps)).

Felleisen's λ-v-CS-calculus [Felleisen, 1988] was designed to reason about programs in Scheme-like languages. The approach is to develop a calculus that models the two most salient features of Scheme: side-effects and continuations. It is notable in the achievement of a *calculus* to reason about state, and in that sense shares much with the calculus presented here. However, whereas λ-v-CS-calculus is used to reason about *unrestricted* side-effects in an imperative language, poly-λ_{st} is used to reason about a *disciplined* form of side-effects in a purely functional language. I hope that the latter will prove easier, while at the same time providing enough expressiveness to enable natural manipulations to state.

This work has also spawned the work of Odersky[Odersky, 1991] where a richer set of operational properties was used to more precisely abstract the behavior of mutation in a Functional Language.

1.5 Thesis Organization

Chapter 2 presents an overview of Lambda Calculus and Graph Reduction. Single-Threaded Lambda Calculus (λ_{st}-calculus) is introduced in Chapter 3. This calculus enables the user to reason about sharing and mutation simultaneously. Also in Chapter 3, the operational properties of λ_{st}-calculus that are of interest for this dissertation are presented. The Extended Type System, a type system polymorphic in types, and abstract properties of programs is presented in Chapter 4. A soundness proof relating the type system's inferred types to the actual operational behavior of the program is also provided in that chapter. Several examples of actual programs typed with the system are shown in Chapter 5. These present the reader how problems can be attacked using the typed λ_{st}-calculus. Finally, a type reconstructor algorithm is given in Chapter 6, and an actual implementation in Chapter 7. The algorithm is proven guaranteed to compute the principal type for any legal λ_{st}-expression for which a type can be inferred.

1.6 A Word on Referential Transparency

Referential transparency has long been a pretext for declarative programming. The premise *replace equals by equals* has been the driving force for functional programming: side effects are considered harmful, since they do not satisfy the premise.

Paradoxically, this notion, which is central to all declarative programming is not precisely defined anywhere. Most people have some intuition of what it means, but cannot describe it precisely. The following quotations are two attempts at characterizing what referential transparency means:

> "... each expression denotes a single value which cannot be changed by evaluating the expression or by allowing different parts of a program to share the expression. Evaluation of the expression simply changes the form of the expression but never its value. All references are therefore equivalent to the value itself and the fact that the expression may be referred to from other parts of the program is of no concern." [Field and Harrison, 1988]

> "... The most important feature of mathematical notation is that an expression is used solely to describe (or *denote*) a *value*. ... Furthermore, the value of an expression depends only on the value of its constituent expressions (if any), and these subexpressions may be replaced freely by others possessing the same value. ... every mathematician understands that variables do *not* vary: they always denote the same quantity, provided we remain within the same context of definitions associated with them ..." [Bird and Wadler, 1988]

In the above quotations, the word *value* represents the meaning of the expression under the semantics in which it is being considered—usually, the so called standard semantics. However, *value* may well represent the meaning under the static

semantics—its type—or its complexity, or the meaning under any other semantics. I will illustrate this by example:

"Untyped λ-calculus" has been the language of choice for many advocates of referential transparency. The expressions

```
length [1,2] + length ['a','b']
```

and

```
f length
where f g = g [1,2] + g['a','b']
```

are interchangeable.

However, with the introduction of types and the design of the Hindley-Milner type system [Hindley, 1978,Milner, 1978], the above expressions are no longer interchangeable: The type property—static semantics—of these expressions differ (the latter expression does not type check), although their most important characteristic—their dynamic semantics—remains the same.

On the other hand, consider the expressions

```
f x + f x
```

and

```
y + y where
    y = f x
```

and their operational semantics in a typical functional programming language. The static types for these expressions are the same but they are not operationally interchangeable because the complexity of the latter is less.

Then, as these examples point out, missing from the previous quotations is the notion of semantics. Referential transparency needs to be defined in terms of the semantics under consideration, since a replacement rule may be referentially transparent

under one semantics, and opaque under another. Undoubtedly, the most important semantics under which referential transparency is desired is the standard semantics. But it is also true that transparency is desired for other semantics of interest.

Alternative semantics deal with aspects of the computation which are not captured by the standard semantics. Type disciplines, strictness properties, partial evaluation strategies have all been developed as alternative semantics. All of them attack problems which are at best semidecidable in their pure form, but, because these semantics are used as compilation tools, the problems have to be simplified into decidable, and even into polynomial heuristics. An interesting trade-off appears then, when deciding which aspects of the semantics to simplify in order to make it decidable, while retaining the spirit of the original property. The trade-off is typically solved by imposing additional restrictions that guarantee a decidable semantics at the expense of weakening other properties like referential transparency. In the first example above, the decision to restrict the domain of types to the so called *shallow types*—quantifiers only permitted at the outermost level—led both to a semantics that can be computed effectively, and to the opacity of the example. In this respect, the Hindley-Milner type system has been criticized for the special treatment of the types of let-bound variables. It has survived as a practical polymorphic type system for two reasons:

- the locality and simplicity of the opacity—transformation between λ- and let-variables are not transparent, and
- the opacity itself is the result of a trade-off which was solved in an ingenuous way.

In my thesis, I address another facet of referential transparency—the opacity of replacements under operational semantics, while maintaining transparency under the standard semantics. Further, in my work, I will point out objects that are operationally distinct, but indistinguishable in terms of the standard semantics.

Chapter 2

Lambda-Calculus and Graph Reduction

2.1 Introduction

Lambda Calculus (λ-calculus) is widely recognized as the operational semantics of functional languages. In turn, implementations of λ-calculus are usually based on Graph Reduction (GR), which provides an efficient way to transform expressions into normal form. However, an essential difference between these two models of computation, is that object sharing is present in graph reduction, but it is absent in λ-calculus. Graphs in GR constitute not only the program, but also the store while expressions in λ-calculus lack store. In fact, λ-expressions are represented as *trees* in λ-calculus. Sharing is only an operational notion arising from the way expressions in the calculus are implemented, and various implementations of the λ-calculus are free to introduce sharing for efficiency considerations. In this respect, a single reduction on a shared graph corresponds to multiple reductions on the "corresponding" λ-expression.

This chapter presents both paradigms, along with basic results that will be needed in later chapters. Lambda calculus is developed in Section 2.2, and graph reduction

$$
\begin{array}{rcll}
f, v & \in & Ide & \textit{identifiers} \\
e & \in & Exp & \textit{expressions} \\
\textit{where } e & ::= & v & \textit{identifiers} \\
 & & \mid (\lambda v.e) & \lambda\textit{-abstractions} \\
 & & \mid (e_1\ e_2) & \textit{applications}
\end{array}
$$

Figure 2.1: Lambda Calculus Syntax

is introduced in Section 2.3. Readers familiar with either formalism may wish to skip to the next chapter. In any event, readers should read the introductory part of Section 2.3 since it introduces the notation for graph reduction used throughout this dissertation.

2.2 Lambda Calculus—λ

In this section, I do a quick review of Lambda Calculus without, and with constants: their syntax, reduction rules, and some of their most important properties, including the Church-Rosser property.

2.2.1 "Pure" Lambda Calculus

Pure λ-calculus is a language of identifiers, λ-abstractions, and applications. Its syntax is shown in Figure 2.1.

In applications $(e_1\ e_2)$, e_1 is called the *function*, and e_2 the *argument*. In abstractions $(\lambda v.e)$, v is called the *bound variable* or *bound identifier*, and e the *body* of the abstraction. It is said that v *is bound by* $(\lambda v.e)$, or that $(\lambda v.e)$ *binds* v. An occurrence of an identifier is said to be *bound* if there is an enclosing λ-abstraction binding that identifier, and it is said to be *free* otherwise. A variable v is free in e if

$$
\begin{array}{rcl}
\_[\_/\_]: & & Exp \rightarrow Exp \rightarrow Var \rightarrow Exp \\
k[e'/v] & = & k \\
v[e'/v] & = & e' \\
v_1[e'/v_2] & = & v_1 \\
(\lambda v.e)[e'/v] & = & (\lambda v.e) \\
(\lambda v_1.e)[e'/v_2] & = & (\lambda v.e[e'/v_2]) \\
(e_1\ e_2)[e'/v] & = & (e_1[e'/v]\ e_2[e'/v]))
\end{array}
$$

Figure 2.2: Substitution in λ-Calculus

there is a free occurrence of v in e. A λ-expression is *closed* if it does not contain any free occurrences of any variable. The notation $C[\ ]$ will be used to indicate a *context with holes*—a λ-expression with missing subexpressions. $C[e]$ is the expression that results from filling all the "holes" of C with expression e.

Variable substitution $e[e_1/v]$ is defined in the λ-calculus in the usual way. However, care must be taken in order not to inadvertently bind any free variable of e' in e. It is assumed that free variables of e' are not bound in e. Its definition appears in Figure 2.2.

Binary relations can be defined on the expressions of this language, also known as reduction rules. They are rules of the form

$$e_1 \rightarrow e_2$$

$$\beta: \quad ((\lambda v.e_1)\ e_2) \quad \rightarrow \quad e_1[e_2/v]$$

Figure 2.3: β-Reduction Rule for the λ-Calculus

and indicate that any expression $C[e_1]$ (any expression where e_1 occurs as a subexpression) can be *reduced* to $C[e_2]$. The β-reduction rule is one such relation, and is defined in Figure 2.3. It states that any application for which the function is a λ-abstraction can be replaced by the expression resulting from substituting all free occurrences of v inside e_1 by e_2 (i.e., $e_1[e_2/v]$). The occurrence of e_1—a subexpression of the form $((\lambda v.e_1)\ e_2)$ for β—is called a *redex* (shorthand for *red*ucible *ex*pressions), and the resulting subexpression, e_2 is called the *contractum.* Computation proceeds in this calculus by successive applications of the reductions rules.

Strictly speaking, $e_1 \rightarrow^1_\beta e_2$ if e_2 is obtained from e_1 by one application of the β-rule (*one-step β-reduction*), $e_1 \rightarrow^*_\beta e_2$ notates the reflexive, and transitive closure (*β-reduction*), and $e_1 \leftrightarrow^*_\beta e_2$ denotes the reflexive, symmetric, and transitive closure, also known as *β-conversion.* Redexes and contracta are usually denoted by Δ. As such, $C[\Delta]$ is an expression with redex Δ. Reductions are characterized by the redex they reduce. Therefore, reductions will sometimes be written as $e_1 \rightarrow^1_\Delta e_2$ if $e_1 = C[\Delta]$, $e_2 = C[\Delta']$, and $\Delta \rightarrow_\beta \Delta'$. A *reduction path* is a sequence of reductions $e_0 \rightarrow_\beta e_1 \rightarrow_\beta e_2 \rightarrow_\beta \ldots$. Reduction paths can also be specified by just giving the sequence of redexes being selected for reduction. Thus $e_0 \rightarrow_{\Delta_1\Delta_2\ldots}$ denotes the reduction path

$$e_0 \rightarrow_{\Delta_1} e_1 \rightarrow_{\Delta_2} e_2 \rightarrow_{\Delta_3} \cdots$$

This reduction sequence may be of finite, or infinite length. A reduction path is *finite* if it is of finite length, and it is infinite otherwise. Figure 2.5 shows Ω, a λ-expression that has an infinite reduction path.

An expression is said to be in *normal form* if the β-reduction rule cannot be applied to any of its subexpressions. Normal forms are considered the result of computation of expressions; i.e., their *values.* Figure 2.4 shows several examples of λ-expressions. There, 1 represents the *Church Numeral* 1, a λ-calculus encoding for number 1. Similarly 2 represents Church Numeral 2, and *succ* is the λ-expression that implements the successor function. These three expressions are already in normal form. In that figure, a series of reductions show that the normal form of (*succ* 1) is convertible

$$
\begin{array}{rcl}
1 & \equiv & \lambda x.\lambda y.(x\ y) \\
2 & \equiv & \lambda x.\lambda y.(x\ x\ y) \\
succ & \equiv & \lambda a.\lambda b.\lambda c.(b\ (a\ b\ c)) \\
(succ\ 1) & \equiv & (\lambda a.\lambda b.\lambda c.(b\ (a\ b\ c))\ \lambda x.\lambda y.(x\ y)) \\
 & \rightarrow_\beta & \lambda b.\lambda c.(b\ (\lambda x.\lambda y.(x\ y)\ b\ c)) \\
 & \rightarrow_\beta & \lambda b.\lambda c.(b\ (\lambda y.(b\ y)\ c)) \\
 & \rightarrow_\beta & \lambda b.\lambda c.(b\ (b\ c)) \\
 & \rightarrow_\alpha^* & 2
\end{array}
$$

Figure 2.4: A λ-expression and its Reduction to Normal Form

to 2. The expression Ω (Figure 2.5) has no normal form: it only contains a redex—the whole expression—which reduces to itself. There are also expressions which may have both finite and infinite reduction paths. That is the case of $(\lambda x.\lambda y.y)\ \Omega$, which is shown in Figure 2.6.

2.2.2 Residuals

Sometimes it will be necessary to keep track on how a particular subexpression e is affected by a reduction path $\Delta_1 \ldots \Delta_n$ starting with the expression $C[e]$. It will usually be the case that e will be transformed somehow. Maybe Δ_i reduced a subexpression of e, or it was duplicated, and each of its copies was affected in a different way. A way to control how e changes, is by marking it with an identifier that will perdure the reduction path. By examining the resulting expression e', and looking for the mark, it can be determined which parts of the new expression came from e. This marking must satisfy that it cannot alter the reduction process in any way. In order to be able to provide such a marking scheme, a modification to the λ-calculus needs to be done. This modification affects both expressions (the markings), and reductions (how to

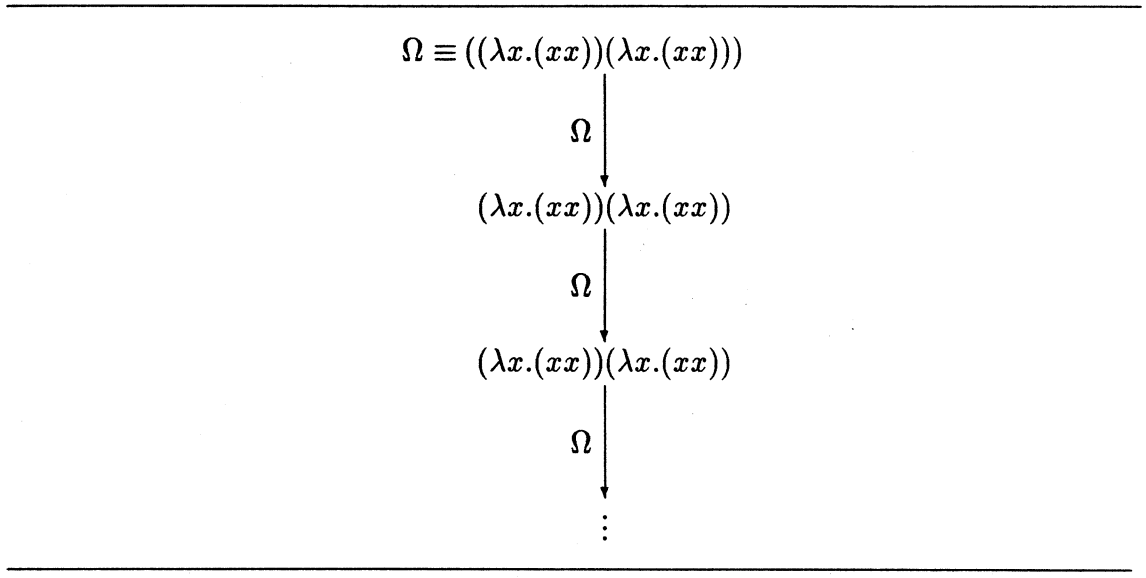


Figure 2.5: Expression with Infinite Reduction Paths

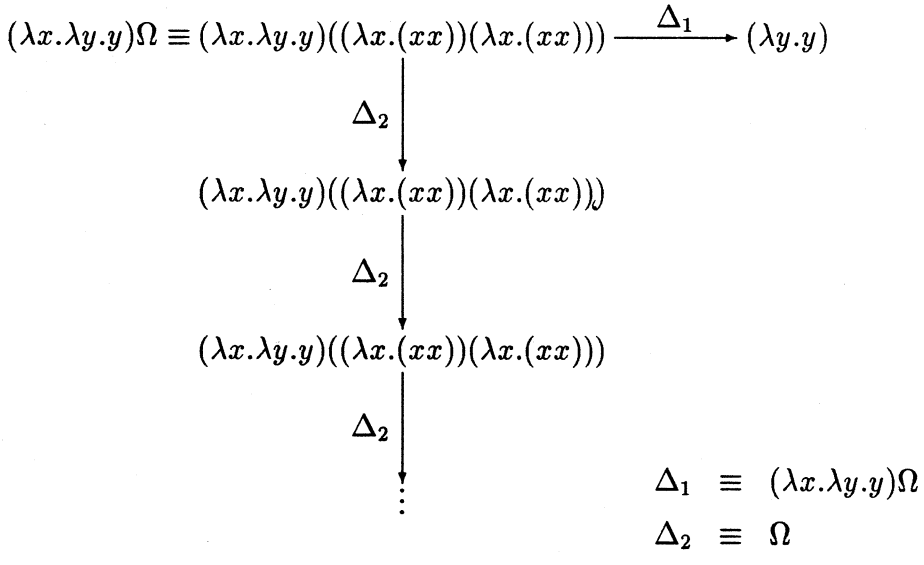


Figure 2.6: Expression with Finite and Infinite Reduction Paths

reduce marked expressions). The marks are done by annotating the expression with an integer index, as follows:

$$\begin{aligned} e \ ::= \ & \ldots \\ & \mid v_i \\ & \mid (\lambda v.e)_i \\ & \mid (e_1 \ e_2)_i \end{aligned}$$

An expression is *indexed* if its main node has an index. The notation e_* will be used to refer to an expression without indicating whether or not it is indexed. The reduction rule β is modified to β_*, which reduce redexes with, and without indices.

$$\beta_* : \ ((\lambda v.e_1)_* \ e_2)_* \ \rightarrow \ e_1[e_2/v]$$

Note that no new indices are generated by the application of the β_*-reduction rule. Rather, it consumes indexed expression nodes.

Let $e, C[e] \in Exp$ *Lift* $C[e]$ to $C[e']$ by replacing the occurrences of e with an expression were all abstractions, applications, constants, and variables have been tagged with tag 1. The *residuals* of e relative to $\Delta_1 \ldots \Delta_n$ is the set of occurrences of maximal indexed expressions. This definition can be extended to control the residuals of a set of expressions.

2.2.3 Confluence in λ-Calculus

As mentioned before, computation in λ-calculus is performed by means of β-reduction—expressions are reduced until normal form is reached. If a normal form is obtained, then it is the *value* of the original expression. There is no guarantee that an expression can be reduced to normal form, and even if it can, Figure 2.6 showed that there can still be reduction paths that do not lead to normal form. Further, there are expressions with more than one redex; Can there be expressions with more than one normal form? Can it be guaranteed that, if a normal form can be reached, there will be a way of reducing to it?

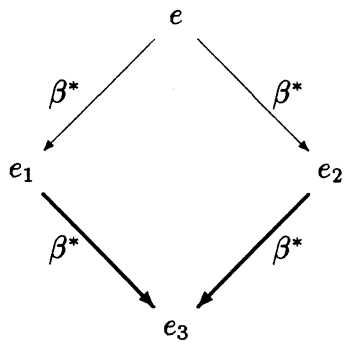


Figure 2.7: Church-Rosser Theorem for λ-Calculus

These questions address the issue of *confluence.* Confluence is regarded as a desirable property for any deterministic calculus. It states that any expression has at most one normal form. As such, all reduction paths leading to normal form will result in the *same* normal form. Fortunately, there is a landmark result in the λ-calculus that guarantees confluence. It is known as the *Church-Rosser Theorem*:

Theorem 2.2.1 Church-Rosser Theorem, *[Barendregt, 1984, Theorem 3.2.8].*

1. *For all λ-expressions* e, e_1, *and* e_2, *if* $e \rightarrow_\beta^* e_1$, *and* $e \rightarrow_\beta^* e_2$, *then there exists* e_3 *such that* $e_1 \rightarrow_\beta^* e_3$, *and* $e_2 \rightarrow_\beta^* e_3$.

2. *If* $e_1 \leftrightarrow_\beta^* e_2$ *then there exists* e_3 *such that* $e_1 \rightarrow_\beta^* e_3$, *and* $e_2 \rightarrow_\beta^* e_3$.

Figure 2.7 shows a diagram that summarizes this result. Confluence is guaranteed as a corollary to the previous theorem:

Corollary 2.2.2 *[Barendregt, 1984, Corollary 3.2.9].*

1. *For all* e_1 *and* e_2, *if* e_2 *is a normal form of* e_1, *then* $e_1 \rightarrow_\beta^* e_2$.

2. *For all* e, e *can have at most one normal form.*

It is often the case that a λ-expression has several redexes. By virtue of the Church-Rosser Theorem, it does not matter which redex is reduced, the normal form for that expression can always be found if it exists.

In order to control how computation is performed, *reduction strategies* are established. These are "selection policies" that limit the number of possible redexes to reduce. Two of the most important strategies are *normal-order* reduction which for any expression, the next redex to reduce is always the one that starts *leftmost*, and *applicative-order* reduction, in which the redex to choose is the one that starts *rightmost*. A second version of the Church-Rosser Theorem establishes the fact that if an expression has a normal form, it can be obtained by following the normal-order reduction strategy (hence its name).

The reader is encouraged to read [Barendregt, 1984] for a throrough development of the theory of the λ-calculus.

2.2.4 Lambda Calculus with Constants

A very important variant of the Lambda Calculus is a version which allows the use of constants, or atoms—objects of known semantics, such as the natural numbers, and arithmetic functions on them. Constants are used pragmatically wherever it is more practical to encode values as known objects, rather than in λ-expressions, as is the case of natural numbers. Also, functional languages are not really based on pure λ-calculus, but on this variant due to implementation issues including efficiency, and closer machine representation of constants and functions on them. The syntax of the λ-calculus version that I will be using throughout the remainder of the thesis is presented in Figure 2.8. Among the member of the constants' set Kon are the boolean constants *true* and $false$, the numbers and arithmetic operations, the conditional function if, the array constant $Array$, the array constructor mka, the lookup function on arrays *lookup* the update function on arrays *update*, and the fixpoint operator fix, with the usual reduction rules. I use the convention that $(Array\ {}^1e_1 \ldots {}^ie_i \ldots {}^ne_n)$

$$
\begin{array}{rcll}
f, v & \in & Ide & \textit{identifiers} \\
k & \in & Kon & \textit{constants} \\
i & \in & Int \subset Kon & \textit{integers} \\
 & & & \\
e & \in & Exp & \textit{expressions} \\
\textit{where } e & ::= & k & \textit{constants} \\
 & & \mid v & \textit{identifiers} \\
 & & \mid (\lambda v.e) & \lambda\textit{-abstractions} \\
 & & \mid (e_1\ e_2) & \textit{applications}
\end{array}
$$

Figure 2.8: Syntax of Lambda Calculus with Constants

$$
\begin{array}{lrcl}
\beta & ((\lambda v.e_1)\ e_2) & \rightarrow & e_1[e_2/v] \\
\delta_{if} & (if\ true\ e_1\ e_2) & \rightarrow & e_1 \\
 & (if\ false\ e_1\ e_2) & \rightarrow & e_2 \\
\delta_{mka} & (mka\ i\ e) & \rightarrow & (Array\ {}^1e\ \ldots\ {}^ie) \\
\delta_{update} & (update\ (Array\ {}^1e_1 \ldots {}^ie_i \ldots {}^ne_n)\ i\ e') & \rightarrow & (Array\ {}^1e_1 \ldots {}^ie' \ldots {}^ne_n) \\
\delta_{fix} & (fix\ f\ e) & \rightarrow & e[(fix\ f\ e)/f]
\end{array}
$$

Figure 2.9: Reduction Rules for the Lambda Calculus with Constants

$$((\lambda x.((+\ x)\ 1))\ 1)\ \rightarrow_{\beta}\ ((+\ 1)\ 1)\ \rightarrow_{\delta_+}\ 2$$

Figure 2.10: A λ-expression with Constants and Its Reduction to Normal Form

denotes an array of n elements, with $\overset{i}{e}$ being the ith element.

The reduction rules for this calculus appear in Figure 2.9, and include the traditional β-rule, as well as several δ-rules that encode the operation of constant functions.

A sufficient condition for this calculus to remain Church-Rosser is that the δ-rules satisfy the condition that at most one of them can be applied at any single redex (the rules are disjoint), and that they are closed under reduction and substitution. This is guaranteed by [Barendregt, 1984, pp 401, Theorem 15.3.3 (Mitschke 1976)]. As in the case of pure λ-calculus, normal-order, and applicative-order reduction strategies are defined to control reduction to normal form. Figure 2.10 shows a λ-expression and its reduction to normal form.

2.3 Graph Reduction

In this section, I provide an overview of graph reduction (GR), a modification to the λ-calculus that formalizes sharing properties among expressions (alternatively called *graphs*). Embedded in the calculus is a notion of location, although no formal store is introduced. Hence, it is possible in this calculus not only to reason about the value of a graph (or its normal form), but also to reason about the sharing properties among its components. Its reduction rules *are* expressed as *graph* reduction rules, rather than *term* reduction rules. Its syntax is introduced in Figure 2.11. The graphs just introduced differ from λ-expressions in that all sub-expressions are labelled; the notation e^{ℓ} attaches label ℓ to expression e. The intended operational semantics of these labels is to represent sharing: expressions with the same label represent the

$$
\begin{array}{rcll}
f, v & \in & Ide & \textit{identifiers} \\
k & \in & Kon & \textit{constants} \\
i & \in & Int \subset Kon & \textit{integers} \\
\ell & \in & Label = Int & \textit{labels} \\
 & & & \\
g & \in & Graph & \textit{graphs} \\
where\ g & ::= & k^{\ell} & \textit{constants} \\
 & & \mid v^{\ell} & \textit{identifiers} \\
 & & \mid (\lambda v.g)^{\ell} & \lambda\textit{-abstractions} \\
 & & \mid (g_1\ g_2)^{\ell} & \textit{applications}
\end{array}
$$

Figure 2.11: Syntax of Graph Reduction

same object. Operationally, this means that all occurrences of expressions with the same label must be reduced at the same time. Several restrictions are needed to guarantee that this property is satisfied across reductions. These are

1. graphs may only have a finite number of nodes;
2. if two nodes are labelled the same, then they have to be *consistent*:
   - they have to be isomorphic, and
   - the labels of the corresponding subgraphs must match;
3. the binding of a variable node must be consistent throughout the graph; to that effect, a variable node cannot be bound in more than one place, and all variables nodes with the same label must be bound at the same place, or else they all must be free.

The first two restrictions imply that a node cannot be labelled the same as one of its predecesor, thus disallowing any circularity; i.e., these graphs are really finite

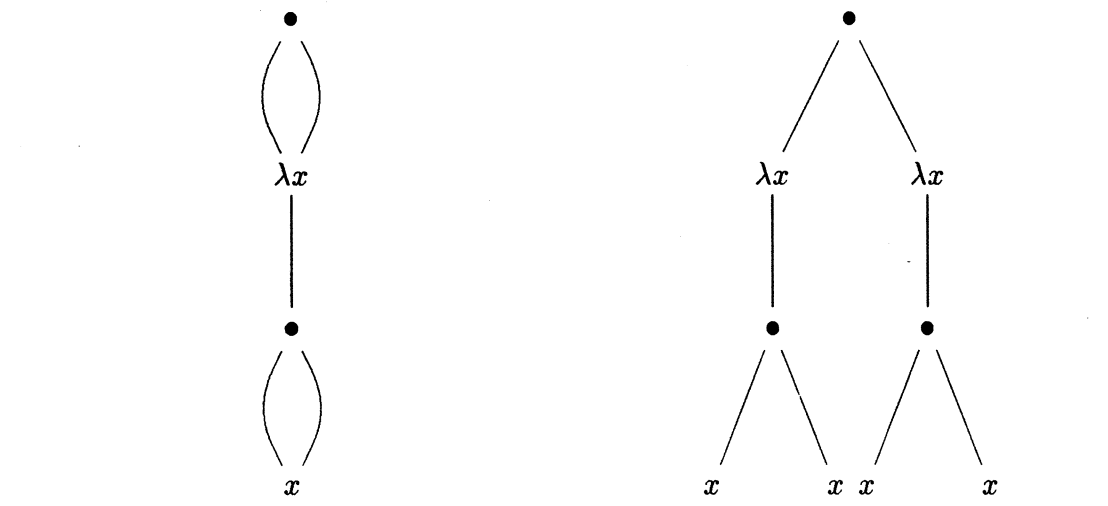


Figure 2.12: Two Different Graphs and Their Notations

DAGs. Graphs that satisfy these restrictions are called *well-formed graphs.* Graphical representation is only possible for graphs that satisfy the first two restrictions. In graphical representations there is no duplication of nodes with the same label—labels are actually omitted. The third restriction imposes a discipline on sharing of DAGs: a DAG with free variables can be shared as long as the variables remain free. A similar restriction was first introduced by Wadsworth in [Wadsworth, 1971, chapter 3]. Well-formed graphs that satisfy restriction 3 are called *admissible*, and *inadmissible* otherwise.

Figure 2.12 shows two graphs that differ only in their sharing properties. There you can appreciate the difference between the alternative notations.

Let me first introduce a series of primitives that will prove useful in manipulating graphs. Their formal definitions appear in Figure 2.13. For an arbitrary graph g, the label set $(lab\ g)$ is the collection of all labels that appear in g, the labelling relation $\mathcal{L}$ is a mapping of labels to sets of subgraphs of g labeled by ℓ (ℓ is the label of the root of the subgraph), and $\mathcal{B}$ associates the labels of every variable node with the labels of

the lambda nodes where such variables are bound. There is also a special value ***free***, not in the domain of labels, which is associated with variables that are not bound. This is called the *binder* relation, since it associates each variable with its 'binder'.

Note that for any domains D, D_1, and D_2, the domain $D \rightarrow 2^{D_1 \times D_2}$ (the domain of functions from D to sets of pairs (d_1, d_2), $d_1 \in D_1$, $d_2 \in D_2$) is isomorphic to the domain $D \rightarrow D_1 \rightarrow 2^{D_2}$ (the domain of functions that map objects from D to functions from D_1 to subsets of D_2). The following function $\mathcal{F}$ is an isomorphism between them:

$$\begin{aligned} \mathcal{F}: &\quad (D \rightarrow 2^{D_1 \times D_2}) \rightarrow D \rightarrow D_1 \rightarrow 2^{D_2} \\ \mathcal{F}\ g &= h \textit{ such that } \forall d \in D,\ d_1 \in D_1,\ d_2 \in D_2 \\ &\qquad (d_1, d_2) \in (g\ d) \iff d_2 \in (h\ d\ d_1) \end{aligned}$$

Hence, $(\mathcal{L}\ g)$ is a set of tuples, or a relation between labels and graphs, whereas $(\mathcal{L}\ g\ \ell)$ is the set of subgraphs to which ℓ is related to. Also, $|D|$ is the cardinality of the set D.

Well-formed graphs are precisely those for which the subgraphs associated with any two occurrences of the same label must be identical; i.e.,

$$\forall \ell \in (lab\ g)\ |(\mathcal{L}\ g\ \ell)| = 1$$

for these graphs, the labelling function $\vec{\mathcal{L}}$ is defined as

$$\begin{aligned} \vec{\mathcal{L}}: &\quad Graph \rightarrow Label \rightarrow Graph \\ \vec{\mathcal{L}}\ g\ \ell &= g' \ iff\ (\ell, g') \in (\mathcal{L}\ g) \end{aligned}$$

When a graph does not satisfy the property, then the graph is *ill-formed.*

Admissible graphs [Wadsworth, 1971] are well-formed graphs for which each variable node has exactly one binder;[1] i.e.,

$$\forall \ell \in (lab\ g)\ |(\mathcal{B}g\ \ell)| \leq 1$$

[1]Wadsworth's binder relation associates *variable names* with places in the graph where they are bound. The notion of binder relation introduced here is slightly different: it associates labels of variables with points in the graph where the variables are bound. Labels of non-variable nodes are associated with the empty set ($\varnothing$).

$$
\begin{array}{rcl}
lab: & Graph \rightarrow 2^{Label} & \\
\mathcal{L}: & Graph \rightarrow Label \rightarrow 2^{Graph} & \\
 & \equiv Graph \rightarrow 2^{Label \times Graph} & \\
\mathcal{B}: & Graph \rightarrow Label \rightarrow 2^{Label \bigcup \{\mathbf{free}\}} & \\
 & \equiv Graph \rightarrow 2^{Label \times\ (Label \bigcup \{\mathbf{free}\})} &
\end{array}
$$

$$
\begin{array}{rcl}
lab\ k^{\ell} & = & \{\ell\} \\
lab\ v^{\ell} & = & \{\ell\} \\
lab\ (\lambda v.g)^{\ell} & = & \{\ell\} \bigcup (lab\ g) \\
lab\ (g_1\ g_2)^{\ell} & = & \{\ell\} \bigcup (lab\ g_1) \bigcup (lab\ g_2)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{L}\ k^{\ell} & = & \{(\ell, k^{\ell})\} \\
\mathcal{L}\ v^{\ell} & = & \{(\ell, v^{\ell})\} \\
\mathcal{L}\ (\lambda v.g)^{\ell} & = & \{(\ell, (\lambda v.g)^{\ell})\} \bigcup (\mathcal{L} g) \\
\mathcal{L}\ (g_1\ g_2)^{\ell} & = & \{(\ell, (g_1\ g_2)^{\ell})\} \bigcup (\mathcal{L} g_1) \bigcup (\mathcal{L} g_2)
\end{array}
$$

$$
\mathcal{B}\ g = \mathcal{B}'\ g\ (\lambda v.\mathbf{free})
$$

$$
\begin{array}{rcl}
\mathcal{B}'\ k^{\ell}\ env & = & \varnothing \\
\mathcal{B}'\ v^{\ell}\ env & = & \{(\ell, env\ v)\} \\
\mathcal{B}'\ (\lambda v.g)^{\ell}\ env & = & \mathcal{B}'\ g\ env[v \mapsto \ell] \\
\mathcal{B}'\ (g_1\ g_2)^{\ell}\ env & = & (\mathcal{B}'\ g_1\ env) \bigcup (\mathcal{B}'\ g_2\ env)
\end{array}
$$

Figure 2.13: Important Relations for Graphs

Therefore, the binder function $\vec{\mathcal{B}}$ is defined for admissible graphs as

$$
\begin{array}{rcl}
\vec{\mathcal{B}}: & & Graph \rightarrow Label \rightarrow Label \bigcup \{\mathbf{free}\} \\
\vec{\mathcal{B}}\ g\ \ell & = & \ell'\ iff\ (\ell, \ell') \in (\mathcal{B}\ g)
\end{array}
$$

I will be using $\mathcal{L}$, and $\mathcal{B}$ rather than $\vec{\mathcal{L}}$, and $\vec{\mathcal{B}}$ in contexts when it is clear that the graph is admissible, and thus these functions exist.

Figure 2.14 shows an admissible graph to the left, as well as an inadmissible graph to the right. In the latter, x^{ℓ_2} is bound both to ℓ_4, and ℓ_7. The requirement of a graph to be admissible is akin to the usual requirement that variables be bound in only one

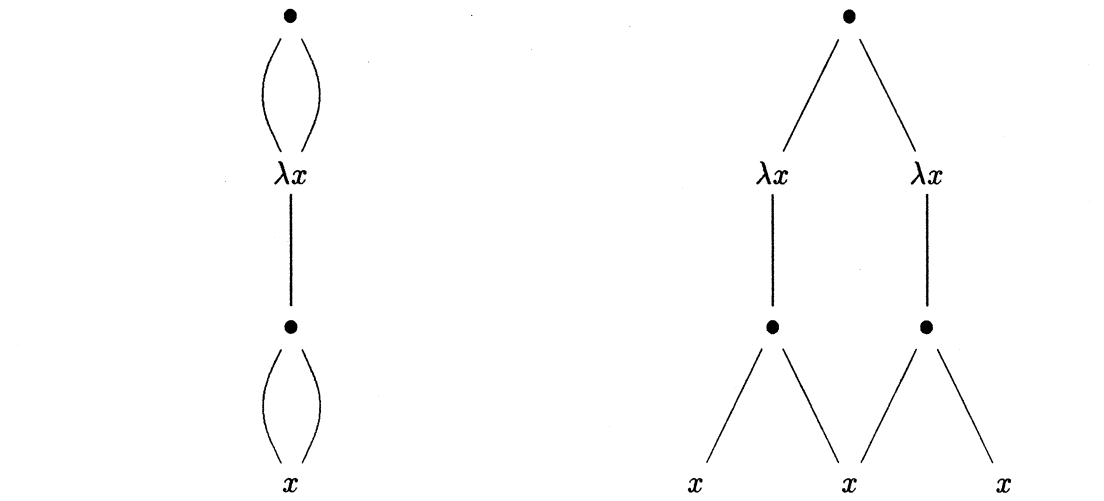


Figure 2.14: An Admissible, and an Inadmissible Graph

place. Substitution performed to inadmissible graphs may result in a new graph which does not "behave" as intended; e.g., the expansion of the substituted graph as a lambda expression may not match to the substitution of the lambda expression represented by the graph.

Similar to λ-calculus, the notation $C[\,]$ is used to indicate a context with holes. $C[g]$ means that the holes are filled with graph g. Note that g may be shared by several graphs; in graph reduction, all instances of g must be peers.

Labels were introduced solely for identifying peer expressions, and have no other significance. Therefore renaming of labels has no relevance in equality of graphs, provided the new label is not already used in the graph (no new sharing relations are introduced) and the old label is replaced across the entire graph; i.e.. that all the instances of the old label were replaced by the new one (no sharing relations are eliminated). When one graph (g_1) is obtained from another (g_2) by doing such a label substitution, the former is obtained from the latter by γ-reduction ($g_2 \rightarrow^{\gamma} g_1$);

$$\_[\_/\_] : \; Graph \rightarrow Graph \rightarrow Var \rightarrow Graph$$

$$\begin{aligned}
k^{\ell}[g'^{\ell'}/v] &= k^{\ell} \\
v^{\ell}[g'^{\ell'}/v] &= g'^{\ell'} \\
v_1^{\ell}[g'^{\ell'}/v_2] &= v_1^{\ell} \\
(\lambda v.g)^{\ell}[g'^{\ell'}/v] &= (\lambda v.g)^{\ell} \\
(\lambda v_1.g)^{\ell}[g'^{\ell'}/v_2] &= (\lambda v.g[g'^{\ell'}/v_2])^{\ell} \\
(g_1\ g_2)^{\ell}[g'^{\ell'}/v] &= (g_1[g'^{\ell'}/v]\ g_2[g'^{\ell'}/v])^{\ell})
\end{aligned}$$

Figure 2.15: Substitution in Graph Reduction

γ-convertibility is the reflexive and transitive closure of γ-reduction. Formally,

$$C[g^{\ell}] \rightarrow^{\gamma} C[g^{\ell'}] \; \textit{iff} \; \ell' \notin (lab\ C[g^{\ell}])$$

where $(lab\ g)$ is the set of labels identifying the nodes of g, and is defined in Figure 2.13.

2.3.1 Substitution

Substitution on graphs is essentially the same as substitution on λ-expressions. It definition appears in Figure 2.15. All subgraphs sharing the same label are called *peers* and denote the *same* graph (the reduction rules act 'simultaneously' on them) [Wadsworth, 1971, pp 146]. In constrast, there can be isomorphic graphs with different labels, in which case they are reduced independently.

Note that sharing of graph nodes is made explicit in the substitution: all free occurrences of the target variable are replaced with peers of the same graph g'. In fact, no new labels are introduced by the substitution.

Unfortunately, allowing unrestricted graph substitution of well-formed graphs may render an ill-formed graph, since labels may be 'incompatible', i.e., a label may be

used in both operands to label two different subgraphs. In operational terms, this may be interpreted as having two different graph nodes in one memory location!

The following lemma characterizes a sufficient condition for obtaining well-formed graphs from substitution.

Lemma 2.3.1 *For all well-formed graphs g_1, and g_2. If $(\vec{\mathcal{L}}g_1) \sqcup (\vec{\mathcal{L}}g_2)$ is a partial function, then $g_1[g_2/v]$ is also a well-formed graph.*

Proof $(\vec{\mathcal{L}}g_1) \sqcup (\vec{\mathcal{L}}g_2)$ is a partially defined function if, and only if the labels common to both g_1 and g_2 are labelling the same subgraph in both graphs. □

2.3.2 Reduction Rules

Substitution to an admissible graph will behave as intended. However, reduction rules do not act on graphs as a whole, but on particular subgraphs (the redexes). In this case, admissibility is just enough to guarantee that the semantics of the graph substitution be compatible with that of the term substitution for the *redex*, but *not* for the whole graph. In order to ensure semantic compatibility for the whole graph when the substitution operation is performed on a subgraph, an extra condition needs to be imposed to the graph: The root of the subgraph where the substitution takes place needs to be *unshared*, i.e., pointed to by only one node. Otherwise, the expression resulting from the substituted graph may not be as intended.

In the specific case of the β-rule, in order to guarantee semantic compatibility of substitution, and admissibility of the reduced graph, it is necessary to ensure that the operator node of the redex is only accessible through this redex, i.e., that the redex does not share this node. Such an admissible graph is called an *R-admissible* graph. Figure 2.16 shows an example of an admissible graph and an R-admissible graph derived from it.

One way to enforce the R-admissibility of the redex is to copy enough of the function node of the redex in such a way that the resulting graph is still admissible, just when performing the β-rule. Some duplication of the function graph—a

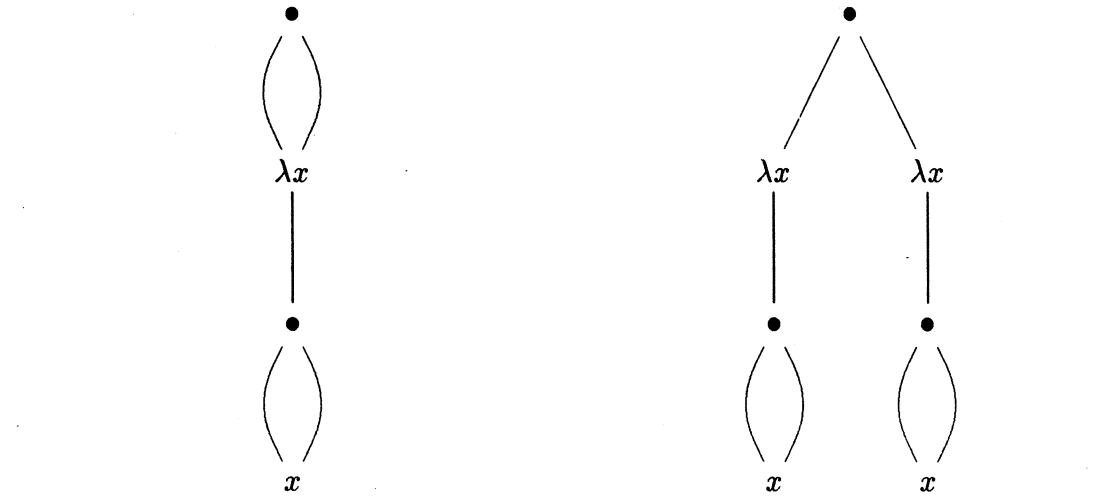


Figure 2.16: An Admissible, and an R-admissible Graph

λ-abstraction—will take place. The whole λ-abstraction could be copied, thus guaranteeing the R-admissibility, but that would be wasteful. A technique presented in [Wadsworth, 1971] is to copy only the nodes from the abstraction that cannot be abstracted away; viz., all the nodes that are ancestors of any occurrence of the bound variable of the abstraction, including—of course—the root node of the abstraction. That guarantees admissibility of the resulting graph, since all the occurrences of the bound variable are duplicated to ensure that the original occurrences of the variables are only bound to the original abstraction. Also, R-admissibility is guaranteed for the redex, since a fresh (unshared) copy of the λ-node has been created. It is actually unnecessary to copy the root λ-node in the copy operation, since it will be discarded when the β-reduction takes place. In the remainder of the dissertation, I will call this copy technique as *Wadsworth's copy operation.* Its notation is $\overline{g}^{\ell}$, where g is the graph being partially copied, and ℓ is the node label of the λ-abstraction binding the variable that causes the duplication. By abuse of notation, $\overline{g}^{x}$ may also be used, where x is the bound variable of the λ-abstraction. Figure 2.17 presents an implementation

$$
\begin{aligned}
\overline{g^{\ell}}^{\ell'} &= \textit{let } L = \textit{set of unused labels} \\
&\qquad \langle g', L' \rangle = \mathcal{C}\ g^{\ell}\ \ell'\ L\ (\mathcal{B}\ g^{\ell}) \\
&\qquad \textit{in } g'
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}\ g^{\ell}\ \ell'\ L\ b &= \langle g^{\ell}, L \rangle && \textit{if } \ell' \notin (b\ \ell) \\
\mathcal{C}\ g^{\ell}\ \ell'\ L\ b &= \mathcal{C}'\ g^{\ell}\ \ell'\ L\ b && \textit{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}'\ v\ \{\ell\} \bigcup L\ b &= \langle v^{\ell}, L \rangle \\
\mathcal{C}'\ (\lambda v.g^{\ell_1})\ \{\ell\} \bigcup L\ b &= \langle (\lambda v.g^{\ell_1})^{\ell}, L \rangle \\
\mathcal{C}'\ (g_1^{\ell_1}\ g_2^{\ell_2})\ \{\ell\} \bigcup L\ b &= \textit{let } \langle g_3^{\ell_3}, L_1 \rangle = \mathcal{C}\ g_1^{\ell_1}\ L\ b \\
&\qquad \langle g_4^{\ell_4}, L_2 \rangle = \mathcal{C}\ g_2^{\ell_2}\ L_1\ b \\
&\qquad \textit{in } \langle (g_3^{\ell_3}\ g_4^{\ell_4})^{\ell}, L_2 \rangle
\end{aligned}
$$

Figure 2.17: Wadsworth's Copy Operation

of the operation.

The notation $C[g_1] \rightarrow C[g_2]$ is used to indicate that *all* peers of graph g_1 in graph C are *simultaneously* transformed to graph g_2. Note that $C[g_2^{\ell}]$ will be well-formed only if *all* occurrences of g_1 have been transformed, else ℓ will be associated with two different subgraphs rendering the graph ill-formed. The rules are designed in such a way that it is guaranteed that if one peer graph is a redex the all its peers are redexes, and thus the simultaneous reduction of all the peers can in fact be performed. Note that this parallels graph reduction where all peers of a graph are represented by a single graph, and reducing that graph corresponds to the simultaneous reduction of all the peers.

Computation is performed in graph reduction by applying a set of reduction rules

$$
\begin{array}{lrcl}
\beta & C[\![((\lambda v.g_1)^{\ell_1}\ g_2^{\ell_2})]\!] & \rightarrow & C[\![\overline{g_1}^{v}[g_2^{\ell_2}/v]]\!] \\
\delta_{if} & C[\![(if\ true\ g_1^{\ell} g_2)]\!] & \rightarrow & C[\![g_1^{\ell}]\!] \\
 & C[\![(if\ false\ g_1\ g_2^{\ell})]\!] & \rightarrow & C[\![g_2^{\ell}]\!] \\
\delta_{mka} & C[\![(mka\ i\ g^{\ell})]\!] & \rightarrow & C[\![(Array\ {}^1g^{\ell}\ \ldots\ {}^ig^{\ell})]\!] \\
\delta_{update} & C[\![(update\ (Array\ {}^1g_1^{\ell_1}\ldots{}^ig_i\ldots{}^ng_n^{\ell_n})\ i\ g^{\ell})]\!] & \rightarrow & C[\![(Array\ {}^1g_1^{\ell_1}\ldots{}^ig^{\ell}\ldots{}^ng_n^{\ell_n})]\!] \\
\delta_{fix} & C[\![(fix\ (\lambda v.g)^{\ell_1})^{\ell}]\!] & \rightarrow & C[\![\overline{g}^{v}[(fix\ (\lambda v.g))^{\ell}/v]]\!] \\
\delta_{copy} & C[\![(copy\ (Array\ g_1^{\ell_1}\ldots g_n^{\ell_n}))]\!] & \rightarrow & C[\![(Array\ g_1^{\ell_1}\ldots g_n^{\ell_n})]\!]
\end{array}
$$

Figure 2.18: Reduction Rules for Graph Reduction

to the original graph, or "program". These rules are shown in Figure 2.18, and qualify as notions of reduction.

The notion of substitutivity in graph reduction is as follows (note the complication introduced to ignore the labels):

$$If\ g_1 \rightarrow g_2,\ then\ \forall C \exists g_2' C[g_1] \rightarrow C[g_2']\ \&g_2 \equiv^{\gamma} g_2'$$

One-step reduction in graph reduction does not satisfy the diamond property. In fact, it is not even confluent in the traditional sense; Figure 2.19 shows a counterexample. In Subsection 2.3.4, I will introduce a notion of confluence that graph reduction satisfies.

2.3.3 Relation between λ-calculus and Graph Reduction

In this section, I explore the relation between expressions in λ-calculus and graphs in graph reduction. On one side, graph reduction has a richer syntax, since it allows graphs to be labelled whereas λ does not allow any kind of labelling. Labelling, as mentioned before, is the mechanism through which sharing properties are specified. I

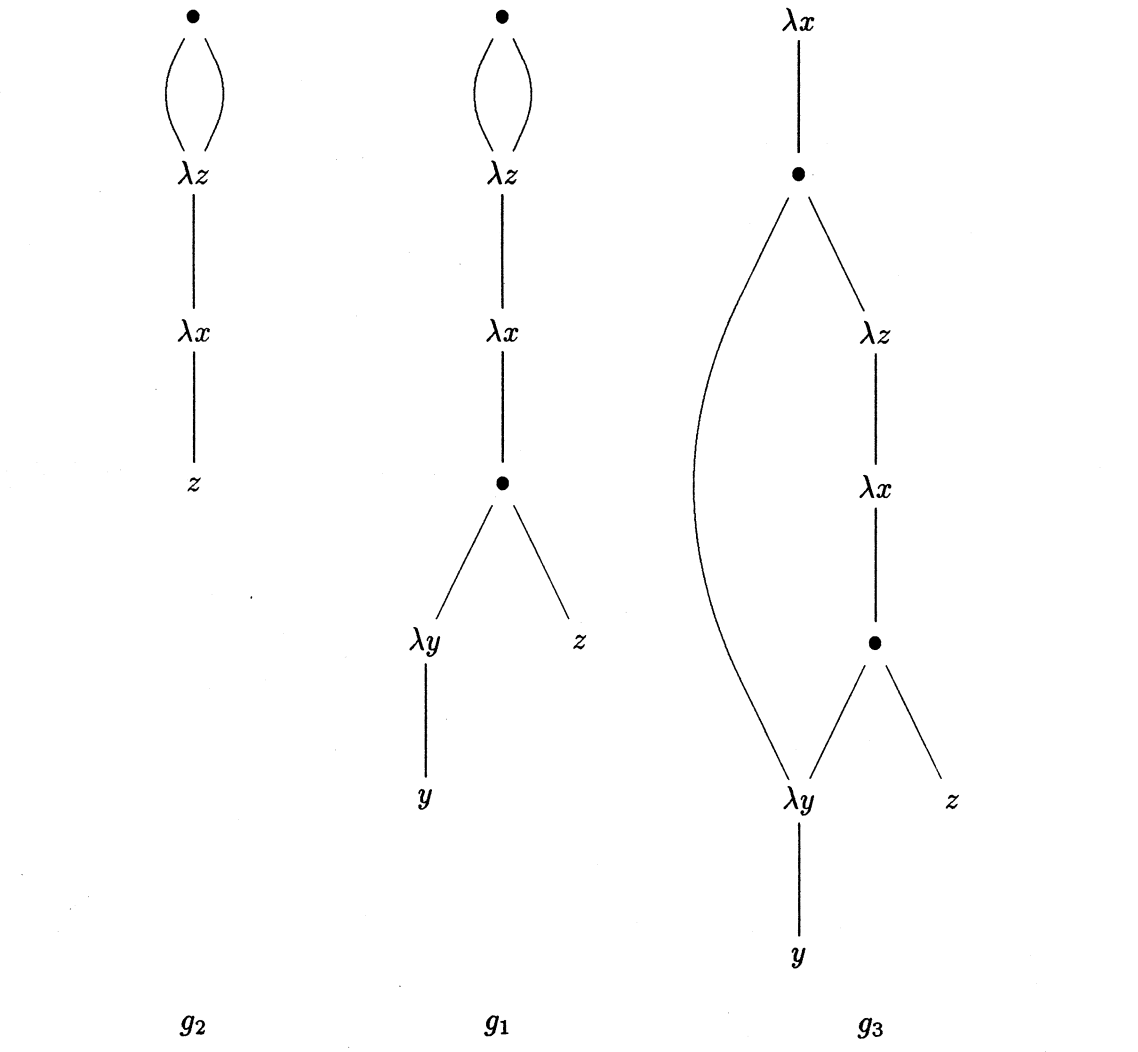


Figure 2.19: Reductions that do not Satisfy the Diamond Property

will define an unravelling function U that strips all labelling information from graphs, removes all sharing information that could be akin to van Eekelen's unravel [Eekelen, 1988].

$$U : Graph \rightarrow Exp$$

In the opposite direction, there are several sensible functions that map λ-expressions to graphs. Among them is the function that folds all identical subexpressions into the same DAG. Here I will only consider the mapping that transforms each expression into a tree (a graph with no shared nodes). This seems natural, given that λ-expressions do not have any sharing information.

$$T : Exp \rightarrow Graph$$

U and T behave respectively like abstraction and concretization functions in abstract interpretation jargon; U removes information, and T provides just the most conservative assumptions about any sharing information that it is not able to reconstruct. As expected, $U \cdot T = id_{Exp}$, i.e., λ-expressions do not gain, or loose any information by being concretized to a graph, and later abstracted back to Exp. On the other hand, $T \cdot U \neq id_{Graph}$, which reflects the fact that information is being lost by the abstraction. However, $T \cdot U$ is idempotent (modulo γ-conversion), which means that all sharing information is lost at once. The actual definitions of U, and T appear in Figure 2.20.

In his dissertation, Wadsworth introduced Graph Reduction as an operational semantics for the λ-Calculus. There he provided projections between the language of λ-expressions and that of admissible graphs. He also showed that reduction steps within graph reduction corresponds to reduction paths in λ-calculus. However, due to the fact that graph reduction incorporates the notion of sharing, there are graphs that are not β_{GR}-convertible, but when their sharing features are ignored via the trivial abstraction to λ-calculus, the resulting expressions are β_{λ}-convertible. Figure 2.21 exhibits an example of such an anomaly. This fact was not relevant in Wadsworth work, since he was interested in abstracting the sharing features away.

$$
\begin{array}{rl}
U: & \mathit{Graph} \rightarrow \mathit{Exp} \\
T: & \mathit{Exp} \rightarrow \mathit{Graph} \\
T': & \mathit{Exp} \rightarrow \mathit{Label}^* \rightarrow \mathit{Graph}
\end{array}
$$

$$
\begin{array}{rcl}
U\ k^\ell & = & k \\
U\ v^\ell & = & v \\
U\ (\lambda v.g)^\ell & = & (\lambda v.(U\ g)) \\
U\ (e_1\ e_2)\ell & = & ((U\ e_1)\ (U\ e_2))
\end{array}
$$

$$
\begin{array}{rcl}
T\ e & = & \mathit{let}\ \langle g, Ls\rangle = T'\ e\ Ls \\
 & & \mathit{in}\ g
\end{array}
$$

$$
\begin{array}{rcl}
T'\ k\ \ell : Ls & = & \langle k^\ell, Ls\rangle \\
T'\ v\ \ell : Ls & = & \langle v^\ell, Ls\rangle \\
T'\ (\lambda v.e)\ \ell : Ls & = & \mathit{let}\ \langle g, Ls'\rangle = T'\ e\ Ls \\
 & & \mathit{in}\ \langle(\lambda v.g), Ls'\rangle \\
T'\ (e_1\ e_2)\ \ell : Ls & = & \mathit{let}\ \langle g_1, Ls_1\rangle = T'\ e_1\ Ls \\
 & & \quad \langle g_2, Ls_2\rangle = T'\ e_2\ Ls_1 \\
 & & \mathit{in}\ \langle(g_1\ g_2)^\ell, Ls_2\rangle
\end{array}
$$

Figure 2.20: Functions U and T

2.3.4 Confluence in Graph Reduction

As it was shown in Figure 2.19, graph reduction is not confluent—in the spirit of the Church-Rosser Theorem presented in Section 2.2.3—because the sharing properties of graphs are not preserved across different reduction paths. However, different normal forms for the same graph differ only in their sharing properties: their unravellings are isomorphic.

Confluence is a desirable property to have in any calculus, since it guarantees determinism. In general, confluence is dependent on a subsidiary property—uniqueness of normal form in λ-calculus. However, the property of interest may not be that

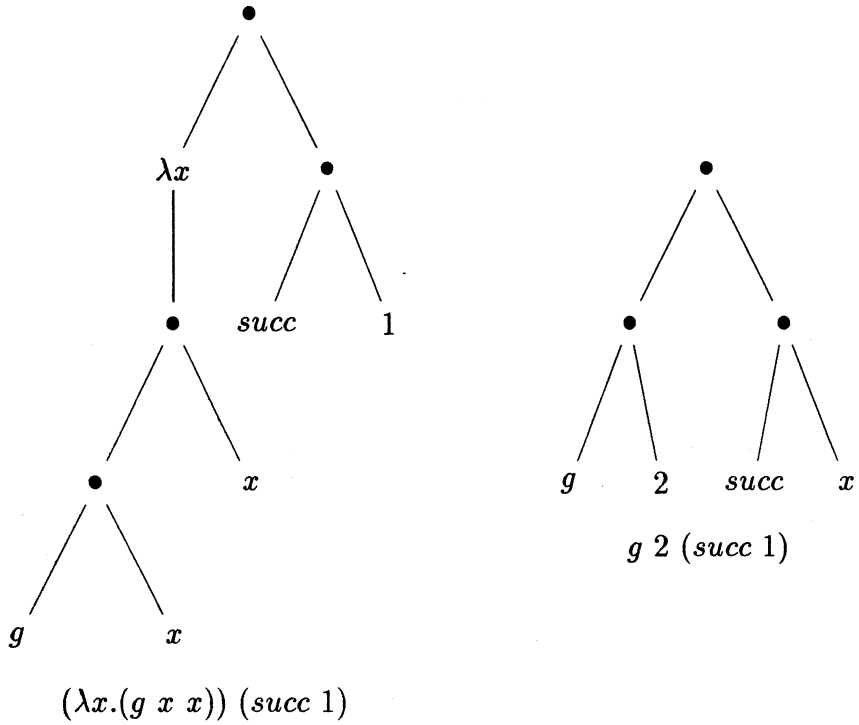


Figure 2.21: Non-β-Convertible Graphs whose λ-Expressions Are β-Convertible

one, but any property of the normal form, or, in general, *any property of the reduction sequence.* This shows that the notion of confluence is as relative as referential transparency (see Appendix 1.6), and should be interpreted as the *satisfiability of a uniqueness property*, rather than being fixed to its notion in λ-calculus. In graph reduction, the uniqueness of the *unravelling* of the normal form serves as the notion of confluence. This property reflects the intention of computing via graph reduction, but reasoning about the value of the associated expressions within the framework of the λ-calculus.

Lemma 2.3.2 *Let g be a graph. If $g \rightarrow_{\beta^*_{GR}} g'$ then $(U\ g) \rightarrow_{\beta^*_\lambda} (U\ g')$.*

Proof By induction on the number n of β^*_{GR}-reductions needed to reduce g to g'.

$$g = g_0 \rightarrow^*_{\beta_{GR}} g_1 \rightarrow^*_{\beta_{GR}} \cdots \rightarrow^*_{\beta_{GR}} g_n = g'$$

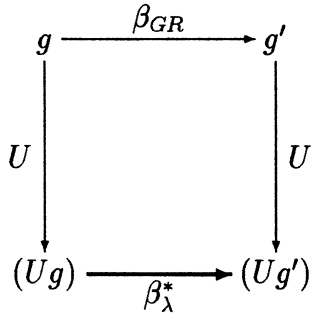


Figure 2.22: Relation Between β_λ, and β_{GR} (I)

For each β_{GR}-reduction, taking g_i into g_{i+1} by reducing redex Δ corresponds to multiple simultaneous reductions of $(U\ g_i)$ to $(U\ g_{i+1})$, each of them reducing a peer of the redex Δ. Figure 2.22 provides a visual representation of the relation. □

Lemma 2.3.3 *Let g be a graph. Then g is in normal form iff $(U\ g)$ is in normal form.*

Proof Suppose g is not in normal form; i.e., it has a redex g'. Then $(U\ g)$ has all peers of $(U\ g')$ as redexes.

Conversely, suppose $(U\ g)$ is not in normal form, so it has redex e. Then e must correspond to some g' in g such that $U\ g' = e$. Therefore g' is a redex in g. □

Theorem 2.3.4 Church-Rosser Theorem for Graph Reduction.

1. *For all graphs g, g_1, and g_2, if $g \rightarrow^*_{\beta_{GR}} g_1$, and $g \rightarrow^*_{\beta_{GR}} g_2$, then it is true that $(U\ g) \rightarrow^*_{\beta_\lambda} (U\ g')$, $(U\ g) \rightarrow^*_{\beta_\lambda} (U\ g'')$, and there exists e such that $(U\ g') \rightarrow^*_{\beta_\lambda} e$, and $(U\ g'') \rightarrow^*_{\beta_\lambda} e$.*

2. *If $g_1 \leftrightarrow^*_{\beta_{GR}} g_2$ then there exists g_3 such that $g_1 \rightarrow^*_{\beta_{GR}} g_3$, and $g_2 \rightarrow^*_{\beta_{GR}} g_3$.*

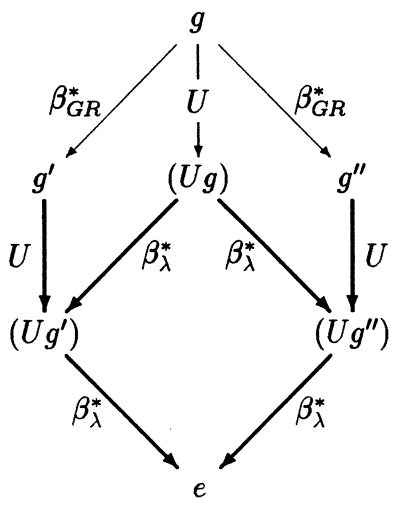


Figure 2.23: Church-Rosser Theorem for Graph Reduction

Proof The theorem states that the diagram of Figure 2.23 is true. The two upper boxes are consequence by Lemma 2.3.2. The lower diamond is the Church-Rosser theorem for λ-calculus (Theorem 2.2.1). □

Extended confluence is guaranteed as a corollary to the previous theorem:

Corollary 2.3.5 *1. For all g_1 and g_2, if g_2 is a normal form of g_1, then there exists g_3 such that $g_1 \rightarrow^*_{\beta_{GR}} g_3$, and $(U\ g_3) = (U\ g_2)$.*

2. For all g, if $g_1,\ldots,g_n$ are all normal forms of g, then, for all $1 \leq i,j \leq n$, $(U\ g_i) = (U\ g_j)$.

Proof By simplification of the diagram in Figure 2.23, when g', and g'' are assumed to be in normal form. The simplification is shown in Figure 2.24. □

Theorem 2.3.4, and Corollary 2.3.5 are of particular importance, not only because they guarantee confluence in the calculus, but also because they relate graph normal forms to λ-normal forms. In fact, the following theorem states that a λ-expression can

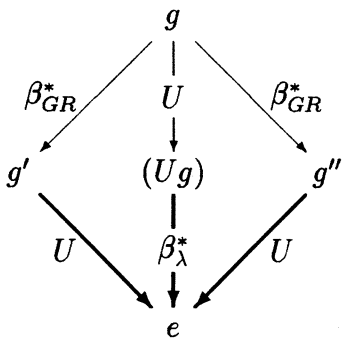


Figure 2.24: Confluence for Graph Reduction

be transformed to a graph, and be reduced via normal order reduction to a normal form in graph reduction, and transformed back to a λ-expression. The resulting expression—in normal form—is guaranteed to be the same one computed by β^*.

Theorem 2.3.6 *For all λ-expressions e, if e is reducible to a λ-normal form e', then $(T\ e)$ is reducible to a graph normal form g'.*

Proof This is a restatement of Wadsworth's Theorem 4.4.4 [Wadsworth, 1971, pp 176]. □

2.4 Summary

Lambda calculus and graph reduction were introduced in this chapter. Confluence properties were reviewed for both paradigms. Transformations between both models were presented. These are of particular importance since lambda calculus is the semantics of choice for functional languages, while graph reduction is just an efficient operational semantics for lambda calculus. With these transformations, users take advantage of faster computation by graph reduction while reasoning about their programs within the lambda calculus.

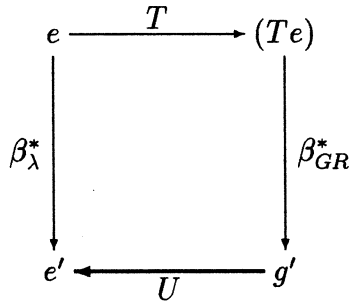


Figure 2.25: Relation Between β_λ, and β_{GR} (II)

Chapter 3

Single-Threaded Lambda Calculus

3.1 Introduction

The basic elements of lambda calculus and graph reduction were presented in Chapter 2. My interest in graph reduction is to use it as the basis for Single-Threaded Lambda Calculus—an extension of graph reduction were space reutilization can be expressed.

Object sharing becomes an issue when manipulating aggregate data structures, like arrays. These can be manipulated monolithically—the data structure is built up, and all values within it are specified at the same time—or incrementally—small changes to the structure are performed in order to *update* the contents of a certain field. In case they are manipulated incrementally, it is usually the case that the previous data structure is not manipulated any further. Several works have been done in order to detect when this is the case and extend the implementation to *reuse* the old data structure rather than creating a new one with the change in it. However, this *optimization* is performed "behind the scenes", and there is no feedback to the user to report where the optimization was done and where it was not done. In any case, the user intention of reusing a data structure cannot be expressed, therefore the user has to rely on the implementation to make sure that his/her intentions were

detected by the compiler.

I have followed another line of reasoning, in which the user is allowed to be explicit about his intentions of reusing an object. It is thus guaranteed that any implementation of the program will reuse the object. For this purpose, I introduce an alternative model of computation, single-threaded lambda calculus (λ_{st}-calculus), an extension to graph reduction that allows the user to reason about sharing, and how to manipulate and reuse shared objects.

The advantage of λ_{st}-calculus over λ-calculus and graph reduction is that the user has a simple way to *reason* about efficiency. Explicit sharing introduced by the user is not dependent on a particular implementation, but it is guaranteed to happen on all conformant implementations. In order to accomplish this goal, it is necessary to understand the notions of

1. sharing of structures,
2. structure copying,
3. updating a structure (or store reuse), and
4. sequencing expressions.

Section 3.2 extends graph reduction to include notions 2, 3, and 4. Section 3.3 summarizes Single-Threaded Lambda Calculus. Section 3.4 characterizes operational properties of the extended calculus. Section 3.5 introduces *abstract uses*, and abstraction of the operational properties. Finally, Section 3.6 presents *abstract liabilities*, a general technique designed for computation of abstract properties within a type system.

3.2 Extending Graph Reduction

Graph reduction already has the notion of structure sharing. I will introduce three other elements, one by one, and then form a new calculus—Single-Threaded Lambda

Calculus—which

- structure copying,
- mutation of state, and
- sequencing expressions.

3.2.1 Introducing Copying of Structures

The first extension to graph reduction is to provide a primitive to explicitly copy a structure. This takes the form of a constant function with the following δ-rule:

$$\delta_{copy} \quad C[(copy\ (Array\ g_1^{\ell_1} \ldots g_n^{\ell_n}))] \ \rightarrow \ C[(Array\ g_1^{\ell_1} \ldots g_n^{\ell_n})]$$

Note that the rule copies the array while preserving the sharing properties of its elements.

3.2.2 Introducing Mutation of State

Intuitively, a *mutation* is a change of value of any object. In the context of graph reduction, a *mutation* is any change on the contents of any pre-existing graph node in the graph to which the rule is applied.

Implementations based on graph reduction use mutation to implement graph substitution. Specifically, the substitution $\overline{g_1}^v[g_2/v]$ in the β-reduction rule

$$\beta \quad C[((\lambda v.g_1)^{\ell_1}\ g_2)^{\ell_2}] \ \rightarrow \ C[\overline{g_1}^v[g_2/v]]$$

is implemented as a mutation of $\overline{g_1}^v$. However, given that $\overline{g_1}$ is a copy of g_1, it can be proven that any node that is changed in $\overline{g_1}$ is not in g_1; i.e., it is newly created by the copy. Thus, the mutation does not occur in any node of the original graph g.

If the β_{GR}-rule were not so precise as to establish that the function needed to be copied, but rather, that the graph $C[((\lambda v.g_1)^{\ell_1}\ g_2)^{\ell_2}]$ be R-admissible in the reducing

redex (thus implying that when the graph is R-admissible, no copy would be necessary), then there would be cases for already R-admissible graphs where the original graph was mutated. However, the mutated g_1 would only be visible through the hole $C[\ ]$ in this case, and thus, the resulting graph would have the same unravelling regardless of its mutation.

In other words, if $\zeta{:}g \rightarrow g'$ is a rule in the calculus, a mutation is any change of g implied by the application of the ζ-rule. The change consists of a label that appear in both g and g' labelling different objects. The rule is called a *mutator rule*, and the redex g is called a mutator. As an example of a mutation, the following δ-rule for *update*! specifies that the label associated with the original array be the same as the label associated with the resulting array.

$$\delta_{update!} \quad C[(update!\ (Array\ {}^1g_1 \ldots {}^ig_i \ldots {}^ng_n)^\ell\ i\ g')] \rightarrow C[(Array\ {}^1g_1 \ldots {}^ig' \ldots {}^ng_n)^\ell]$$

The fact that field i of location ℓ is mutated is represented by using the *same label* on both sides of the arrow for the modified object in location ℓ. In graph reduction terms, the graph node is rewritten, or mutated to reflect the change. Unfortunately, the introduction of such rules renders graph-reduction non-confluent, as the example of Figure 3.1 shows, where different values are obtained for different reduction sequences.

3.2.3 Introducing Sequential Constraints

The Church-Rosser property is one of the most important features of the λ-calculus. With the introduction of mutator rules—as the example of the previous section points out—graph-reduction loses this property: i.e., there are graphs that reduce to different normal forms on different reduction paths. For a deterministic language, this is very distressing, but is a consequence of the unrestricted coexistence of sharing and mutation. In fact, graph reduction is confluent, and so is a calculus where no sharing

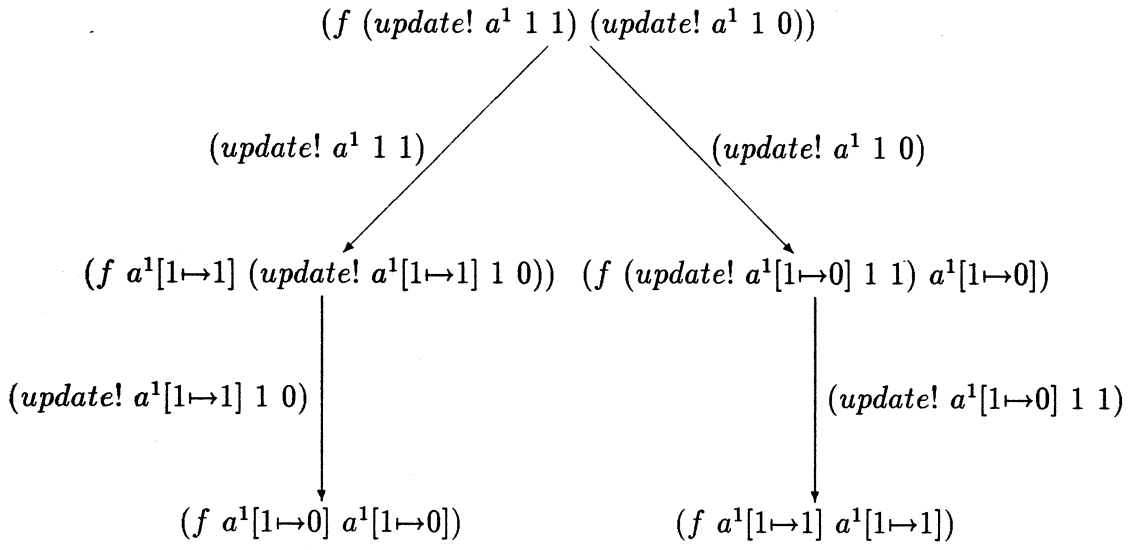


Figure 3.1: Nonconfluent Expressions Involving Mutators

is available, as the λ-calculus. Note that the substitution operation on graphs—a mutation operation—is only accessible "packaged" within β-reduction, along with a copy operation that ensures confluence. This is an example of a controlled coexistence of sharing and mutation.

In an otherwise non-confluent calculus, a way of guaranteeing confluence is to restrict redexes only to those that satisfy a property which, in turn, would imply confluence.

This can be done either by specifying a reduction strategy, and verifying that confluence is satisfied by all reduction paths that are permissible under the reduction strategy, or by modifying the calculus in such a way of restricting the choice of redexes. In any case, a well-formedness property may be imposed on expressions in such a way that the confluence of the calculus can be proven when restricted to well-formed expressions.

The choice of the solution largely depends on the nature of the problem being solved. In that respect, reduction strategies can be devised to select redexes from

within a particular group. Chosen redexes usually satisfy a global property on the expression, such as being outermost redex, or being leftmost redex, etc. These usually work well when the property being verified is a property on the current expression. On the other hand, modifying the calculus to devise notions of reduction that would work as intended have the advantage that the restrictions are encoded within the would-be redex, and thus is a local property, and the restrictions can be altered by the modified reduction rules. In either case, confluence becomes an issue to be solved if the starting calculusis non-confluent.

In this dissertation, sequential constraints are added to graph reduction in two fashions:

- by introducing a new type of application, evaluated in applicative order, that would indeed guarantee—for first order values—that the argument to the application is in normal order before the reduction of the applicative redex takes place; and
- a control mechanism whereby it is guaranteed that only certain redexes can be considered for reduction.

Strict Application

A new application is introduced to graph reduction with the following syntax:

$$\begin{aligned} g \ ::= \ & \ldots \\ & \mid \langle g_1 \ g_2 \rangle^{\ell} \quad \textit{strict applications} \end{aligned}$$

with extension to the β reduction rule to reduce only whenever the argument is in *weak head normal form.* This is the standard implementation of *call by value*:

$$\beta \quad C[\langle (\lambda v.g_1)^{\ell_1} \ g_2^{\ell_2} \rangle] \ \rightarrow \ C[\overline{g_1}^{v}[g_2^{\ell_2}/v]] \quad \textit{if } g_2 \textit{ is in whnf}$$

Weak head normal form (whnf) is a term used to indicate that the form of g_2 is *not* and application: it can be either a variable, a constant, an abstraction, or a data structure whose elements have to satisfy these same constraints. For expressions that evaluate to basic domains (integers, etc.), weak head normal form and normal form are the same. An array is in weak head normal form only if all its elements are.

Boxing

The solution proposed in order to regain confluence in the λ_{st}-calculus with mutators involves

- an extension to the calculus to introduce notions of reductions that, in effect, impose sequentiality restrictions to the order on which redexes can be chosen for reduction, and
- a well-formedness property of graphs which takes the form of a type property.

A characterization of well-formed graphs, as those that satisfy the *single-threadedness restriction* will be introduced later in the chapter.

As it is, λ-calculus already has a notion of sequentiality, based exclusively on data dependency. Data dependencies control where redexes appear, and it is possible to write expressions for which there is only one redex at any point in their reduction path, like in continuation passing style. However, sometimes sequentiallity is desired but there is no data dependency to enforce it, as in the case of conditionals. For example, in the λ-expression

$$(if\ e\ e_1\ e_2)$$

the expression e_1 is "needed" if, and only if, e_2 is not. Without going into the details of strictness analysis, and with the assumption that if e does indeed reduce to normal form $\hat{e}$ then $\hat{e}$ is either *true* or $false$[1], it might well be advantageous to fully reduce e before reducing any single redex in e_1 or e_2. Enforcing this fact in graph reduction

[1]This assumption is guaranteed by a type discipline: e must be of boolean type.

has the added advantage of allowing mutators to be in either e, e_1, and/or e_2, and restrict *when* they can be reduced. In fact, if no reductions in e_1 and e_2 take place until the if node is reduced, then mutations in e_1 will only be reduced if e_2 is not reduced, and vice versa.

Sequentiality is introduced by segregating graphs nodes that can be considered for reduction from those which cannot. This can be accomplished in two ways:

- using reduction strategies that enforce the sequentiality constraints, or
- by building these constraints within the reduction rules of the calculus.

The first option would lead to a calculus which is not necessarily confluent over all reduction strategies, but which would be confluent for those reduction strategies that satisfy some property The second option, on the other hand, can result in a "fully" confluent calculus and is the option we choose here. A key advantage of this approach is that deciding whether an expression is a redex is a *local* problem—the decision never depends on the context in which the expression appears.

The way we accomplish this is to mark syntactically with a *box* all would-be redexes that cannot be considered for reduction. These nodes are said to be *boxed*, and *unboxed* nodes are those which are not boxed. Boxed nodes may have unboxed nodes among its descendants. The reduction rules are modified so that new nodes can become susceptible for reduction as the graph is reduced (*unboxing*—this is typically done when sequentiality restrictions are lifted). Also, once a node becomes unboxed it never becomes boxed again. Thus, boxing is a mechanism through which sequential restrictions on how to reduce the expression are imposed. These restrictions are lifted when appropriate redexes are reduced, i.e., when the sequential restriction has been satisfied.

Technically, boxing is considered an extension to the language of expressions by providing two versions of each expression type, one boxed, and one unboxed. Below are the extensions to λ-calculus and graph reduction that support boxing.

Boxing in λ-calculus

In this subsection, the concept is introduced in the familiar framework of the λ-calculus. Although λ-calculus does not need boxing, it is introduced here because this calculus is presumed to be a familiar framework for the reader, and because comparisons between this calculus and graph reduction are useful.

As was explained above, boxing is introduced by extending the syntax of expressions to boxed nodes:

$$\begin{aligned} e ::= \; & \ldots \\ & \mid k_\Box \mid v_\Box \\ & \mid (\lambda v.e)_\Box \\ & \mid (e_1 \; e_2)_\Box \mid \langle e_1 \; e_2 \rangle_\Box \\ & \mid (if \; e_p \; e_c \; e_a)_\Box \end{aligned}$$

where $e_\Box$ means a boxed expression. The notation e_* is used to represent an expression e without specifying whether or not it is boxed.

Users are not supposed to directly write boxed expressions. Instead, there exists a boxing function $\mathcal{B} : \; Exp \rightarrow Exp$ which takes a standard λ-expression and returns a corresponding one with the following restrictions:

- The consequent and alternate subexpressions of if-expressions are boxed, thus imposing the restriction that the expression corresponding to the predicate must be reduced before any redexes in the branches.

- The body of lambda abstractions are boxed to ensure that only closed expressions are reduced.

- No other nodes are boxed.

Its definition appears in Figure 3.3. Figure 3.2 gives an example of a boxed expression.

The function $\mathcal{O} : \; Exp \rightarrow Exp$, on the other hand, is provided to opening selected boxes of an expression. It does not recursively unbox expressions within lambda

$$
\begin{aligned}
e &\equiv (\lambda x.\lambda y.y)\ ((\lambda x.(x\ x))\ (\lambda x.(x\ x))) \\
\mathcal{B}e &\equiv (\lambda x.(\lambda y.y_{\square})_{\square})\ ((\lambda x.(x_{\square}\ x_{\square})_{\square})\ (\lambda x.(x_{\square}\ x_{\square})_{\square}))
\end{aligned}
$$

Figure 3.2: A λ-expression and Its Boxed Counterpart

abstractions, nor does it unbox the consequent and alternate subgraphs of if-nodes. Its precise definition appears in Figure 3.3.

Finally, the function $\mathcal{U} : Exp \rightarrow Exp$ is the unboxing function: it removes any boxes in the expression. Its definition also appears in Figure 3.3.

A partial order $\preceq$ is defined on the structure of boxed expressions as follows: $e_1 \preceq e_2$ if

- $e_{1*} \equiv e_{2*}$, and
- for each boxed subexpression of e_1, the corresponding subexpression of e_2 is also boxed.

i.e., $e_1 \preceq e_2$ if e_1 is "less constrained" by boxing than e_2.

The reduction rules are changed so that the new β-rule unboxes the resulting expression, and the new δ_{if}-rule unboxes the consequent or alternate expression depending on the predicate. Formally, the new rules are as follows:

$$
\begin{array}{lrcl}
\beta & ((\lambda x.g_1)\ g_2) & \rightarrow & (\mathcal{O}\ \bar{g_1}[g_2/x]) \\
\delta_{if} & (if\ true\ g_1\ g_2) & \rightarrow & (\mathcal{O}\ g_1) \\
 & (if\ false\ g_1\ g_2) & \rightarrow & (\mathcal{O}\ g_2)
\end{array}
$$

Lemma 3.2.1 *For all expressions e such that $(\mathcal{B}\ e) \rightarrow^{\Delta} e''$ then, $e \rightarrow^{\Delta} e'$, and $e'' \preceq (\mathcal{B}\ e')$.*

Figure 3.4 shows graphically the contents of this lemma.

$$
\begin{aligned}
\mathcal{B}\ k &= k \\
\mathcal{B}\ k_{\square} &= k_{\square} \\
\mathcal{B}\ v &= v \\
\mathcal{B}\ v_{\square} &= v_{\square} \\
\mathcal{B}\ (\lambda v.g) &= (\lambda v.(\mathcal{B}'\ g)) \\
\mathcal{B}\ (\lambda v.g)_{\square} &= (\lambda v.(\mathcal{B}'\ g))_{\square} \\
\mathcal{B}\ (g_1\ g_2) &= ((\mathcal{B}\ g_1)\ (\mathcal{B}\ g_2)) \\
\mathcal{B}\ (g_1\ g_2)_{\square} &= ((\mathcal{B}\ g_1)\ (\mathcal{B}\ g_2))_{\square} \\
\mathcal{B}\ \langle g_1\ g_2\rangle &= \langle(\mathcal{B}\ g_1)\ (\mathcal{B}\ g_2)\rangle \\
\mathcal{B}\ \langle g_1\ g_2\rangle_{\square} &= \langle(\mathcal{B}\ g_1)\ (\mathcal{B}\ g_2)\rangle_{\square} \\
\mathcal{B}\ (if\ g_p\ g_c\ g_a) &= (if\ (\mathcal{B}\ g_p)\ (\mathcal{B}'\ g_c)\ (\mathcal{B}'\ g_a)) \\
\mathcal{B}\ (if\ g_p\ g_c\ g_a)_{\square} &= (if\ (\mathcal{B}\ g_p)\ (\mathcal{B}'\ g_c)\ (\mathcal{B}'\ g_a))_{\square}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{B}'\ k_* &= k_{\square} \\
\mathcal{B}'\ v_* &= v_{\square} \\
\mathcal{B}'\ (\lambda v.g)_* &= (\lambda v.(\mathcal{B}'\ g))_{\square} \\
\mathcal{B}'\ (g_1\ g_2)_* &= ((\mathcal{B}'\ g_1)\ (\mathcal{B}'\ g_2))_{\square} \\
\mathcal{B}'\ \langle g_1\ g_2\rangle_* &= \langle(\mathcal{B}'\ g_1)\ (\mathcal{B}'\ g_2)\rangle_{\square} \\
\mathcal{B}'\ (if\ g_p\ g_c\ g_a)_* &= (if\ (\mathcal{B}'\ g_p)\ (\mathcal{B}'\ g_c)\ (\mathcal{B}'\ g_a))_{\square}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{O}\ k_* &= k \\
\mathcal{O}\ v_* &= v \\
\mathcal{O}\ (\lambda v.e)_* &= (\lambda v.e) \\
\mathcal{O}\ (e_1\ e_2)_* &= ((\mathcal{O}e_1)\ (\mathcal{O}e_2)) \\
\mathcal{O}\ \langle e_1\ e_2\rangle_* &= \langle(\mathcal{O}e_1)\ (\mathcal{O}e_2)\rangle \\
\mathcal{O}\ (if\ e_p\ e_c\ e_a)_* &= (if\ (\mathcal{O}e_p)\ e_c\ e_a)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{U}\ k_* &= k \\
\mathcal{U}\ v_* &= v \\
\mathcal{U}\ (\lambda v.e)_* &= (\lambda v.(\mathcal{U}e)) \\
\mathcal{U}\ (e_1\ e_2)_* &= ((\mathcal{U}e_1)\ (\mathcal{U}e_2)) \\
\mathcal{U}\ \langle e_1\ e_2\rangle_* &= \langle(\mathcal{U}e_1)\ (\mathcal{U}e_2)\rangle \\
\mathcal{U}\ (if\ e_p\ e_c\ e_a)_* &= (if\ (\mathcal{U}e_p)\ (\mathcal{U}e_c)\ (\mathcal{U}e_a))
\end{aligned}
$$

Figure 3.3: The "boxing" function $\mathcal{B}$ for the λ-calculus

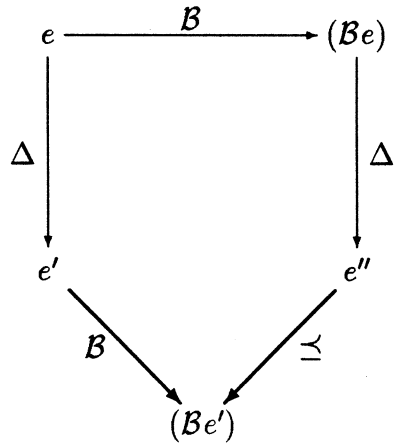


Figure 3.4: Weakening of Boxing Constraints by Reduction

Proof This proof is essentially the weak diamond property proof necessary to prove that β is Church-Rosser, except that fewer cases exist. Let us examine the relative positions of Δ_1, and Δ_2, the redexes of e that cause reductions to e_1, and e_2, respectively.

- $\Delta_1 = \Delta_2$, Then $e_1 = e_2 = e_3$.
- $\Delta_1 \cap \Delta_2 = \varnothing$ (the two redexes are disjoint). Then $e \rightarrow^{\Delta_1} e_1 \rightarrow^{\Delta_2} e_3$, and $e \rightarrow^{\Delta_2} e_2 \rightarrow^{\Delta_1} e_3$.
- $\Delta_1 \subset \Delta_2$, then it must be the case Δ_2 appears in the argument of Δ_1 (it would be boxed otherwise). e_3 can be constructed by either reducing Δ_1, and then reducing all occurrences of Δ_2 (note that (Δ_2 does not affect the boxing property of any other expressions in Δ_1). It can also be obtained by reducing Δ_2 first, and then reducing Δ_1. In both cases, the boxing property the Δ_2-redex remains the same.

□

Theorem 3.2.2 *The λ-calculus thus modified to support boxing is Church-Rosser.*

Proof This is a direct consequence of Lemma 3.2.1, which ensures the weak diamond property for $\beta_\Box$, and the fact that $\beta_\Box$ is a subset of β. □

Boxing in Graph Reduction

Boxing is introduced in graph reduction in a fashion similar to the way it was introduced in the λ-calculus.

$$
\begin{aligned}
g \ ::= \ & \ldots \\
& \mid k^{\ell}_{\Box} \mid v^{\ell}_{\Box} \\
& \mid (\lambda v.g)^{\ell}_{\Box} \\
& \mid (g_1 \ g_2)^{\ell}_{\Box} \mid \langle g_1 \ g_2 \rangle^{\ell}_{\Box} \\
& \mid (if \ g_p \ g_c \ g_a)^{\ell}_{\Box} \\
& \mid (fix \ g)^{\ell}_{\Box} \\
& \mid (copy \ g)^{\ell}_{\Box}
\end{aligned}
$$

with the corresponding modifications to the reduction rules:

$$
\begin{array}{lrcl}
\beta & C[\![(\lambda v.e_1)^{\ell_1} \ e_2)^{\ell_2}]\!] & \rightarrow & C[\![(\mathcal{O} \ \overline{e_1}^{v}[e_2/v])]\!] \\
\delta_{if} & C[\![(if \ true \ e_1 \ e_2)^{\ell}]\!] & \rightarrow & C[\![(\mathcal{O} \ e_1)]\!] \\
& C[\![(if \ false \ e_1 \ e_2)^{\ell}]\!] & \rightarrow & C[\![(\mathcal{O} \ e_2)]\!]
\end{array}
$$

Also, the functions $\mathcal{O}$, and $\mathcal{B}$ described for the λ-calculus are defined similarly for graph reduction, with the caveat that $\mathcal{B}$ is only well-defined for trees, as it is intended that the boxing of all peers of a node be consistent (either all are boxed, or none are).

Although graph reduction plus boxing is confluent, the introduction of mutators still renders a non-confluent calculus. The Example 3.1 shown previously is still a valid example of non-confluence in graph reduction thus modified. However, with boxing, it is much easier to impose restrictions that will result in a confluent reduction. Figure 3.5 shows an example of a confluent graph due to boxing.

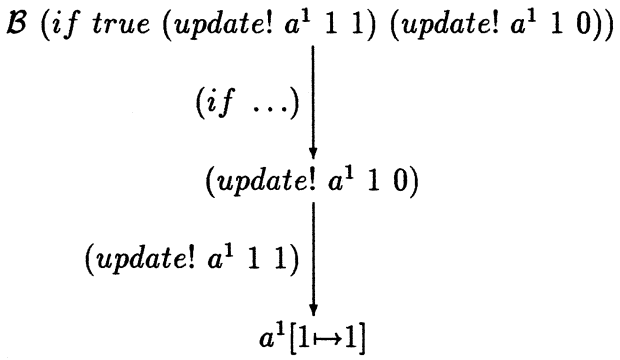


Figure 3.5: Example of a Confluent Graph using Boxing

3.3 Single-Threaded Lambda Calculus—λ_{st}

Single-Threaded Lambda-Calculus (λ_{st}-calculus) is a calculus that gathers the modifications suggested in Section 3.2. Note the presence of the strict application. Also, the conditional if, fixpoint operator fix, and copy function *copy* are presented as part of syntax. Although not strictly necessary, this will be useful in stressing the fact that the type system presented in Chapter 4 handles them in a special way. The *copy* operator creates an unshared copy of its argument while preserving sharing properties internal to its argument. Its syntax is introduced in Figure 3.6, and Figure 3.7 shows the corresponding reduction rules.

Concepts leading to R-admissible graphs extend directly to λ_{st}-calculus. Substitution is also 'borrowed' from graph reduction. Also, the Church-Rosser theorem for graph reduction holds in λ_{st}-calculus for those graphs that do not contain mutators.

An advantage of this computational model over the λ-calculus is that it allows the user to reason about sharing and object reuse *without* the introduction of a formal store. Thus, the user has a simple way to *reason* about efficiency. Explicit object reutilization introduced by the user is not dependent on a particular implementation, but it is guaranteed to happen on all conformant implementations.

$$
\begin{array}{rcll}
f, v & \in & Ide & \textit{identifiers} \\
k & \in & Kon & \textit{constants} \\
i & \in & Int \subset Kon & \textit{integers} \\
\ell & \in & Label = Int & \textit{labels} \\
 & & & \\
g & \in & Graph & \textit{graphs} \\
where\ g & ::= & k^{\ell} & \textit{constants} \\
 & & \mid v^{\ell} & \textit{identifiers} \\
 & & \mid (\lambda v.g)^{\ell} & \textit{abstractions} \\
 & & \mid (g_1\ g_2)^{\ell} & \textit{applications} \\
 & & \mid \langle g_1\ g_2 \rangle^{\ell} & \textit{strict applications} \\
 & & \mid (if\ g_p\ g_c\ g_a)^{\ell} & \textit{conditionals} \\
 & & \mid (fix\ g)^{\ell} & \textit{fixpoints} \\
 & & \mid (copy\ g)^{\ell} & \textit{copy primitive}
\end{array}
$$

Figure 3.6: Syntax of Single-Threaded Lambda Calculus

However, the introduction of object reutilization renders the calculus non-confluent. Even with the extra apparatus provided by adding sequentiality the resulting calculus remains non-confluent. A new notion of well-formedness is needed. One that can guarantee only to be satisfied by confluent graphs. A method for providing such a property is to actually perform all possible evaluations of the graph and verify that they all provide the same normal form. Unfortunately, finding a normal form is a *semi-decidable* problem (a problem which is not enumerable, but recursively enumerable), which means that there is no *effective* algorithm (an algorithm that always terminates) to compute it.

Other approaches must be taken. Mine is to enforce a type system on graphs such

$$
\begin{array}{lrcl}
\beta & C[\![((\lambda v.g_1)^{\ell_1}\ g_2^{\ell_2})]\!] & \rightarrow & C[\![\overline{g_1}^{v}[g_2^{\ell_2}/v]]\!] \\
 & C[\![\langle(\lambda v.g_1)^{\ell_1}\ g_2^{\ell_2}\rangle]\!] & \rightarrow & C[\![\overline{g_1}^{v}[g_2^{\ell_2}/v]]\!] \\
 & & & \textit{if } g_2 \textit{ is in whnf} \\
\delta_{if} & C[\![(if\ true\ g_1^{\ell} g_2)]\!] & \rightarrow & C[\![g_1^{\ell}]\!] \\
 & C[\![(if\ false\ g_1\ g_2^{\ell})]\!] & \rightarrow & C[\![g_2^{\ell}]\!] \\
\delta_{mka} & C[\![(mka\ i\ g^{\ell})]\!] & \rightarrow & C[\![(Array\ {}^1g^{\ell}\ \ldots\ {}^ig^{\ell})]\!] \\
\delta_{update} & C[\![(update\ (Array\ {}^1g_1^{\ell_1}\ldots{}^ig_i\ldots{}^ng_n^{\ell_n})\ i\ g^{\ell})]\!] & \rightarrow & C[\![(Array\ {}^1g_1^{\ell_1}\ldots{}^ig^{\ell}\ldots{}^ng_n^{\ell_n})]\!] \\
\delta_{update!} & C[\![(update!\ (Array\ {}^1g_1\ldots{}^ig_i\ldots{}^ng_n)^{\ell}\ i\ g')]\!] & \rightarrow & C[\![(Array\ {}^1g_1\ldots{}^ig'\ldots{}^ng_n)^{\ell}]\!] \\
\delta_{fix} & C[\![(fix\ (\lambda v.g)^{\ell_1})^{\ell}]\!] & \rightarrow & C[\![\overline{g}^{v}[(fix\ (\lambda v.g))^{\ell}/v]]\!] \\
\delta_{copy} & C[\![(copy\ (Array\ g_1^{\ell_1}\ldots g_n^{\ell_n}))]\!] & \rightarrow & C[\![(Array\ g_1^{\ell_1}\ldots g_n^{\ell_n})]\!]
\end{array}
$$

Figure 3.7: Reduction Rules for Single-Threaded Lambda Calculus

that graphs that satisfy the type discipline are guaranteed confluent. The discipline must analyze an abstraction of some operational properties of graphs, such as sequentiality constraints, as well as which graphs may be subject to reutilization. The availability of such a property provides us with a confluent calculus for well-formed expressions. This type system will be introduced in Chapter 4. The remainder of this chapter provides the foundation for that tool.

3.4 Operational Properties of λ_{st}-Calculus

As was shown in Example 3.1, λ_{st}-calculus is not confluent. In order to guarantee confluence, I have chosen to impose a type discipline on the language that discriminates confluent graphs from possibly non-confluent ones. This corresponds to a restriction on the language to well-formed λ_{st}-graphs. In this section, I present operational

properties on λ_{st}-graph that precisely characterize confluent graphs. The elaboration of a type system that calculates abstractions of these properties is the subject of Chapter 4.

I am interested in capturing a combination of three key properties of how objects are used when an expression is evaluated to normal form:

- *Mutability*—an object may be *read-only*, or *written.*
- *Sharing*—an object may be *captured* by another object (i.e. shared as substructure), or it may be *free.*
- *Linearity*—an object may be *single-threaded* (i.e. used at most once), or *multiple-threaded.*

These properties carry a fair degree of intuition, and I feel are the minimum necessary to reason about a suitably rich notion of state. Of course, these properties cannot be inferred precisely; I only ask that the static type system capture and enforce a useful approximation to them (in the same way that the Hindley-Milner type system, for example, approximates a more general notion of types). This dissertation only deals with these three properties, but the methodology is valid for any number of them. The different abstractions to the operational properties of interest (*abstract uses*) are presented in Section 3.5, and the technique through which these properties are associated with objects in the program (*liabilities*) is introduced in Section 3.6.

3.4.1 Definitions

Definition 3.4.1 *Given a graph g with sub-graph g' (e.g., $g = C[g']$), and a redex Δ. it is said that g' is* mutated *by Δ if*

- *Δ is a* mutator,
- *g' is the the graph node that is being overwritten by the reduction of Δ.*

In the case of mutable arrays, Δ is a redex of the form (update! a i x), and g' is the array a.

This definition is extended to reduction paths as follows

Definition 3.4.2 *g is mutated by $\Delta_1 \ldots \Delta_n$ if a residual of g is the target of Δ_i, for some $i \in \{1, 2, \ldots, n\}$ in the reduction path.*

Definition 3.4.3 *A graph g' is used* read-only *in g if it is not subject to mutation.*

Definition 3.4.4 *Given a graph g with sub-graph g' (e.g., $g = C[g']$), g' is* captured *by g if there exists a graph g'' such that $(g''\ g)$ can be reduced to g''', with g''' being a residual of g'. In symbols:*

$$g''' \in (\{g'\}/(g''\ [[]g']))_{\Delta_1 \ldots \Delta_n}$$

Definition 3.4.5 *The graph g' is* free *by g if it is not captured.*

Intuitively, a graph g is captured by a graph g' if part of its structure is accessible within the *value* of the graph (i.e., its normal form $\hat{g'}$), in such a way that part of $\hat{g}$ can be accessed by a suitable selection function. However, graph inclusion does not correspond to capturings: occurrences of g within the condition part of an *if*-expression does not correspond to a capture, since there is no way to extract from the conditional any information on the condition; i.e., the conditional uses g internally but no subpart of g is retained as part of the value for the conditional.

Definition 3.4.6 *Given a graph g with sub-graph g', g' is* directly shared *in g if it has more than one parent graph node, and the labels of all its parents differ.*

Definition 3.4.7 *Given a graph g with sub-graph g', g' is* single-threaded *in g if for all possible reduction paths any residual of g' that is target of a mutation is not directly shared; e.g., If $C[g'] \rightarrow_{\Delta_1 \ldots \Delta_n} C'[g'']$, such that $g'' \in (C[g']/\{g'\})_{\Delta_1 \ldots \Delta_n}$, then g''' is the target of a mutator if, and only if, it is not directly shared. A graph g is single-threaded if all its subgraphs are single-threaded in g.*

A sufficient condition for a graph to have confluent reduction paths is that it be single-threaded.

3.5 Abstract Uses

Let R, F, and S denote the properties of being *read-only*, *free*, and *single-threaded*, respectively. These are the properties of λ_{st}-graphs introduced in the previous section. Each of them provides information about an operational aspect of the computation. However, the *combination* of the three properties will provide a better representation of the actual manipulation of the graphs. Some objects may possess more than one property (a conjunction), and some may possess either of several (a disjunction, arising from indeterminate contexts, when alternative evaluations are possible). In other words, I wish to consider these three properties closed over conjunction ($\wedge$) and disjunction ($\vee$). Alternatively, properties can be characterized by the set of objects that satisfy the property. From this point of view, properties become sets. Conjunction becomes set intersection, and disjunction becomes set union. The property that is not satisfied by any object is the empty set ($\varnothing$), and the one that is satisfied by all objects is the universal set (U). This algebra has 20 different elements if it is assumed that the three sets have non-trivial unions and intersections pairwise. The elements of the algebra along with the inclusion relation form a lattice, which is shown in Figure 3.8.

However, many of the resulting 20 properties are indistinguishable; for example, an object written in a multiple-threaded context results in error, regardless of whether it is captured. The derived properties that are distinguishable are called *abstract uses* (or just *uses*), and are described below:

1. The use $\perp$ denotes no use at all (i.e. *False*, or $\varnothing$).

2. The use rs denotes read-only, free, and single-threaded (i.e. $R \wedge S \wedge F$). Aside from no use at all, this represents the least "degree of use" of an object.

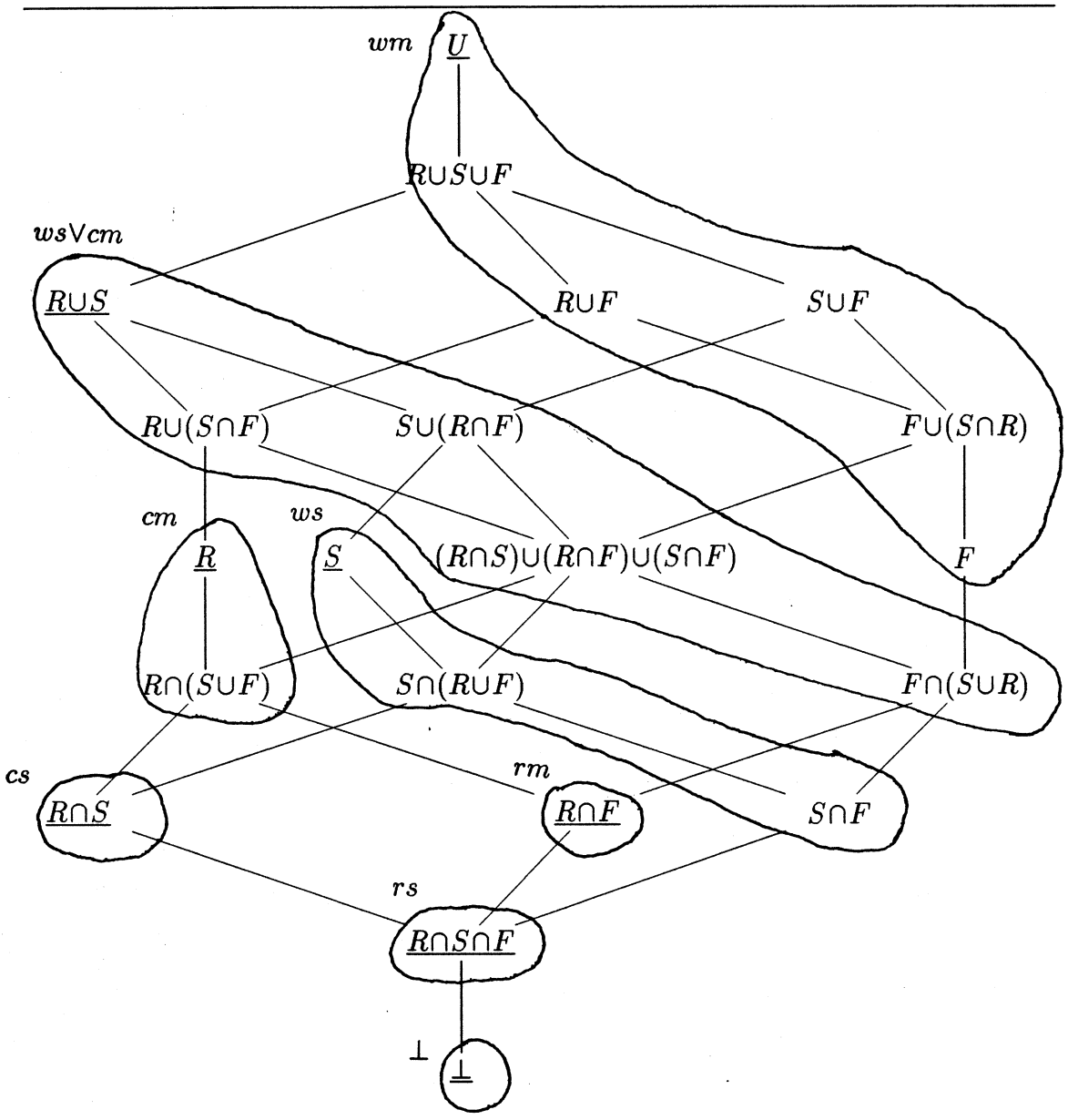


Figure 3.8: Lattice of Algebra of R, S, and F

3. The use rm denotes read-only and free, with no constraint on linearity (i.e. $R \wedge F$).

4. The use cs denotes either rs, or read-only and single-threaded (i.e. $R \wedge S$). This is a more general use than rs in that it allows an object to be captured as long as it is single-threaded (thus the name cs).

5. The use cm denotes read-only (i.e. R), but now with no constraints on capturing or linearity; thus the name cm to denote the possibility of "captured and multiple-threaded."

6. The use ws denotes single-threaded (i.e. S). The name ws denotes the new possibility of "written and single-threaded," since no other consideration is made on the fact that it could be captured on written.

7. The use $ws \vee cm$ (i.e. $R \vee S$) is self-descriptive, and arises in indeterminate contexts (such as in a conditional).

8. The use wm denotes no constraints (i.e. *True*, or **U**) and thus the potential error "write and multiple-threaded."

Treating the boolean domain as a lattice with *False* $\sqsubset$ *True*, these 8 properties also form a lattice, as shown in Figure 3.9. The structure of this lattice results from collapsing the full lattice—each property is clustered by the least use stronger than itself. Figure 3.8 has the properties corresponding to uses underlined.

3.6 Abstract Liabilities

Abstract Liability, or simply, liability, is a technique through which all free identifiers of an expression are associated with the least abstract use they satisfy. It is meant to represent the way the expression manipulates each identifier visible from its environment.

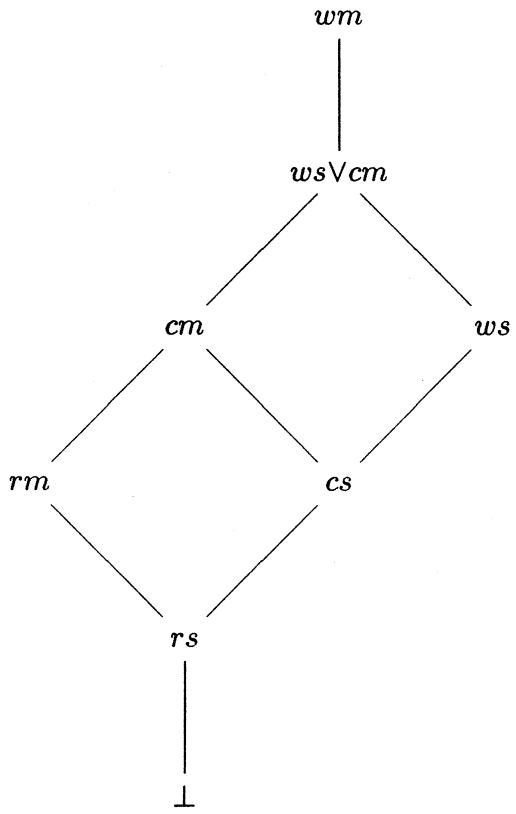


Figure 3.9: The Domain of Abstract Uses

A liability is an environment mapping identifiers to abstract uses:

$$L \in Liab = (Ide + \{\xi\}) \rightarrow Use$$

The most trivial liability maps all variables, and the anonymous object to $\perp$, and is denoted by $\perp_{Liab}$:

$$\perp_{Liab} \in Liab$$
$$\perp_{Liab}\ \xi = \perp$$
$$\perp_{Liab}\ v = \perp$$

Liabilities are built up incrementally using an environment update operation on them:

$$\begin{aligned} \_[\_ \mapsto \_] : \quad & Liab \to (Ide + \{\xi\}) \to Use \to Liab \\ L[v \mapsto u]\ v &= u \\ L[v_1 \mapsto u]\ v_2 &= L\ x \end{aligned}$$

with the following conventions:

$$\begin{aligned} L[x_1 \mapsto u_1, \ldots, x_n \mapsto u_n] &\equiv (\ldots(L[x_1 \mapsto u_1])\ldots)[x_n \mapsto u_n] \\ L^{v_1 \ldots v_n} &\equiv (\ldots(L[x_1 \mapsto \ \bot])\ldots)[x_n \mapsto \ \bot] \end{aligned}$$

The liability of a constant such as 1 is just $\bot_{Liab}$—i.e. the liability that maps every identifier to the use $\bot$. The liability of the expression `x` is just $\bot_{Liab}[x \mapsto cs]$, as is the liability of `x+x` (since `+` does not capture or mutate its arguments). The liability of `(update! a i x)` is $\bot_{Liab}[a \mapsto ws,\ i \mapsto rs,\ x \mapsto cs]$, since the array `a` is mutated, the index `i` is read in order to index into the array (but in no other way forms part of the result), and the object `v` is captured in the result, since a suitable lookup to the resulting array will return the value of `x`.

But what is the liability of an expression such as `(f x)`? Clearly it depends on how `f`, as a function, manipulates its formal parameter. Suppose it is determined that `f`'s use for its formal parameter is ws, then the liability of `(f x)` would be $\bot_{Liab}[f \mapsto cs,\ x \mapsto ws]$ (yes, the way functions are used is also controlled, since applying a function may return an object captured by the function).

3.6.1 Anonymous Objects

Certain expressions, like *mkarray*, generate and manipulate objects which are not associated with any variable. Note that, in any expression level, only named objects can be shared among its subexpressions. Therefore, anonymous objects generated in different subexpressions *must be different.* We use this fact to control with a single abstract use the potential manipulation of all anonymous objects generated in the expression. To that effect, we introduce the special symbol ξ not present in the Var domain.

Example The elements of the array that the expression `(mkarray i x)` generates are all peers of the object `x`, i.e., all elements of the array share the same graph. However, the array structure itself is generated by the function—not received as parameter. In operational terms, everytime this function is reduced, a *new* structure is generated. In fact

```
f (mkarray i x) (mkarray i x)
```

generates two identical, but distinct, arrays. The use of the anonymous object is *cs* since both of them capture an anonymous object, but each one of them captures a *different* one. Furthermore,

```
f (update! (mkarray i x) j y) (mkarray i x)
```

mutates the first array but does not mutate the second one. The use of the anonymous object is *ws* in this case. Even in the case where both arrays were modified like in

```
f (update! (mkarray i x) j y) (update! (mkarray i x) j z)
```

the use of the anonymous object remains *cs* since each of the updates modifies a *different* anonymous object. In fact, since these arrays are not associated with any variable, they can only be manipulated in this context by their super-expressions.

3.6.2 Combining Liabilities

A liability maintains a record of how the free identifiers of an expression are used. Note that the way these identifiers are used in any expression is a consequence on how they get used in the evaluation of constituent subexpressions, and how their evaluation interrelate—the evaluation of a subexpression may prevent the evaluation of another, etc. In this thesis, three different interrelations are considered:

- Parallel—the expressions can be reduced in any order (e.g., the function and argument in an application).

- Sequential—one expression is completely reduced before another (e.g., the predicate is fully reduced before either branch of a conditional).

- Alternate—the reductions are mutually exclusive (e.g., the alternate arms of a conditional).

These interrelations are abstracted in basic operations that describe parallel, sequential, and alternate evaluation.

Determining sequential evaluation is very difficult in lazy languages, since its model of computation specifies that only data dependencies determine the relative evaluation ordering of the expressions.

Parallel Evaluation This is the most general of all operations and assumes no knowledge of the relative order of the evaluation of the subexpressions other than both may eventually be evaluated. This operator is used in function applications to combine the liability of the function to that of the argument, since their relative evaluation order could range from the function being fully evaluated without the argument being demanded to the argument being fully evaluated *before* any reduction is performed in the function.

Sequential Evaluation In order to guarantee that one expression is evaluated before another, there must be a sequential constraint that forces one of the expressions (the *first* subexpression) to be fully evaluated before the other is considered for reduction. This is specially difficult to enforce in a normal order reduction strategy. However, the predicate part of the conditional is one such expression that satisfies the forementioned constraint. Note that in the case that the first subexpression is captured in the value of the expression, it must be the case that no further reductions whose root is inside (a residual of) the subexpression can take place. In particular, some functions can be reduced to normal form, but when they are later passed an argument, they continue to reduce, therefore they do not satisfy this restriction.

Alternate Evaluation Alternate evaluation is easier to determine: it arises at alternate branches of a conditional expression, whenever evaluating one branch precludes evaluating others. This is enforced for expressions properly boxed (those produced by function $\mathcal{B}$).

Figure 3.10 shows a collection of operators that implement these operations on the domain of uses. These functions are actually quite intuitive; $\overset{alt}{\odot}$ is, as expected, just the least upper bound operation on the domain of uses, since it merges information of alternative threads of control. $\overset{par}{\odot}$ and $\overset{seq}{\odot}$ are different ways to combine tasks in the same thread of control. The function $\uparrow$ is used in these operators simply to ensure that the linear component of the result is not single-threaded in the event that both arguments have a single-threaded use. These three operators are extended pointwise to the domain of liabilities:

$$\begin{aligned} \_\odot\_: &\quad Liab \to Liab \to Liab \\ (L_1 \odot L_2)\ v &= (L_1\ v) \odot (L_2\ v) \\ (L_1 \odot L_2)\ \xi &= (L_1\ \xi) \sqcup (L_2\ \xi) \end{aligned}$$

Note that all operations consider the anonymous object as a special case—they just perform a least upper bound on the use of the anonymous object of both operands. The reason is that anonymous objects cannot be aliased, and thus, the argument liabilities are referring to *different* anonymous objects, so the upper bound provides a use that is sure to encompass both uses.

Projection The final operation on abstract uses is that which is induced when a function is applied; a function whose formal parameter has use u "projects" this use onto the identifiers associated with the argument. The operator $\_\cdot\_$ is introduced for this purpose. Intuitively, the use of a variable in the argument gets 'corrected' by the use the function makes of the argument. This correction takes into account that variables captured in the argument are subject to mutation by the function if the function mutates its argument. Also if the function just reads the argument,

$$\begin{aligned}
\overset{alt}{_\odot_},\ \overset{seq}{_\odot_},\ \overset{par}{_\odot_} &: Use \to Use \to Use \\
\uparrow _ &: Use \to Use \\
_ \cdot _ &: Use \to Use \to Use
\end{aligned}$$

$$u_1 \overset{alt}{\odot} u_2 = u_1 \sqcup_{Use} u_2$$

$$\begin{aligned}
u_1 \overset{seq}{\odot} \bot &= u_1 \\
\bot \overset{seq}{\odot} u_2 &= u_2 \\
rs \overset{seq}{\odot} u_2 &= u_2 \\
rm \overset{seq}{\odot} u_2 &= rm \sqcup_{Use} u_2 \\
u_1 \overset{seq}{\odot} u_2 &= (\uparrow u_1) \sqcup_{Use} (\uparrow u_2) \quad otherwise
\end{aligned}$$

$$\begin{aligned}
u_1 \overset{par}{\odot} \bot &= u_1 \\
\bot \overset{par}{\odot} u_2 &= u_2 \\
u_1 \overset{par}{\odot} u_2 &= (\uparrow u_1) \sqcup_{Use} (\uparrow u_2) \quad otherwise
\end{aligned}$$

$$\begin{aligned}
\uparrow \bot &= \bot \\
\uparrow u &= rm && \text{if } u \sqsubseteq rm \\
\uparrow u &= cm && \text{if } u \sqsubseteq cm \\
\uparrow u &= wm && otherwise
\end{aligned}$$

$$\begin{aligned}
u \cdot \bot &= \bot \\
u \cdot rs &= rs \\
\bot \cdot u &= u && \text{if } ws \sqsubseteq u \\
\bot \cdot u &= rs && otherwise \\
cs \cdot u &= u \\
u \cdot cs &= u \\
cm \cdot cm &= cm \\
ws \cdot ws &= ws \\
u_1 \cdot u_2 &= wm && otherwise
\end{aligned}$$

Figure 3.10: Operations on Abstract Uses

any variable that was captured in the argument is read-only by the application. Its complete definition appears in figure 3.10. As with the first three, these operations are extended pointwise to liabilities:

$$\begin{array}{rcl} \_\cdot\_: & & Use \to Liab \to Liab \\ (u \cdot L)\ v & = & u \cdot (L\ v) \end{array}$$

The anonymous object is not treated as a special case for projection since its use by the argument must be corrected by the function in the same way as any named object.

Note that all of the above operations are monotonic with respect to the domain ordering given earlier for abstract uses.

3.7 Conclusions

In this chapter, Single-Threaded Lambda Calculus was introduced. This is a computational model that allows the user to reason about sharing and object reuse *without* the introduction of a formal store. Thus, the user has a simple way to *reason* about efficiency. Also, explicit sharing introduced by the user is not dependent on a particular implementation.

However, reasoning about object reusability has its price. It complicates the model of computation with temporal (sequential) restrictions on the reduction process of the graph. These restrictions have been absent from λ-calculus, or graph reduction. Even with this extra complexity, it is only promisory to write confluent programs, but there is no guarantee yet that the program is confluent; e.g., the underlying semantics, graph reduction, is not guaranteed to be confluent for what is usually regarded "well-formed expressions" in λ-calculus, or admissible graphs in graph reduction.

A new notion of well-formedness is needed. One that can guarantee only to be satisfied by confluent graphs. The approach taken in this dissertation is to enforce a

type system on graphs such that graphs that satisfy the type discipline are guaranteed confluent. The discipline must involve an abstraction of some operational properties of graphs, such as sequentiality constraints, as well as which graphs may be subject to reutilization. These properties, as well as their corresponding abstractions, and a technique to operate on them were introduced in Section 3.5, and 3.6. The type system extended to utilize such properties will be introduced in Chapter 4.

Chapter 4

Extended Type System

4.1 Introduction

A type discipline for the λ_{st}-calculus is introduced in this chapter. This takes the form of an extension to the Hindley-Milner type system [Hindley, 1978,Milner, 1978]. The very notion of type is extended to allow functional types to encode the abstract use of their arguments, and, in addition, the notion of liability of an expression is provided. The type and liability properties of an expression are considered inseparable, and the rules reflect this fact by associating any expression with its type and liability simultaneously.

As the reader will find out when the inference rules are presented in Section 4.4, types and liabilities are intertwined: the liability of an expression may be affected by the type of constituent subexpressions (i.e., in function applications), and the type of an expression depends on the liability of its subexpressions (i.e., in λ-abstractions). Thus changing the liability facet of this analysis would affect the type facet. The type aspect of the analysis is a well known problem, and I have chosen to base the type system in this thesis on the one originally proposed by Hindley [Hindley, 1978] and Milner [Milner, 1978]. The liability aspect, on the other hand, is my contribution, and is loosely based on the Effect Analysis introduced by Lucassen and Gifford [Gifford

and Lucassen, 1986,Lucassen and Gifford, 1988].

With respect to the liability aspect of the system, it should be noted that the liability presented in the thesis was chosen due to its simplicity, and that other feasible choices, however more complex, do exist.

It is desirable for any type system to have a notion of *principal type* property—any expression that can be properly typed by the system can be given a type from which all other types can be derived. It will be shown on Chapter 6 that the revised system introduced in this chapter indeed has a principal type property.

For ease of understanding, the presentation of extended type system has been divided into two stages. Initially, only constant abstract uses allowed inside type expressions. Such a type system is shown in Sections 4.2, 4.3, and 4.4. Section 4.5 shows several examples of extended type reconstruction using the simple system, as well as inherent limitations—it can infer principal first-order types based on assumptions of higher-order types, but it cannot infer principal higher-order types.

Sections 4.6, 4.7, and 4.8 establish a revision to the extended type system by introducing abstract use variables. This type system is capable of producing principal higher-order types as well. Finally, Section 4.9 presents a formal proof that the typing rules are sound.

4.2 Preliminaries

A static type system characterizes the type of values used in programs; i.e., it is concerned with properties *satisfied* by the values in the programs. A liability system on the other hand, is concerned with *how* the values are *used* in the program: i.e., how often they are used, whether they are mutated, etc. In the type system hereby presented a decision was made to qualify with a single use each function's formal argument. This corresponds to associating a use with each free variable in the expression. In addition, it has been necessary to associate a single use with all graphs that are manipulated in the expression, but that cannot be associated with any variable, thus

$$
\begin{array}{rcll}
\sigma \in TypeSch & ::= & \mu & \\
 & & \mid \forall\alpha\sigma & \textit{Type Schemes} \\
 & & & \\
\mu \in MutType & ::= & \tau & \\
 & & \mid Array\ \tau & \textit{Arrays} \\
 & & & \\
\tau \in Type & ::= & Int \mid Bool \mid \ldots & \textit{Basic Types} \\
 & & \mid Pair\ \tau_1\ \tau_2 & \textit{Pairs} \\
 & & \mid \mu_1 \rightarrow \mu_2 & \textit{Functions} \\
 & & \mid \alpha & \textit{Type Variables}
\end{array}
$$

Figure 4.1: Syntax of Type Expressions

called *anonymous objects.* Therefore, liabilities associate abstract uses with named, as well as anonymous, objects.

4.2.1 Type Expressions

The usual syntax for Hindley-Milner types with data structures is adopted, and is shown in Figure 4.1. In the syntax, *Type* is the domain of *immutable monomorphic types, MutType* is the domain of *mutable monomorphic types*, and *TypeSch* is the domain of *type schemes*, or *shallow polymorphic mutable types.* Shallow polymorphic types are those in which type variables may be quantified (thus introducing polymorphism by separate instantiation of the quantified type variable), with the restriction that quantification has to affect the whole type expression; i.e., quantifiers can only appear in the outermost level of the expression. This, in fact, is a restriction over the language of polymorphic types, which has proven to be natural, simple, and effectively computable.

4.2.2 Extended Type Expressions

In order to correctly capture the behavioral aspect of a function within its type, the language of types is extended to annotate the function constructor ($\rightarrow$) with the abstract use of the bound variable above, and the use of anonymous objects below, thus resulting in *extended types.* In these paragraphs, the expressiveness of this addition is explored, and restrictions to this extension (akin to shallow typing) that capture the same "interesting" extended types while being "better behaved" are introduced. The syntax of abstract use expressions and extended type expressions is given in Figure 4.2. I will make the distinction of the sublanguage of types which does not involve functions, defined in Figure 4.3. Note that $\hat{\sigma}$ ranges over $TypeSch_{Ext}$, whereas $\tilde{\sigma}$ ranges over $\widetilde{TypeSch}_{Ext}$; i.e., the former can be instantiated to any type, while the latter can only be instantiated to types that do not contain functions.

Coercions restrict the range of values over which types hold: a type is valid only if every coercion in its coercion set is satisfiable. extended types may have coercion sets at any level there is a bound use variable. This leads to a very general language of extended types.

Type assumptions (T) are environments that map identifiers to types. They are used in the type inference rules to maintain the types of all free identifiers.

First-order types are those that can be associated with first-order values. A type is a *higher-order type* otherwise.

4.3 Extended Type Instantiation/Coercion Rules

The language of types defined in the previous section naturally lends itself to a hierachy of the type domain. On one hand, a type may be "specialized", or *instantiated* when a bound type variable is instantiated to a type expression, or when a bound use variable is instantiated to a use expression. Consider the identity function's extended

$$u \in AbsUse ::= \bot \mid rs \mid rm \mid cs \mid cm \mid ws \mid ws \vee cm \mid wm \quad \textit{Abstract uses}$$

$\hat{\sigma} \in TypeSch_{Ext}$	::=	$\langle \hat{\mu}, C \rangle$	
		$\mid \forall \alpha \hat{\sigma}$	*Type Schemes*
$\hat{\mu} \in MutType_{Ext}$	::=	$\hat{\tau}$	
		$\mid Array\ \hat{\tau}$	*Arrays*
$\hat{\tau} \in Type_{Ext}$	::=	$Int \mid Bool \mid \ldots$	*Basic Types*
		$\mid Pair\ \hat{\tau}_1\ \hat{\tau}_2$	*Pairs*
		$\mid \hat{\mu} \xrightarrow[u_2]{u_1} \hat{\mu}$	*Functions*
		$\mid \alpha$	*Type Variables*
$c \in Coercion$	::=	$u_1 \trianglerighteq u_2$	*Coercion on uses*
		$\mid \hat{\mu}_1 \trianglerighteq \hat{\mu}_2$	*Coercion on types*
$C \in CSet$	::=	$\{c_1, , \ldots, c_n\}$	*Coercion Sets*
$\nu \in Id_{Use}$			*Use Variables*
$\alpha \in Id_{Type}$			*Type Variables*

Figure 4.2: Abstract Uses, and Extended Type Expressions

$$
\begin{array}{lll}
\tilde{\sigma} \in \widetilde{TypeSch}_{Ext} ::= & \langle \tilde{\tau}, C \rangle & \\
& \mid \forall \alpha \tilde{\sigma} & \\
\\
\tilde{\mu} \in \widetilde{MutType}_{Ext} ::= & \tilde{\tau} & \\
& \mid Array\ \tilde{\tau} & \textit{Arrays} \\
\\
\tilde{\tau} \in \widetilde{Type}_{Ext} \subseteq Type_{Ext} ::= & Int \mid Bool \mid \ldots & \textit{Basic Types} \\
& \mid Pair\ \tilde{\tau}_1\ \tilde{\tau}_2 & \textit{Pairs} \\
& \mid \tilde{\alpha} & \textit{Type Variables}
\end{array}
$$

Figure 4.3: Extended Type Expressions not Containing Functions

type

$$\forall \alpha.\alpha \xrightarrow{cs} \alpha \tag{4.1}$$

which means that the function takes an argument of any type and returns an object an object of the *same* type as the argument's, and that the argument's structure is 'captured' within the returned value.[1] More specialized types for this function are:

$$Int \xrightarrow{cs} Int \tag{4.2}$$

$$\forall \alpha.(Array\ \alpha) \xrightarrow{cs} (Array\ \alpha) \tag{4.3}$$

$$(\alpha \xrightarrow{ws} \alpha) \xrightarrow{cs} (\alpha \xrightarrow{ws} \alpha) \tag{4.4}$$

This specialization is known as type instantiation and is present in the Hindley-Milner type system. The type (4.2) is *less general* than the type (4.1) since all instantiation of type (4.2) are also instantiations of type (4.1), but not vice versa.

On the other hand, the type signature

$$\forall \alpha.\alpha \xrightarrow{ws} \alpha \tag{4.5}$$

[1] In fact, it *is* the returned value.

is also a valid type for the identity function. This type indicates that the identity *may* mutate its argument, although it actually does not. As before, (4.5) is less general than (4.1), but in this case, it is said that (4.1) *coerces* to (4.5). This follows because it is true that $cs \sqsubseteq ws$ in the use domain. Therefore, if it can be asserted that the function may capture its formal argument, then it can be asserted that it mutate it—even when it actually does not. This notion of coercion involves knowledge of a partial order on types since non-variable parts are replaced in the type expressions. It is known as *subtyping*, and is not expressible within the Hindley-Milner type system. Instead, an explicit set of rules, the *coercion rules*, are given which specify the valid coercions. This relation subsumes instantiation.

In the case of the extended type system, coercions embed within the type structure the partial order given for the abstract uses. This subtyping relation percolates through the functional types, since they are the ones that first introduce uses into the system.

4.3.1 Substitution/Instantiation

Substitution on extended types is done in the usual way, and thus its definition is omitted. Uses annotating the function types are not substituted for.

$$\_[\_/\_]:(TypeSch_{Ext} \rightarrow TypeSch_{Ext} \rightarrow Id_{Ext} \rightarrow TypeSch_{Ext})$$

It is usually the case that the same substitution is applied to several expressions, then it is natural to abstract the first operand from the definition resulting in a function usually called S.

4.3.2 Coercion Rules

The notation $\tau_1 \trianglerighteq \tau_2$ (τ_1 *coerces to* τ_2) is valid when the set of semantic values associated with type τ_1 is contained in the set of semantic values associated with τ_2. Similarly for $u_1 \trianglerighteq u_2$. A *coercion set* C is just a set of coercions on types and coer-

cions on abstract uses. A substitution (S) is a mapping from type variables to types expressions, and from use variables to use expressions.

Also, the notation $C_1 \Vdash C_2$ is used to indicate that for all substitutions that cause all coercions in C_1 to be valid also satisfy all coercions in C_2. The first three rules just state basic properties on coercion: it subsumes instantiation, and is a reflexive, and transitive property.

1. Instantiation

$$\frac{S\hat{\mu}_1 = \hat{\mu}_2 \quad S\ C = C}{C \Vdash \{\hat{\mu}_1 \trianglerighteq \hat{\mu}_2\}}$$

   If $\hat{\mu}_2$ is an instance of $\hat{\mu}_1$ under a substitution that does not affect the coercion set C, then $\hat{\mu}_1$ can be coerced to $\hat{\mu}_2$.

2. Reflexivity

$$C \Vdash \{u \trianglerighteq u\}$$

$$C \Vdash \{\hat{\mu} \trianglerighteq \hat{\mu}\}$$

3. Transitivity

$$\frac{C \Vdash \{u_1 \trianglerighteq u_2, u_2 \trianglerighteq u_3\}}{C \Vdash \{u_1 \trianglerighteq u_3\}}$$

$$\frac{C \Vdash \{\hat{\mu}_1 \trianglerighteq \hat{\mu}_2, \hat{\mu}_2 \trianglerighteq \hat{\mu}_3\}}{C \Vdash \{\hat{\mu}_1 \trianglerighteq \hat{\mu}_3\}}$$

4. Abstract Uses

$$C \Vdash \{u_1 \trianglerighteq u_2\}, \mathit{if}\ u_1 \sqsubset u_2$$

   Coercion on uses corresponds to domain ordering on them (Figure 3.9).

5. Arrays

$$\frac{C \Vdash \{\hat{\tau}_1 \trianglerighteq \hat{\tau}_2\}}{C \Vdash \{(\mathit{Array}\ \hat{\tau}_1) \trianglerighteq (\mathit{Array}\ \hat{\tau}_2)\}}$$

If the type of the elements of an array is coercible to the type of the elements of a second array, then the type of the first array is coercible to the type of the second one.

6. Pairs

$$\frac{C \Vdash \{\hat{\tau}_{11} \trianglerighteq \hat{\tau}_{21}\} \quad C \Vdash \{\hat{\tau}_{12} \trianglerighteq \hat{\tau}_{22}\}}{C \Vdash \{(Pair\ \hat{\tau}_{11}\ \hat{\tau}_{12}) \trianglerighteq (Pair\ \hat{\tau}_{21}\ \hat{\tau}_{22})\}}$$

Similarly for pairs, if the type of each of the components of a pair is coercible to the type of the corresponding component of the second pair, then the type of the first pair is coercible to the type of the second pair.

7. Functions

$$\frac{C \Vdash \{\hat{\mu}_{21} \trianglerighteq \hat{\mu}_{11}, \hat{\mu}_{12} \trianglerighteq \hat{\mu}_{22}, u_{11} \trianglerighteq u_{21}, u_{12} \trianglerighteq u_{22}\}}{C \Vdash \{\hat{\mu}_{11} \overset{u_{11}}{\underset{u_{12}}{\rightarrow}} \hat{\mu}_{12} \trianglerighteq \hat{\mu}_{21} \overset{u_{21}}{\underset{u_{22}}{\rightarrow}} \hat{\mu}_{22}\}}$$

It has been known that function types are contravariant for standard subtyping disciplines [Mitchell, 1984]—if $\hat{\mu}_{22} \trianglerighteq \hat{\mu}_{21}$ and $\hat{\mu}_{11} \trianglerighteq \hat{\mu}_{12}$, then $\hat{\mu}_{11} \rightarrow \hat{\mu}_{21} \trianglerighteq \hat{\mu}_{12} \rightarrow \hat{\mu}_{22}$. This inversion of the direction of the relation is known as contravariance—coercion is contravariant on the types of the domains of functions. However, it is *not* contravariant on the uses.

8. Type Schemes

$$\frac{C \Vdash \{\hat{\sigma}_1 \trianglerighteq \hat{\sigma}_2\}}{C \Vdash \{\forall\alpha.\hat{\sigma}_1 \trianglerighteq \forall\alpha.\hat{\sigma}_2\}}$$

Coercion is generalized to type schemes in a straight-forward way: if types are coercible, then quantifying them by the *same* variable retains the property.

4.4 Type/Liability Inference Rules

In this section, the type/liability inference rules for λ_{st}-calculus are introduced. These rules resemble those of Hindley-Milner, with extensions to deal with liabilities. The

notation $C,T \vdash e{:}\langle\hat{\sigma}, L\rangle$ asserts that under use and type coercion set C, and type assumptions T, the expression e has type $\hat{\sigma}$ and liability L.

In addition to type expressions, we have the usual domain of type assumptions:

$$T \in Typing = Ide \longrightarrow Type$$

We assume the presence of an environment $\mathcal{K}$ for typing constants of signature

$$\mathcal{K} : Kon \longrightarrow (Type \times Liab)$$

The types of a small group of constants used in this thesis are shown in Figure 4.4.

The commentary after each rule is limited to the liabilities and uses, since the types so closely resemble those of Hindley-Milner.

Note that the coercion set remains constant throughout the rules. However, the set appears in the rules, rather than being assumed as a constant (like the function $\mathcal{K}$) because it is particular to the program being inferred. Part of the job of the type reconstructor is to infer a suitable set that satisfies all coercions imposed by the application of the rules.

The rules presented assume that the objects being typed are programs, i.e., λ_{st}-trees. This may seem a drawback, but, in fact, it is not. Type reconstruction is a static analysis—done at compile time. As mentioned before, source programs in λ_{st}-calculus do not contain shared nodes, only information on reutilization of nodes. Sharing results as a direct consequence of graph reduction. Typing arbitrary λ_{st}-graphs, although interesting, is not really necessary since user programs can be adequately represented as trees.

1. Constants

$$C,T \vdash k : \mathcal{K}(k)$$

The type and liability of a constant are given by the function $\mathcal{K}$. The liability must map every identifier to $\perp$, but could have a non-$\perp$ mapping for the anonymous object, as in the case of `Nil`, or `1`, where the value of the constant is a "new" object.

$$
\begin{aligned}
\textbf{\textit{mkarray}}: &\ \langle Int \xrightarrow{rs} \alpha \xrightarrow[cs]{cm} (Array\ \alpha), \perp_{Liab}[\xi \mapsto cs]\rangle \\
\textbf{\textit{lookup}}: &\ \langle (Array\ \alpha) \xrightarrow{cs} Int \xrightarrow{rs} \alpha, \perp_{Liab}\rangle \\
\textbf{\textit{update}}: &\ \langle (Array\ \alpha) \xrightarrow{cs} Int \xrightarrow{rs} \alpha \xrightarrow{cs} (Array\ \alpha), \perp_{Liab}\rangle \\
\textbf{\textit{update!}}: &\ \langle (Array\ \alpha) \xrightarrow{ws} Int \xrightarrow{rs} \alpha \xrightarrow{cs} (Array\ \alpha), \perp_{Liab}\rangle \\
\\
\textbf{\textit{pair}}: &\ \langle \alpha \xrightarrow{cs} \beta \xrightarrow[cs]{cs} (Pair\ \alpha\ \beta), \perp_{Liab}\rangle \\
\textbf{\textit{fst}}: &\ \langle (Pair\ \alpha\ \beta) \xrightarrow{cs} \alpha, \perp_{Liab}\rangle \\
\textbf{\textit{snd}}: &\ \langle (Pair\ \alpha\ \beta) \xrightarrow{cs} \beta, \perp_{Liab}\rangle
\end{aligned}
$$

Figure 4.4: Type/Liability Signatures for Selected Constants

2. Identifiers

$$C, T \vdash v : \langle T(v), \perp_{Liab}[v \mapsto cs]\rangle$$

The liability of a lone identifier is *cs* (it's been used once in the expression, not mutated, and captured). The liability maps all other identifiers—and the anonymous object—to $\perp$.

3. Lambda Abstractions

$$\frac{C, T[v \mapsto \hat{\mu}_1] \vdash e : \langle \hat{\mu}_2, L[v \mapsto u_1, \xi \mapsto u_2]\rangle}{C, T \vdash (\lambda v.e) : \langle \hat{\mu}_1 \xrightarrow[u_2]{u_1} \hat{\mu}_2, L^{v\xi}\rangle}$$

If it is the case that when the bound variable v is of type $\hat{\mu}_1$ then e is of type $\hat{\mu}_2$, and liability L, associating v with use u_1, and the anonymous object (ξ) to use u_2, then the abstraction is of type $\hat{\mu}_1 \xrightarrow[u_2]{u_1} \hat{\mu}_2$, and its liability is like L, but does not contain entries for v and ξ; i.e., v and ξ are abstracted from the liability.

4. Applications

$$\frac{C,T \vdash e_1 : \langle \hat{\mu}_1 \xrightarrow[u_2]{u_1} \hat{\mu}_2, L_1 \rangle \quad C,T \vdash e_2 : \langle \hat{\mu}_1, L_2 \rangle}{C,T \vdash (e_1\ e_2) : \langle \hat{\mu}_2, (L_1 \overset{par}{\odot} (u_1 \cdot L_2)) \sqcup \bot_{Liab}[\xi \mapsto u_2] \rangle}$$

The type of the argument must match the type of the domain of the function; the function's use of the bound variable is projected onto the argument's liability, and then combined to the function's own liability. Since there is no temporal constraint on the function and argument evaluation, this combination is done using $\overset{par}{\odot}$.

There are uses for the anonymous object derived from the evaluation of the function, the evaluation of the argument, *and* from the application itself (u_2, the use of the anonymous object, retrieved from the function's type). The first two are embedded in the liabilities of the function and argument, and as such as combined as the corresponding liabities are. However, its use from the function application as such has to be explicitly combined: a liability that only contains a use for the anonymous variable is created ($L[\xi \mapsto u_2]$), and that liability is combined by means of the upper-bound operator to the rest of the computed liabilities.

5. Strict Application

$$\frac{C,T \vdash e_1 : \langle \tilde{\mu}_1 \xrightarrow[u_2]{u_1} \hat{\mu}_2, L_1 \rangle \quad C,T \vdash e_2 : \langle \tilde{\mu}_1, L_2 \rangle}{C,T \vdash \langle e_1\ e_2 \rangle : \langle \hat{\mu}_2, ((u_1 \cdot L_2) \overset{seq}{\odot} L_1) \sqcup \bot_{Liab}[\xi \mapsto u_2] \rangle}$$

The liability of strict application differs from the normal one in that the argument is assumed to be evaluated before the body; thus the use of $\overset{seq}{\odot}$. Note the restriction on the argument to not be a function (i.e. its type must be in $\widetilde{Type}$). This is to ensure that the argument can indeed be *completely* evaluated

before the application is made. Finally, the use u_2 to the anonymous object is accounted for in a manner analogous to Rule 4.

6. If-then-else

$$\frac{\begin{array}{c} C,T \vdash e_p : \langle Bool, L_p \rangle \\ C,T \vdash e_c : \langle \hat{\mu}, L_c \rangle \\ C,T \vdash e_a : \langle \hat{\mu}, L_a \rangle \end{array}}{C,T \vdash (if\ e_p\ e_c\ e_a) : \langle \hat{\mu}, (L_p \overset{seq}{\odot} (L_c \overset{alt}{\odot} L_a)) \rangle}$$

The liability of the conditional is that of the predicate sequentially combined with the alternate combination of the then- and else-branches.

7. Let

$$\frac{\begin{array}{c} C,T \vdash e_1 : \langle \hat{\sigma}, L_1 \rangle \\ C,T[v \mapsto \hat{\sigma}] \vdash e_2 : \langle \hat{\mu}, L_2[v \mapsto u_1, \xi \mapsto u_2] \rangle \end{array}}{C,T \vdash (let\ v = e_1\ in\ e_2) : \langle \hat{\mu}, (L_2 \overset{par}{\odot} (u_1 \cdot L_1)) \sqcup \bot_{Liab}[\xi \mapsto u_2] \rangle}$$

Under the standard semantics, the let construct is a combination of an abstraction and an application; i.e.,

$$(let\ v = e_1\ in\ e_2) \equiv ((\lambda v.e_2)\ e_1)$$

However, according to the Hindley-Milner type system, the "λ-bound" v is monomorphic (a mutable type), while the "*let*-bound" v is polymorphic (a type scheme). In the liability aspect, this rule just computes the combination of an abstraction and an application.

8. Let$*$

$$\frac{\begin{array}{c} C,T \vdash e_1 : \langle \tilde{\sigma}, L_1 \rangle \\ C,T[v \mapsto \tilde{\sigma}] \vdash e_2 : \langle \hat{\mu}, L_2[v \mapsto u_1, \xi \mapsto u_2] \rangle \end{array}}{C,T \vdash (let* \ v = e_1\ in\ e_2) : \langle \hat{\mu}, ((u_1 \cdot L_1) \overset{seq}{\odot} L_2) \sqcup \bot_{Liab}[\xi \mapsto u_2] \rangle}$$

Similarly to the previous rule, this rule is a combination of a an abstraction, and a strict application, except that the bound variable of the abstraction has a polymorphic type.

9. Fix

$$\frac{C,T \vdash e : \langle \hat{\mu} \xrightarrow[u_2]{u_1} \hat{\mu}, L\rangle}{C,T \vdash (fix\ e) : \langle \hat{\mu}, (u_1 \cdot L) \overset{alt}{\odot} \bot_{Liab}[\xi \mapsto u_2]\rangle}$$

The fixpoint operator encapsulates the self-application of its argument. As such, its liability is the liability of its argument (L) projected by its own use of the bound variable (u_1). In addition, the use u_2 to the anonymous object handled like in Rule 4.

10. Coercion

$$\frac{\begin{array}{c} C,T \vdash e : \langle \hat{\sigma}, L\rangle \\ C \Vdash \{\hat{\mu} \trianglerighteq \hat{\mu}'\} \end{array}}{C,T \vdash e : \langle \hat{\sigma}', L\rangle}$$

4.5 Examples of Type Reconstruction

It is instructive to see how these rules can be combined in a type derivation. Figure 4.5 shows the type reconstruction of (*swap*! *a i j*). The type derivation resembles a Hindley-Milner proof, except that, in addition, the liabilities are also constructed. It can be observed that the liability of the consequent of a proof step can be directly computed from the liabilities of its premises by just combining them with the binary operators on uses. However, the converse is not true: the same consequent liability can be implied by several configurations of the premises.

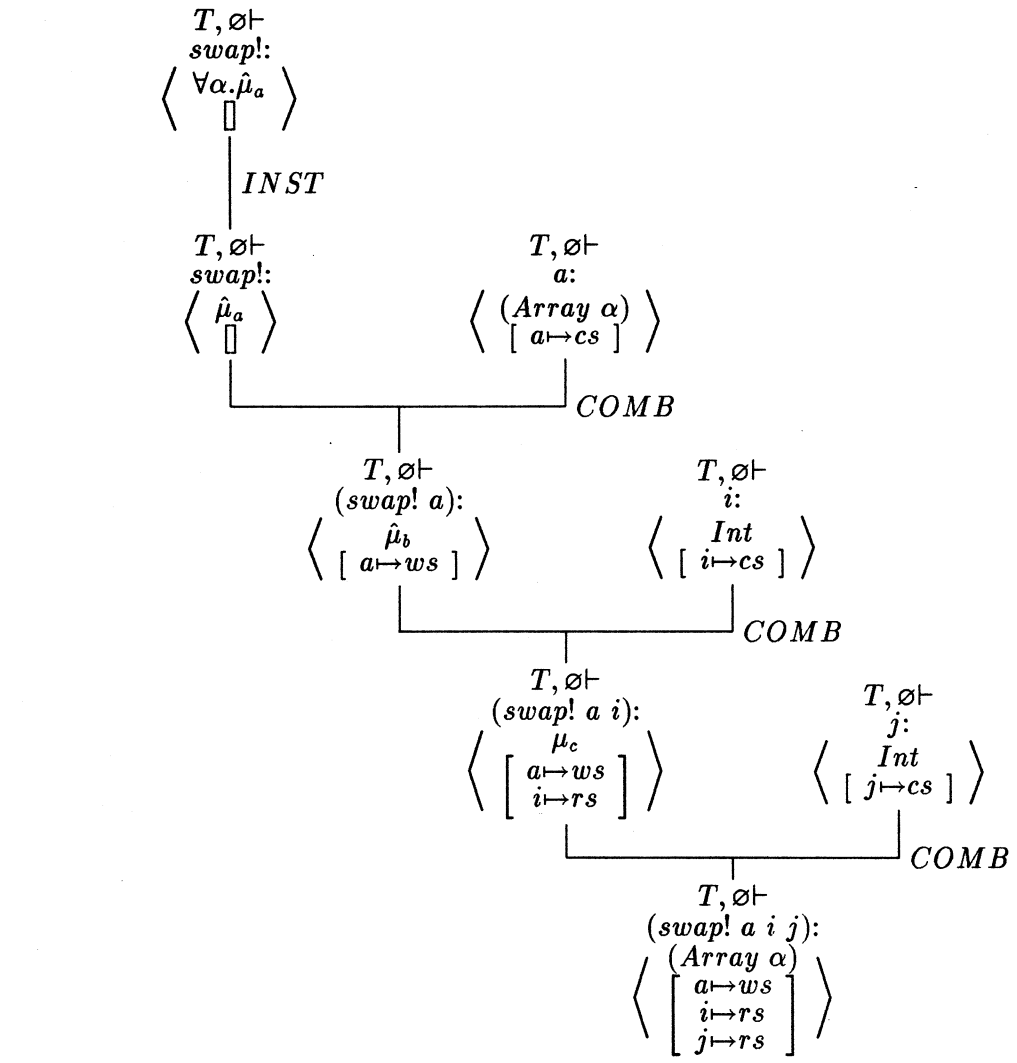


Figure 4.5: Derivation of Extended Type for (*swap*! *a i j*)

Consider the function *mapa!*

$$\begin{aligned} mapa! \equiv\ & fix\ \lambda mapa.\lambda f.\lambda a.\lambda n. \\ & \quad if\ (n = 0) \\ & \quad a \\ & \quad \langle \lambda x.(mapa!\ f\ (upd!\ a\ n\ (f\ x))\ (n-1))\ (lookup\ a\ i)\rangle \end{aligned}$$

This function applies the function f to every element of the array a, updating a in the process. Figure 4.6 highlights some of the derivation steps, including a strict application, and a fixpoint, to compute the type

$$\forall\alpha.(\alpha \xrightarrow{cs} \alpha) \xrightarrow{wm} (Array\ \alpha) \xrightarrow{ws} Int \xrightarrow{rs} (Array\ \alpha)$$

However, the type of f was assumed to be $\alpha \xrightarrow{cs} \alpha$, but what would be the type of *mapa!* if the type for the higher-order function f was assumed to be $\alpha \xrightarrow{ws} \alpha$? In order to ease the discussion on types for higher-order functions, it is instructive to study an example without recursion first. Consider now the type derivation of $g \equiv \lambda f.\lambda a.(f\ a)$ (Figure 4.8). Note that in order to be able to successfully type the program, the type assumed for f must be of the form $\alpha \xrightarrow[u_2]{u_1} \beta$, where u_1 and u_2 are elements of the domain of uses. The derivation shown assumes the type of f to be

$$\alpha \xrightarrow{ws} \beta$$

and hence, the derived type for the expression is

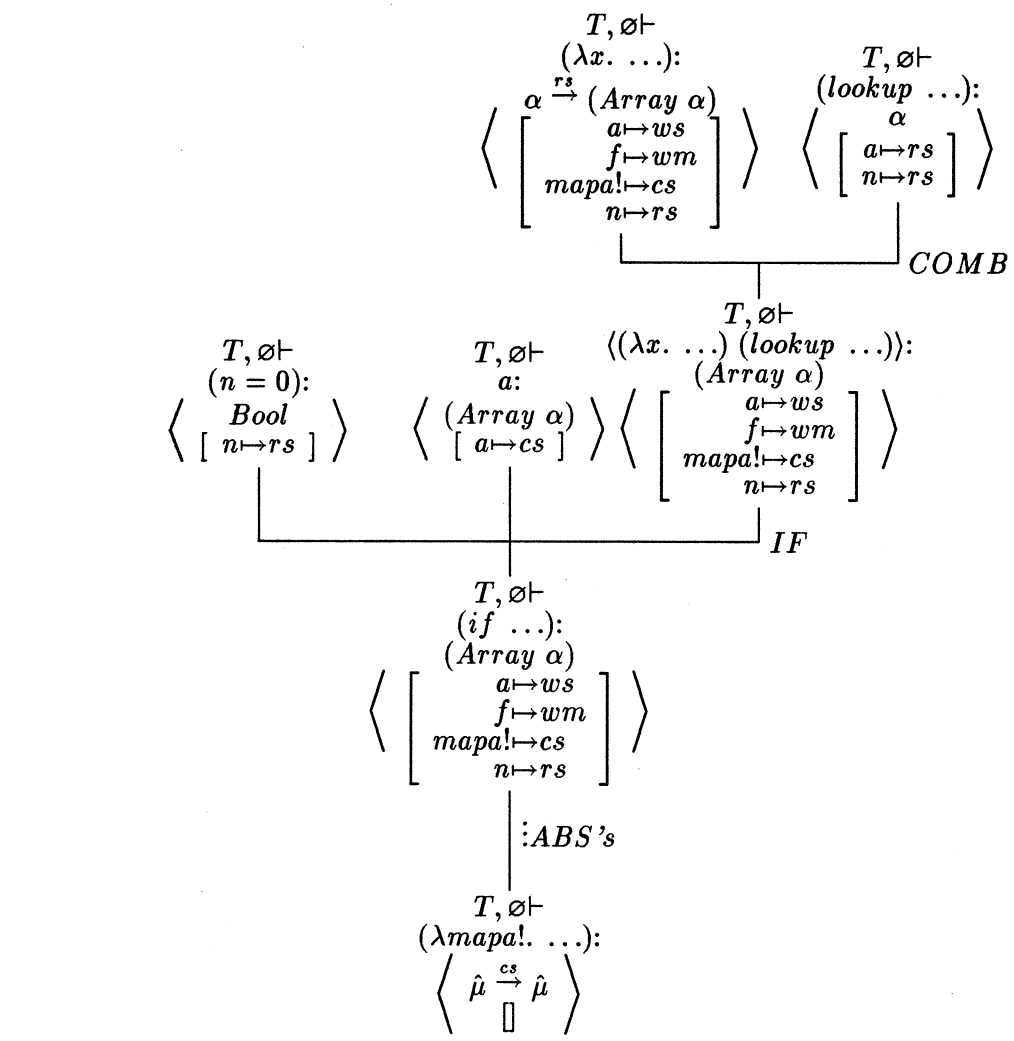
$$(\alpha \xrightarrow{ws} \beta) \xrightarrow{cs} \alpha \xrightarrow{ws} \beta$$

However, if the assumed type for f had been

$$\alpha \xrightarrow{rs} \beta$$

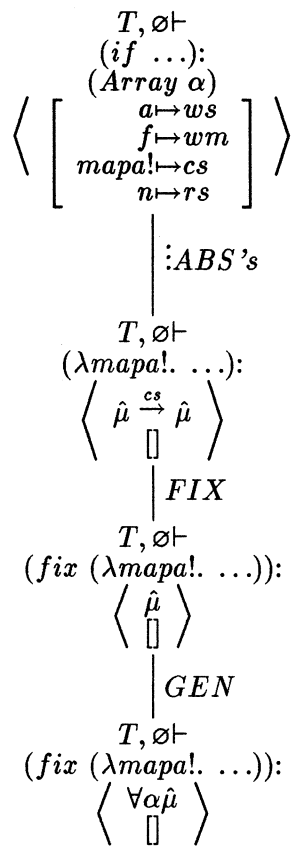
then the type of the expression would have been

$$(\alpha \xrightarrow{ws} \beta) \xrightarrow{cs} \alpha \xrightarrow{ws} \beta$$



$$
\begin{array}{rcl}
\hat{\mu} & \equiv & (\alpha \xrightarrow{cs} \alpha) \xrightarrow{wm} (Array\ \alpha) \xrightarrow{ws} Int \xrightarrow{rs} (Array\ \alpha) \\
T & \equiv & \left[\begin{array}{c} a \mapsto (Array\ \alpha) \\ f \mapsto \alpha \xrightarrow{rs} \alpha \\ mapa! \mapsto \hat{\mu} \\ n \mapsto Int \end{array}\right] \\
mapa! & \equiv & fix\ \lambda mapa.\lambda f.\lambda a.\lambda n. \\
 & & \quad if\ (n = 0) \\
 & & \quad a \\
 & & \quad \langle \lambda x.(mapa!\ f\ (upd!\ a\ n\ (f\ x))\ (n-1))\ (lookup\ a\ i) \rangle
\end{array}
$$

Figure 4.6: Extended Type Derivation of *mapa*! (part 1 of 2)



$$
\begin{array}{rcl}
\hat{\mu} & \equiv & (\alpha \xrightarrow{cs} \alpha) \xrightarrow{wm} (Array\ \alpha) \xrightarrow{ws} Int \xrightarrow{rs} (Array\ \alpha) \\
T & \equiv & \left[\begin{array}{r} a \mapsto (Array\ \alpha) \\ f \mapsto \alpha \xrightarrow{rs} \alpha \\ mapa! \mapsto \hat{\mu} \\ n \mapsto Int \end{array}\right] \\
mapa! & \equiv & fix\ \lambda mapa.\lambda f.\lambda a.\lambda n. \\
 & & \quad if\ (n = 0) \\
 & & \quad a \\
 & & \quad \langle \lambda x.(mapa!\ f\ (upd!\ a\ n\ (f\ x))\ (n-1))\ (lookup\ a\ i) \rangle
\end{array}
$$

Figure 4.7: Extended Type Derivation of *mapa*! (part 2 of 2)

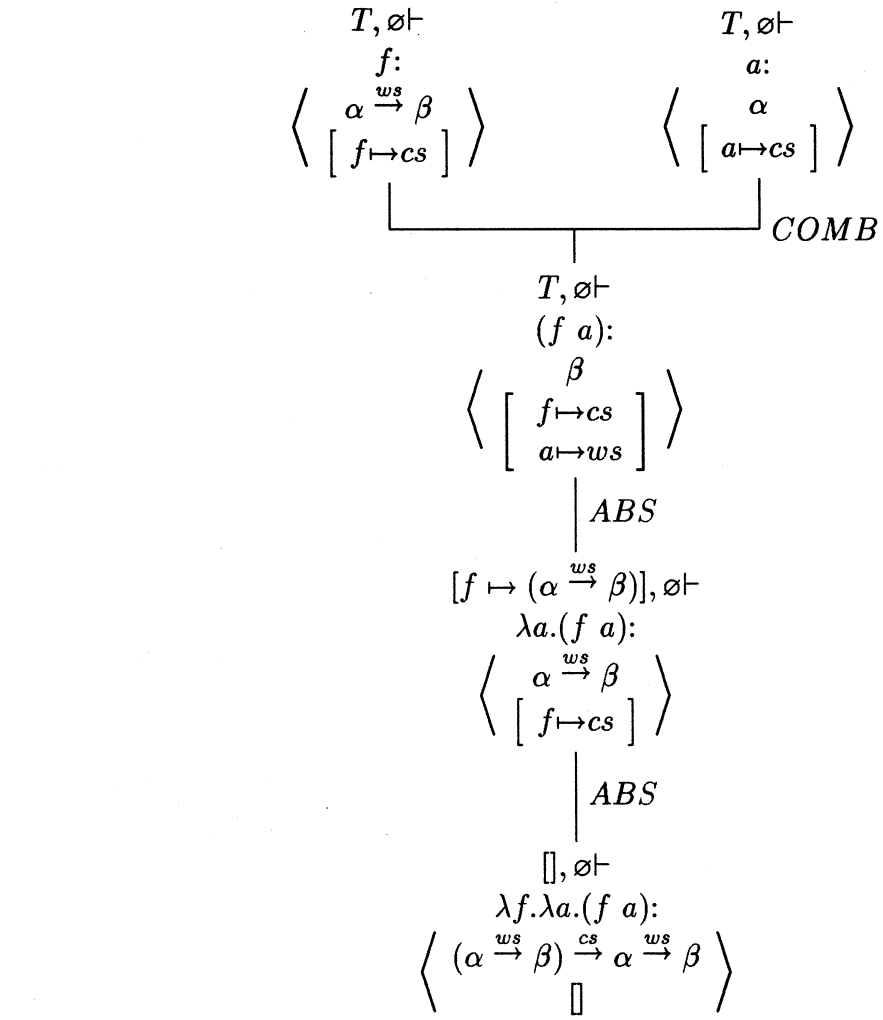


Figure 4.8: Derivation of Extended Type for $(\lambda f.\lambda a.(f\ a))$

or, in general,

$$(\alpha \xrightarrow[u_2]{u_1} \beta) \xrightarrow{cs} \alpha \xrightarrow[u_2]{u_1} \beta$$

where $u_1, u_2 \in AbsUse$. Given that there are these many choices, *which one is better?* Is there any type that 'subsumes' the other? The answer is *no!* Neither of these types can be termed principal, since they cannot be compared. A principal type is one such that is more general than any other type that can be proven for the expression.

In order to obtain a principal type property, variables need to be introduced to represent undetermined uses. Intuitively, if we let ν to be a variable ranging on the domain of Abstract Uses, a type that more precisely describes the behavior of g is

$$(\alpha \xrightarrow[\nu_2]{\nu_1} \beta) \xrightarrow{cs} \alpha \xrightarrow[\nu_2]{\nu_1} \beta$$

This specification can be instantiated by substituting the use variable to any of the types given above.

Although the need for use variables seems reasonable, at least as an analogy to type variables, principal first order types (the types of functions whose arguments are not functions) do not contain type variables, so a first-order language would not need them. It is no wonder why this working example contains a higher-order function.

One complication use variables add to the language of types, and to the inferencing process as a whole is how to solve operations on uses when either operator contains a variable. The value of $\nu \cdot cs$ corresponds to ν in the specific domain of abstract uses presented in this dissertation, but $\nu \overset{par}{\odot} cm$ cannot be computed. Such unresolved operations would need to be left specified in the type. Upon instantiation of ν—to rs, for example—the operation would be solved—to cm.

The use variables were deliberately left out of the initial description of extended types because of these non-trivial complications they add to the language of types.

Now consider how the problem that arises when use variables are involved in a fixpoint computation. We could use the function *mapa*! defined above, but a simpler

function, *many* is considered:

$$\begin{aligned} many \equiv\ & \lambda f\ (fix\ \lambda m.\lambda n.\lambda a. \\ & \quad if\ (n = 0) \\ & \quad a \\ & \quad (m\ (n-1)\ (f\ a))) \end{aligned}$$

This function applies the function f n times to a. Figure 4.9 highlights some of the derivation steps to compute the type

$$(\alpha \xrightarrow{cs} \alpha) \xrightarrow{cm} Int \xrightarrow{rs} \alpha \xrightarrow{cs} \alpha$$

However, by the above discussion, it should be clear that the assumption that f be of type $\alpha \xrightarrow{cs} \alpha$ is over-restrictive, and that a type for *many* should exist if f's type was assumed to be just $\alpha \xrightarrow{\nu} \alpha$.[2] Such a type must satisfy the condition imposed by the fixpoint—the type reconstructed for $(\lambda a.\lambda n.\ \ldots)$ must have the *same* as the type assumed for m.

If we assume the types of f and m to be

$$\begin{aligned} f{:}\alpha &\xrightarrow{\nu_1} \alpha \\ m{:}Int &\xrightarrow{\nu_3} \alpha \xrightarrow{\nu_5} \alpha \end{aligned}$$

then the type inferred for $(\lambda a.\lambda n.\ \ldots)$ would be

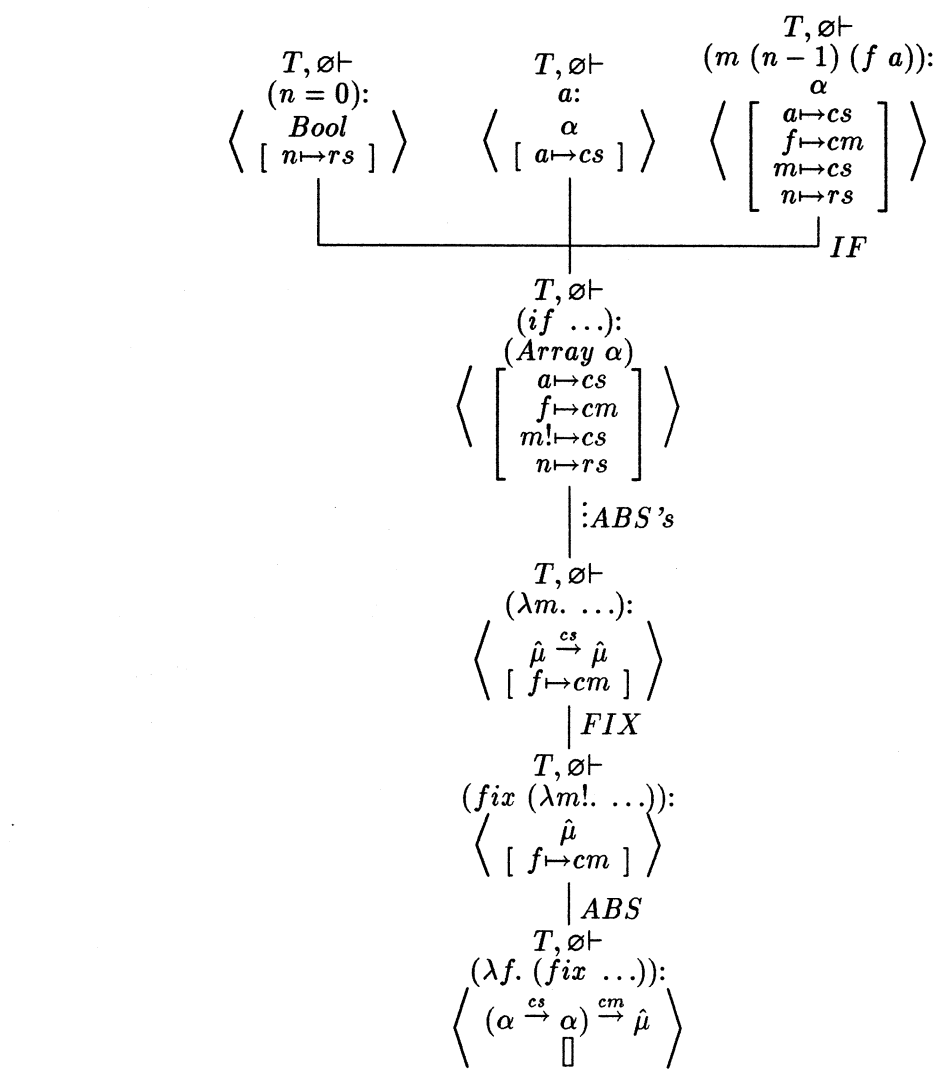
$$Int \xrightarrow{cs \overset{alt}{\odot} \nu_3 \cdot \nu_1} \alpha \xrightarrow{\nu_5 \cdot rs} \alpha$$

The only way to guarantee that the two last types be the same is by solving the set of equations

$$\begin{aligned} \nu_3 &= \nu_3 \cdot rs \\ \nu_5 &= cs \overset{alt}{\odot} \nu_5 \cdot \nu_1 \end{aligned}$$

The equation for ν_3 can be solved ($\nu_3 \equiv rs$), but the one for ν_5 cannot, with the added complication that it is recursive. How can recursive equations be expressed in a type?

[2]For clarity of exposition, the use of the anonymous object will be ignored.



$$
\begin{array}{rcl}
\hat{\mu} & \equiv & (\alpha \xrightarrow{cs} \alpha) \xrightarrow{cm} Int \xrightarrow{rs} \alpha \xrightarrow{cs} \alpha \\
T & \equiv & \left[\begin{array}{l} a \mapsto (Array\ \alpha) \\ f \mapsto \alpha \xrightarrow{cs} \alpha \\ m \mapsto \hat{\mu} \\ n \mapsto Int \end{array}\right] \\
many & \equiv & \lambda f\ (fix\ \lambda m.\lambda n.\lambda a. \\
 & & \quad if\ (n = 0) \\
 & & \quad a \\
 & & \quad (m\ (n-1)\ (f\ a)))
\end{array}
$$

Figure 4.9: Extended Type Derivation of *many*

Just as any recursive equation. As can be seen from this example, there can be more than one recursive equation. The type of *many*, under the assumpions would be

$$(\alpha \xrightarrow{\nu_1} \alpha) \xrightarrow{cm} Int \xrightarrow{rs} \alpha \xrightarrow{\nu_5} \alpha$$
$$where\ \nu_5 = cs \overset{alt}{\odot} \nu_5 \cdot \nu_1$$

Types are operated upon restrictions on their structure. In the rule for application, it can be seen that if e_1 is of type $\hat{\mu}_1 \xrightarrow[u_2]{u_1} \hat{\mu}_2$ and e_2 is of type $\hat{\mu}_1$ (exactly the type of the argument), then the rule can be applied (the type of the application is $\hat{\mu}_2$). As a result, inference on types can proceed whenever there is enough information on the structure of types without requiring that types be completely instantiated (i.e., without type variables).

Unlike types, the domain of abstract uses has no structure. Uses are operated exclusively by the binary operations presented in Chapter 3. These operations act on uses and, in general, cannot be performed if any of its arguments is unkown—the value of the operation $\nu \overset{par}{\odot} cs$ can range from cs to wm, depending on the actual use ν represents. So in order for the system to properly handle use variables, it must also handle operations on use variables, and fixpoint equations on them to deal with recursive constraints.

Although the types computed by this system are simple, the existence of expressions without principal type suggests that the system is not powerful enough to provide concise specifications on functions.

The existence of recursive equations raise serious issues on the soundness of the type model (what is precisely the meaning of a fixpoint equation? Are the type rules complete?), as well as computational issues (Can these fixpoints be solved effectively?), and pragmatic issues (Is such a type system too complex for practical use?). These issues will be discussed in the remainder of the dissertation. In the remainder of this chapter, I revise the definition of the language of extended types, and prove the soundness of the inference rules.

4.6 Extended Type Expressions (Revised)

The revised syntax of abstract use expressions and extended type expressions is given in Figure 4.10. For ease of presentation, the names of all syntactic structures are preserved, except the sub language abstract uses ($AbsUse$), which now is $AbsUse_{Exp}$ to indicate the fact that use expressions are also permitted. Note the introduction of the infrastructure on uses to deal with use variables, as well as the set of recursive equations on uses.

Conventions

- The syntax $\{\nu_1, \ldots, \nu_n\}\tau$ is used to stress the fact that there are free occurrences of use variables $\nu_1, \ldots, \nu_n$ in τ. Further, by using this notation, it is also implied that only those use variables are free in τ
- The notation $\forall\nu_1, \ldots, \nu_n.\langle\tau, C\rangle$ is shorthand for the type expression $\forall\nu_1. \ldots \forall\nu_n.\tau$.
- The group of fixpoint equations will be collectively denoted by P, as in μ *where* P. Further P will be manipulated like a set: if

$$P_1 \equiv \langle\nu_{11}, \ldots, \nu_{1m}\rangle = \langle u_{11}, \ldots, u_{1m}\rangle$$

and

$$P_2 \equiv \langle\nu_{21}, \ldots, \nu_{2n}\rangle = \langle u_{21}, \ldots, u_{2n}\rangle$$

then

$$P_1 \cup P_2 \equiv \langle\nu_{11}, \ldots, \nu_{1m}, \nu_{21}, \ldots, \nu_{2n}\rangle = \langle u_{11}, \ldots, u_{1m}, u_{21}, \ldots, u_{2n}\rangle$$

The type

$$\{\nu_1, .., \nu_n\}\hat{\mu}$$

denotes all types

$$\{\}(\hat{\mu}[u_1/\nu_1, .., u_n/\nu_n])$$

$u \in AbsUse_{Exp}$	$::=$	$\perp \mid rs \mid rm \mid cs \mid cm \mid ws \mid ws \vee cm \mid wm$	*Abstract uses*
		$\mid u_1 \overset{alt}{\odot} u_2 \mid u_1 \overset{seq}{\odot} u_2 \mid u_1 \overset{par}{\odot} u_2$	*Binary operators*
		$\mid u_1 \cdot u_2$	*Projections*
		$\mid \nu$	*Use Variables*
$\hat{\sigma} \in TypeSch_{Ext}$	$::=$	$\langle \hat{\mu}, C \rangle$ *where* P	
		$\mid \forall \alpha \hat{\sigma}$	
		$\mid \forall \nu \hat{\sigma}$	
$\hat{\mu} \in MutType_{Ext}$	$::=$	$\hat{\tau}$	
		$\mid$ *Array* $\hat{\tau}$	*Arrays*
$\hat{\tau} \in Type_{Ext}$	$::=$	*Int* $\mid$ *Bool* $\mid \ldots$	*Basic Types*
		$\mid$ *Pair* $\hat{\tau}_1$ $\hat{\tau}_2$	*Pairs*
		$\mid \hat{\mu} \overset{u_1}{\underset{u_2}{\rightarrow}} \hat{\mu}$	*Functions*
		$\mid \alpha$	*Type Variables*
P	$::=$	$\langle \nu_1, \ldots, \nu_n \rangle = \langle u_1, \ldots, u_n \rangle$	*Use Recursive Equations*
$c \in Coercion$	$::=$	$u_1 \trianglerighteq u_2$	*Coercion on uses*
		$\mid \hat{\mu}_1 \trianglerighteq \hat{\mu}_2$	*Coercion on types*
$C \in CSet$	$::=$	$\{c_1, , \ldots, c_n\}$	*Coercion Sets*
$\nu \in Id_{Use}$			*Use Variables*
$\alpha \in Id_{Type}$			*Type Variables*

Figure 4.10: Abstract Use and Extended Type Expressions (Revised)

$$
\begin{array}{lll}
\tilde{\sigma} \in \widetilde{TypeSch}_{Ext} ::= & \langle \tilde{\tau}, C \rangle \ where \ P & \\
& \mid \forall \alpha \tilde{\sigma} & \\
& \mid \forall \nu \tilde{\sigma} & \\
\\
\tilde{\mu} \in \widetilde{MutType}_{Ext} ::= & \tilde{\tau} & \\
& \mid Array \ \tilde{\tau} & Arrays \\
\\
\tilde{\tau} \in \widetilde{Type}_{Ext} \subseteq Type_{Ext} ::= & Int \mid Bool \mid \ldots & Basic \ Types \\
& \mid Pair \ \tilde{\tau}_1 \ \tilde{\tau}_2 & Pairs \\
& \mid \tilde{\alpha} & Type \ Variables
\end{array}
$$

Figure 4.11: Extended Type Expressions not Containing Functions (Revised)

where $u_i \in AbsUse$ for $i = 1 \ldots n$. Also, the type

$$\{\nu_1, .., \nu_n\}(\mu \ where \ \langle \nu_{n+1}, \ldots \nu_m \rangle = \langle u_{n+1}, \ldots, u_m \rangle)$$

means the type

$$S \ \hat{\mu}$$

where

$$S = [u'_1/\nu_1, \ldots u'_n/\nu_n, u'_{n+1}/\nu_{n+1}, \ldots u'_m/\nu_m]$$

and

$$(\mathcal{E}_\nu \ S\nu_i) \equiv (\mathcal{E}_\nu \ Su_i)$$

As before, I make the distinction of the sublanguage of types which does not involve functions, defined in Figure 4.11. Note that $\hat{\sigma}$ ranges over $TypeSch_{Ext}$, whereas $\tilde{\sigma}$ ranges over $\widetilde{TypeSch}_{Ext}$; i.e., the former can be instantiated to any type, while the latter can only be instantiated to types that do not contain functions.

Note that the language is also shallow on use variables—quantification of these can only appear affecting the whole type expression. However, use variables are very different in nature from type variables, and are accordingly manipulated in a different way.

4.7 Extended Type Instantiation/Coercion Rules (Revised)

Since the revised language allows for use variables, the instantiation and coercion rules must be expanded acordingly. Type specialization occurs by the instantiation of either a type variable, or a use variable. The type of the substitution function now changes to reflect the fact that use variables can be substituted. The definition of extended substitution is shown in Figure 4.12, and has type

$$\begin{aligned} \_[\_/\_]: \quad & (Type_{Ext} \rightarrow Type_{Ext} \rightarrow Id_{Ext} \rightarrow Type_{Ext}) \\ & +(Type_{Ext} \rightarrow AbsUse \rightarrow Id_{Use} \rightarrow Type_{Ext}) \\ & +(AbsUse \rightarrow Type_{Ext} \rightarrow Id_{Ext} \rightarrow AbsUse) \\ & +(AbsUse \rightarrow AbsUse \rightarrow Id_{Use} \rightarrow AbsUse) \end{aligned}$$

As before, the abstraction of the first operand from a substitution operation is refered to by S.

Coercion Rules do not change significantly from the rules introduced in Section 4.3. Only additional rules are added, each of them as a direct consequence of a revised feature. Only the additional rules are listed.

1' Instantiation

$$\frac{Su_1 = u_2}{C \Vdash \{u_1 \trianglerighteq u_2\}}$$

If u_2 is an instance of u_1 then u_1 can be coerced to u_2

$$
\begin{array}{rcll}
\nu[u/\nu] & = & u & \\
\nu_1[u/\nu_2] & = & \nu_1 & \text{if } \nu_1 \neq \nu_2 \\
\nu[\hat{\tau}/\alpha] & = & \nu & \\
\alpha[\hat{\tau}/\alpha] & = & \hat{\tau} & \\
\alpha_1[\hat{\tau}/\alpha_2] & = & \alpha_1 & \text{if } \alpha_1 \neq \alpha_2 \\
\alpha[u/\nu] & = & \alpha & \\
(\forall\nu\hat{\chi})[u/\nu] & = & \forall\nu\hat{\chi} & \\
(\forall\nu_1\hat{\chi})[\hat{\tau}/\nu_2] & = & \forall\nu_1\hat{\chi}[\hat{\tau}/\nu_2] & \text{if } \nu_1 \neq \nu_2 \\
(\forall\nu\hat{\chi})[u/\nu] & = & \forall\nu\hat{\chi} & \\
(\forall\alpha\hat{\sigma})[\hat{\tau}/\alpha] & = & \forall\alpha\hat{\sigma} & \\
(\forall\alpha_1\hat{\sigma})[\hat{\tau}/\alpha_2] & = & \forall\alpha_1\hat{\sigma}[\hat{\tau}/\alpha_2] & \text{if } \alpha_1 \neq \alpha_2 \\
(\forall\alpha\hat{\sigma})[u/\nu] & = & \forall\alpha\hat{\sigma} &
\end{array}
$$

$$
\begin{array}{rcll}
S\ \tau & = & \tau & \text{if } \tau \text{ is a basic type} \\
S\ (Array\ \hat{\tau}) & = & (Array\ (S\ \hat{\tau})) & \\
S\ (Pair\ \hat{\tau}_1\ \hat{\tau}_2) & = & (Pair\ (S\ \hat{\tau}_1)\ (S\ \hat{\tau}_2)) & \\
S\ (\hat{\chi}\ \overset{u1}{\underset{u_2}{\rightarrow}}\ \hat{\tau}) & = & (S\ \hat{\chi})\ \overset{(S\ u1)}{\underset{(S\ u2)}{\rightarrow}}\ (S\ \hat{\tau}) & \\
S\ \langle\hat{\tau}, C\rangle & = & \langle(S\ \hat{\tau}), (S\ C)\rangle & \\
S\ \{c_1, \ldots, c_n\} & = & \{(S\ c_1), \ldots, (S\ c_n)\} & \\
S\ (u_1 \trianglerighteq u_2) & = & (S\ u_1) \trianglerighteq (S\ u_2) & \\
S\ (\hat{\tau}_1 \trianglerighteq \hat{\tau}_2) & = & (S\ \hat{\tau}_1) \trianglerighteq (S\ \hat{\tau}_2) & \\
S\ (u_1 \odot u2) & = & (S\ u_1) \odot (S\ u_2) & \\
S\ (u_1 \cdot u2) & = & (S\ u_1) \cdot (S\ u_2) &
\end{array}
$$

Figure 4.12: Substitution on Extended Types

8' Type Schemes

$$\frac{C \Vdash \{\hat{\sigma}_1 \trianglerighteq \hat{\sigma}_2\}}{C \Vdash \{\forall \nu.\hat{\sigma}_1 \trianglerighteq \forall \nu.\hat{\sigma}_2\}}$$

If two types are coercible, then quantifying them by the *same* variable retains the property.

9' Fixpoint Types

$$\frac{C \Vdash \{\hat{\mu}_1 \trianglerighteq \hat{\mu}_2\}}{C \Vdash \{(\mu_1 \; where \; P) \trianglerighteq (\mu_2 \; where \; P)\}}$$

If two types are coercible, then restricting them by fixpoint does not alter their relation.

4.8 Type/Liability Inference Rules (Revised)

The inference rules do not vary significantly from the ones presented in 4.4. The only differences are that the type of each subexpression is allowed to have a set of fixpoint restrictions, which are unioned to produce the set of restrictions of the result.

1. Constants

$$C, T \vdash k : \mathcal{K}(k)$$

2. Identifiers

$$C, T \vdash v : \langle T(v), \bot_{Liab}[v \mapsto cs] \rangle$$

3. Lambda Abstractions

$$\frac{C, T[v \mapsto \hat{\mu}_1 \; where \; P_1] \vdash e : \langle \hat{\mu}_2 \; where \; P_2, L[v \mapsto u_1, \xi \mapsto u_2] \rangle}{C, T \vdash (\lambda v.e) : \langle \hat{\mu}_1 \xrightarrow[u_2]{u_1} \hat{\mu}_2 \; where \; P_1 \cup P_2, L^{v\xi} \rangle}$$

If there is any occurrence of v inside e, then $P_1 \subseteq P_2$. However, P_2 is not guaranteed to contain P_1 if threre is no occurrence of v in e. Since $\hat{\mu}_1$ becomes

part of the type being concluded by the rule, all fixpoint restrictions P_1 must appear in the qualifying the type. Thus the explicit union.

4. Applications

$$\frac{\begin{array}{c} C,T \vdash e_1 : \langle \hat{\mu}_1 \xrightarrow[u_2]{u_1} \hat{\mu}_2 \ where\ P_1, L_1 \rangle \\ C,T \vdash e_2 : \langle \hat{\mu}_1 \ where\ P_2, L_2 \rangle \end{array}}{C,T \vdash (e_1\ e_2) : \langle \hat{\mu}_2 \ where\ P_1 \cup P_2, (L_1 \overset{par}{\odot} (u_1 \cdot L_2)) \sqcup \bot_{Liab}[\xi \mapsto u_2] \rangle}$$

5. Strict Application

$$\frac{\begin{array}{c} C,T \vdash e_1 : \langle \tilde{\mu}_1 \xrightarrow[u_2]{u_1} \hat{\mu}_2 \ where\ P_1, L_1 \rangle \\ C,T \vdash e_2 : \langle \tilde{\mu}_1 \ where\ P_2, L_2 \rangle \end{array}}{C,T \vdash \langle e_1\ e_2 \rangle : \langle \hat{\mu}_2 \ where\ P_1 \cup P_2, ((u_1 \cdot L_2) \overset{seq}{\odot} L_1) \sqcup \bot_{Liab}[\xi \mapsto u_2] \rangle}$$

6. If-then-else

$$\frac{\begin{array}{c} C,T \vdash e_p : \langle Bool \ where\ P_p, L_p \rangle \\ C,T \vdash e_c : \langle \hat{\mu} \ where\ P_c, L_c \rangle \\ C,T \vdash e_a : \langle \hat{\mu} \ where\ P_a, L_a \rangle \end{array}}{C,T \vdash (if\ e_p\ e_c\ e_a) : \langle \hat{\mu} \ where\ P_p \cup P_c \cup P_a, (L_p \overset{seq}{\odot} (L_c \overset{alt}{\odot} L_a)) \rangle}$$

Note that the fixpoint restrictions are collected even in the case that the type of one of the premises is discarded, therefore P_p participates in the resulting restrictions.

7. Let

$$\frac{\begin{array}{c} C,T \vdash e_1 : \langle \hat{\sigma}, L_1 \rangle \\ C,T[v \mapsto \hat{\sigma}] \vdash e_2 : \langle \hat{\mu} \ where\ P, L_2[v \mapsto u_1, \xi \mapsto u_2] \rangle \end{array}}{C,T \vdash (let\ v = e_1\ in\ e_2) : \langle \hat{\mu} \ where\ P, (L_2 \overset{par}{\odot} (u_1 \cdot L_1)) \sqcup \bot_{Liab}[\xi \mapsto u_2] \rangle}$$

8. Let*

$$\frac{\begin{array}{c} C,T \vdash e_1 : \langle \tilde{\sigma}, L_1 \rangle \\ C,T[v \mapsto \tilde{\sigma}] \vdash e_2 : \langle \hat{\mu} \ where\ P, L_2[v \mapsto u_1, \xi \mapsto u_2] \rangle \end{array}}{C,T \vdash (let* \ v = e_1\ in\ e_2) : \langle \hat{\mu} \ where\ P, ((u_1 \cdot L_1) \overset{seq}{\odot} L_2) \sqcup \bot_{Liab}[\xi \mapsto u_2] \rangle}$$

9. Fix

$$\frac{C,T \vdash e : \langle \hat{\mu} \overset{u_1}{\underset{u_2}{\rightarrow}} \hat{\mu} \ where \ P, L\rangle}{C,T \vdash (fix\ e) : \langle \hat{\mu} \ where \ P, (u_1 \cdot L) \overset{alt}{\odot} \perp_{Liab}[\xi \mapsto u_2]\rangle}$$

Interestingly enough, the rule for fixpoint expressions is not affected much by the language revision. However, it is precisely this revision that allows P to be a non-trivial fixpoint restriction necessary to satisfy the recursive constraints on the type.

10. Coercion

$$\frac{C,T \vdash e : \langle \hat{\sigma}, L\rangle \qquad C \Vdash \{\hat{\sigma} \trianglerighteq \hat{\sigma}'\}}{C,T \vdash e : \langle \hat{\sigma}', L\rangle}$$

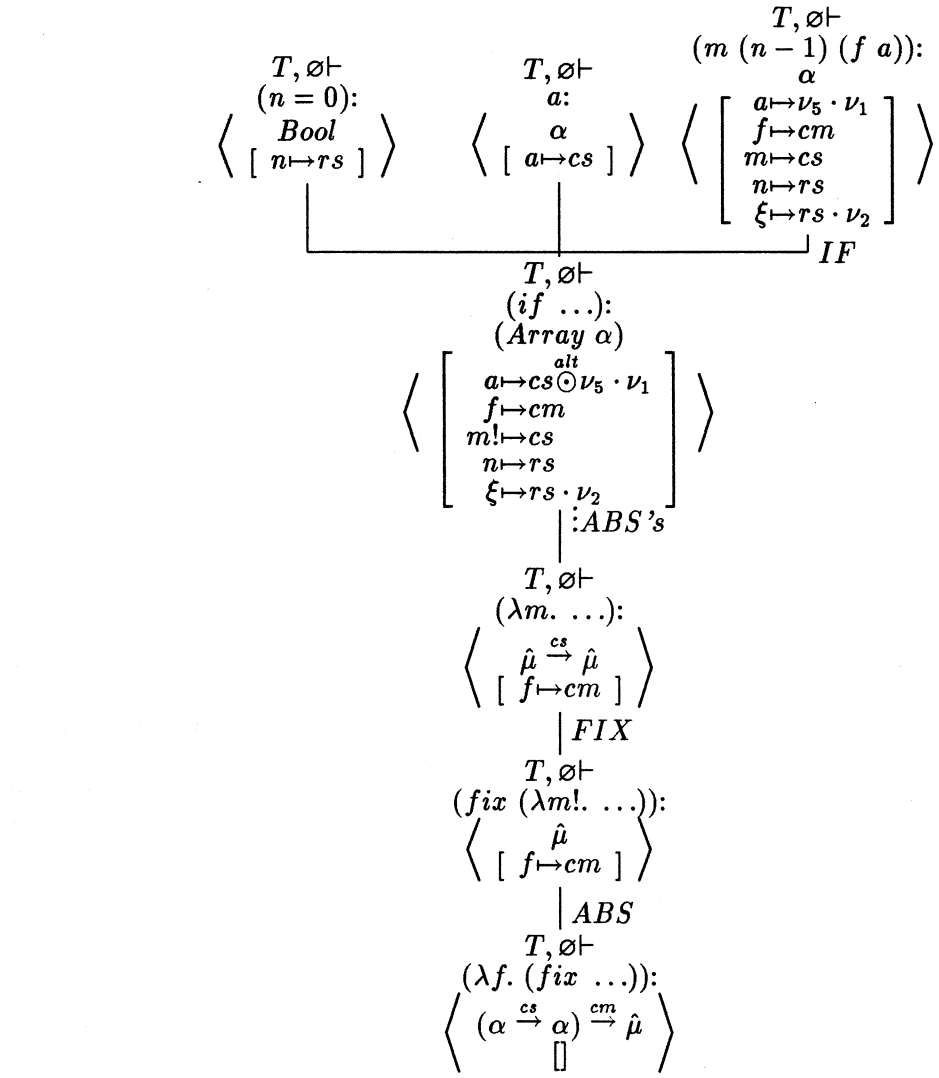
Figure 4.13 shows an extended type derivation for *many*. This derivation does not assume as known the uses of the bound variable and the anonymous object., e.g. f is assumed to be of type

$$f\alpha \overset{\nu_1}{\underset{\nu_2}{\rightarrow}} \alpha$$

4.9 Soundness of the Inference Rules

In this section a proof is given that guarantees that the extended type system for λ_{st}-calculus is *sound*—the rules only infer valid extended types. This implies that the inferred abstract uses are always conservative with respect to the way the graph is actually used in reduction to normal form.

In Chapter 3, the concept of abstract liability as a mapping from free variables to abstract uses was introduced. The purpose of that environment was to account for how the expressions manipulated their free variables upon evaluation.



$$
\begin{array}{rcl}
\hat{\mu} & \equiv & (\alpha \xrightarrow[\nu_2]{\nu_1} \alpha) \xrightarrow{cm} Int \xrightarrow{rs} \alpha \xrightarrow[\nu_2]{\nu_5} \alpha \\
 & & \textit{where}\ \ \nu_5 = cs \overset{alt}{\odot} \nu_5 \cdot \nu_1 \\
 & & \qquad\ \ \nu_6 = rs \cdot \nu_2 \\
T & \equiv & \left[\begin{array}{l} a \mapsto (Array\ \alpha) \\ f \mapsto \alpha \xrightarrow[\nu_2]{\nu_1} \alpha \\ m \mapsto \hat{\mu} \\ n \mapsto Int \end{array} \right] \\
many & \equiv & \lambda f\ (fix\ \lambda m.\lambda n.\lambda a. \\
 & & \qquad if\ (n = 0) \\
 & & \qquad a \\
 & & \qquad (m\ (n-1)\ (f\ a)))
\end{array}
$$

Figure 4.13: Extended Type Derivation of *many* Using Fixpoints

Intuitively, free variables represent undetermined graphs in the calculus. As such, free variables may be replaced by *any* λ_{st}-graph provided that all occurrences of the same variable *share* the same graph. Further, in λ_{st}, the type of a variable must coerce to the type of the graph that is being substituted for the variable.

It is then natural to relate the abstract liability of a graph to how any actual graph that would replace the free variables would be manipulated when the expression is fully reduced. To that end, the corresponding concepts of concrete uses and liabilities are introduced and the relation between the abstract and concrete domains are established.

For a given graph, *concrete uses* are properties satisfied by subgraphs when the graph is reduced to normal form. In contrast, *abstract uses* are properties of free variables in subgraphs—an abstract use of a variable is a conservative approximation of the concrete use satisfied by any graph to which that variable may be instantiated when the graph is reduced to normal form. Note that $g : \langle \hat{\tau}, L \rangle$, indicates that the type of g is $\hat{\tau}$, and that there can be a context where the reduction of g to normal form produces the effects specified by L. *In the case that $\hat{\tau}$ is a functional type, it may be possible that portions of the liability become effective only when the function is applied.*

Definition 4.9.1 *Two graphs g_1, and g_2 are* disjoint *if they do not share any substructure, i.e., when $lab(g_1) \cap lab(g_2) = \varnothing$.*

Definition 4.9.2 *Given a graph g with subgraphs ${}^{\ell_1}g_1, \ldots, {}^{\ell_n}g_n$, where g_i and g_j are disjoint if $i \neq j$. Then* concrete liability *of g with respect to $\{g_i\}$ on reduction path $p = \Delta_1 \ldots \Delta_m$ is an environment $M_{p,\{\ell_1,\ldots,\ell_n\}}$ associating the labels $\ell_1, \ldots, \ell_n$ with the concrete use of the graph g_i when the reduction path p is performed.*

The following lemma just states that β-reductions do not affect the concrete liability of expressions.

Lemma 4.9.3 *The concrete liability of* $g_1[g_2/x]$ *is the same as the concrete liability of* $((\lambda x.g_1)\ g_2)$.

Proof For every reduction path

$$g_1'[g_2/x] \xrightarrow{\Delta^*} \tilde{g}_2$$

there is a corresponding reduction path

$$(\lambda x.g_1)\ g_2 \rightarrow g_1'[g_2/x] \xrightarrow{\Delta^*} \tilde{g}_2$$

where g' is a copy of g using Wadsworth copy operation. For every reduction path

$$(\lambda x.g_1)\ g_2 \xrightarrow{\Delta_1^*} (\lambda x.g_{11})\ g_{21} \rightarrow g_{11}[g_{21}/x] \xrightarrow{\Delta_2^*} g_3$$

there is the corresponding

$$g_1[g_2/x] \xrightarrow{\Delta_1^*} \rightarrow g_{11}[g_{21}/x] \xrightarrow{\Delta_2^*} g_3$$

where g_{11}, and g_{21} are residuals of g_1, and g_2.

The extra β-reduction does not contribute at all to the *threading* and *mutability* properties, since these only care about the relative orders of mutators and observers. With respect to the *capturing* property, both graphs have the same normal form, hence they capture the same objects. □

Theorem 4.9.4 Soundness of Extended Type Inference Rules. *Given a graph* g *with free variables* $v_1, \ldots, v_n$, *type* μ, *and abstract liability* $\perp_{Liab}[\xi \mapsto u_0, v_1 \mapsto u_1, \ldots, v_n \mapsto u_n]$. *Then for all graphs* $g_1, \ldots, g_n$ *(not necessarily disjoint) labeled* $\ell_1, \ldots, \ell_n$, *the concrete liability of* $g[v_1 \mapsto g_1, \ldots, v_n \mapsto g_n]$ *with respect to* $\ell_1, \ldots, \ell_n$, $\perp_{Liab}[\ell_1 \mapsto u_1', \ldots, \ell_n \mapsto u_n']$ *will be such that for all* i *such that* $1 \leq i \leq n$, $u_i' \sqsubseteq u_i$.

Proof By induction on the structure of g:

1. Constants

   $g = c$

   $C, T \vdash k : \mathcal{K}(k)$

   The theorem trivially holds for g, since it has no variables to be substituted for.

2. Variables

   $g = v$

   $C, T \vdash v : \langle T(v), \bot_{Liab}[v \mapsto cs] \rangle$

   For all graphs g, the reduction of $v[{}^{\ell}g/v]$ *is* the reduction of ${}^{\ell}g$. The actual use of $v[{}^{\ell}g/v]$ is either cs, if part of the graph is captured within the value of $v[{}^{\ell}g/v]$, or rs, otherwise. Note that parts of g may be either multiple-threaded, or mutated, but that does not affect the use of ℓ. Thus, the theorem holds for g; i.e, for all ${}^{\ell}g$, and p, $M_{p,\{\ell\}}(v[{}^{\ell}g/v]) = \bot_{Liab}[\ell \mapsto u']$, and $u' \sqsubseteq u$.

3. Abstractions

   $g = (\lambda v.g')$

   $C, T[v \mapsto \hat{\mu}_a] \vdash g' : \langle \hat{\hat{\mu}}_b, \bot_{Liab}[v_1 \mapsto u_1, \ldots, v_n \mapsto u_n, v \mapsto u_a, \xi \mapsto u_b] \rangle$

   Note that v is the only free variable of g' which is not free in g. Therefore g' must actually be of the form $\{v_1, \ldots, v_n, v\}g'$. By the inductive hypothesis, the theorem holds for g'. Therefore, for any path p

   $$M_{p,\{\ell_1,\ldots\ell_n\}}(g'[{}^{\ell_1}g_1/v_1, \ldots, {}^{\ell_n}g_n/v_n]) = \bot_{Liab}[\ell_1 \mapsto u'_1, \ldots, \ell_n \mapsto u'_n]$$

   such that $u'_1 \sqsubseteq u_1, \ldots, u'_n \sqsubseteq u_n$. By Lemma 4.9.3, the concrete liability of g' with respect to $\ell_1, \ldots, \ell_n$ is the same as that of $(\lambda v.g')$. The inferred abstract liability for $(\lambda v.g')$ is $\bot_{Liab}[v_1 \mapsto u_1, \ldots, v_n \mapsto u_n]$. Therefore the theorem holds for g.

4. Applications

   $g = (g_a\ g_b)$

   $C, T \vdash g_a : \langle \hat{\mu}_a \xrightarrow[u_b]{u_a} \hat{\mu}_b, L_a \rangle$

   $C, T \vdash g_b : \langle \hat{\mu}_a, L_b \rangle$

Reductions in g_a happen independently from reductions in g_b. As a consequence, the only safe way to combine liabilities in g_a to liabilities in g_b is by using the $\overset{par}{\odot}$ operator. If g_b is a first-order graph, then g_b will not be duplicated, but if it of higher-order type, then parts of it may be duplicated each time it is applied. A conservative assumption is to estimate that each use of g_b within the graph g_a would result in g_b being duplicated. This is precisely what $u_a \cdot L_b$ does: if u_a is muliple-threaded, then any single-threaded use in L_b is transformed to the corresponding multiple-threaded use in $u_a \cdot L_b$. In addition, if u_a implies a mutation, then all uses in L_b that mean capturing will be transformed as mutated in $u_a \cdot L_b$.

5. Strict Applications

   $g = \langle g_a \; g_b \rangle$

   $C, T \vdash g_a : \langle \tilde{\mu}_a \overset{u_a}{\underset{u_b}{\rightarrow}} \hat{\mu}_b, L_a \rangle$

   $C, T \vdash g_b : \langle \tilde{\mu}_a, L_b \rangle$

   Similar to the previous case, except that τ_b is a first-order type, and as such, it is guaranteed that it can be *completely* reduced to normal form *before* evaluation on the application begins. Therefore, the operator $\overset{seq}{\odot}$ can safely be used instead of $\overset{par}{\odot}$ in the inference rule.

6. If-then-else

   $g = (if \; g_a \; g_b \; g_c)$

   $C, T \vdash g_a : \langle Bool, L_a \rangle$

   $C, T \vdash g_b : \langle \hat{\mu}, L_b \rangle$

   $C, T \vdash g_c : \langle \hat{\mu}, L_c \rangle$

   Note that no object within g_b or g_c can be reduced until the predicate node g_a is reduced. There is no case where both are reduced, because after if can be reduced, one of the subgraphs will become inaccessible. The liability assumed is thus conservative since it assumes one of them will be reduced, *after* the graph g_a is reduced.

7. Let

   $g = (let\ v = g_a\ in\ g_b)\ C, T\ \vdash g_a : \langle \hat{\sigma}, L_a \rangle$

   $C, T[v \mapsto \sigma]\ \vdash g_b : \langle \hat{\mu}, L_b \rangle$

   Note that this construction is equivalent (in the untyped λ_{st}-calculus) to

   $$((\lambda x.g_b)\ g_a)$$

   As such, their concrete liabilities are the same. With respect to the type aspect, the difference between the *let*-construct and its translation is that x type is allowed to generalize to σ in the in the construct, but not in its translation. $(\lambda x.g_b)$ would not possess a shallow type if x was assumed a type scheme. However, the *let*-construct can still be inferred a shallow type in that case.

8. Let$*$

   $g = (let* \ v = g_a\ in\ g_b)\ C, T\ \vdash g_a : \langle \hat{\sigma}, L_a \rangle$

   $C, T[v \mapsto \sigma]\ \vdash g_b : \langle \hat{\mu}, L_b \rangle$

   Similar to the previous case, but using a strict application instead of a standard application.

9. Fixpoint

   $g = (fix\ g_a)$

   $C, T\ \vdash g_a : \langle \mu \xrightarrow[u_b]{u_a} \mu, L \rangle$

   This is just a special case of applications with the function being the fixpoint operator. The liability of the graph resulting from recursively splicing g in all instances of the bound variable in g_a amounts to projecting u_a to L, i.e., $u \cdot L$.

10. Coercion

    $C, T\ \vdash e : \langle \hat{\mu}, L \rangle$

    $C\ \Vdash\ \{\hat{\mu} \trianglerighteq \hat{\mu}'\}$

    Immediate.

□

This theorem states the soundness of the inference rules: any extended type that can be inferred under the rules makes conservative assumptions on the way objects are used. There are two places where the abstract liability is more conservative than the concrete liability: function applications, and conditionals. In the conditionals, information is lost on which arm of the conditional provides the liability. Data-dependency information is lost in abstractions. Applications, unable to reconstruct the lost data dependencies, must infer the most conservative liability—one that can be satisfied no matter what the data dependency between the formal argument and its environment within the function.

4.10 Conclusions

The inferred liabilities will be conservative (i.e., the actual use of any object will be weaker than its abstract use) only if a *thread-preserving* reduction path is followed. This was explained in detail in Section 4.9.

Only named objects and anonymous objects are accounted for in the liability analysis. No consideration is made on substructures. This makes the simplification that a function abstraction manipulates the use of *one* object—its bound variable—regardless of the bound variable's structure. As a result, composite data structures are associated with *one* consolidated abstract use, even though different parts of the structure are manipulated in a radically different way. A more elaborate extension to the type system would allow to associate independent uses with different substructures. Although this seems desirable, the type system becomes much more complicated. A first-order type system where the capturing aspect of objects is isolated is explored in [Guzmán and Hudak, 1991].

Additionally, a simpler structure of extended types is possible by the way the operations on uses and liabilities are expressed on the inference rules. This will translate into a simpler type reconstruction algorithm which will be presented in Chapter 6. Not only type reconstruction is proven feasible for this restricted language, but ex-

tensions to the language are nonsensical in absence of any behavioral information provided by the user.

Chapter 5

Examples

5.1 Introduction

In this section, two examples are presented: quicksort and gaussian elimination. They were chosen to demonstrate several different issues when dealing with mutation of state.

In an effort to improve readability, a Haskell-like syntax is adopted:

- Equation groups are used for function definitions in preference to a sequence of lambda abstractions. A function may be defined by more than one equation.

- Pattern-matching operations are used. These include guards on the left-hand-side of an equation. The first equation (top to bottom) that results in a successful match is selected for evaluation. The semantics of pattern matching especifies that the guards must be fully evaluated *before* the evaluation of the right-hand-side can take place.

- Constructions *let* and *let**. The first one is interpreted as usual, whereas the second one is interpreted as a strict application. Both constructions allow polymorphism.

- Multiple definitions are allowed in a *let**. These correspond to nested use (top to bottom) of this construction. On the other hand, multiple definitions in a *let* are mutually recursive.

These constructions form the basis of a high-level functional language. However, translation from these more convenient constructs to the lower level constructs provided by the λ_{st}-calculus present no difficulties, and should not affect the semantics of the programs. For techniques on how these source-to-source translations are made, the reader is encouraged to read [Peyton Jones, 1987]. The actual code for these problem in the syntax of the λ_{st}-calculus is provided in Appendix A.

5.2 Quicksort

This example shows how an array of immutable objects (integers) can be manipulated destructively.

Problem

Sort in-place an array of numbers.

Method

By *divide and conquer*. If the array only has zero, or one element, then it is already sorted. Otherwise, select a pivot element from the array. Rearrange the elements of the array so that all elements less than the pivot are contiguous, and so are the elements that are greater than the pivot (*split*!). Recursively sort the smaller elements, and the larger elements (recursive calls to *qs*!). When both sorts are done, then the array is completely sorted.

$$
\begin{aligned}
swap! &:: (Array\ \tau) \overset{ws}{\to} Int \overset{rs}{\to} Int \overset{rs}{\underset{ws}{\to}} (Array\ \tau) \\
split! &:: (Array\ Int) \overset{ws}{\to} Int \overset{rs}{\to} Int \overset{rs}{\to} Int \overset{rs}{\underset{ws}{\to}} ((Array\ Int), Int) \\
qs! &:: (Array\ Int) \overset{ws}{\to} Int \overset{rs}{\to} Int \overset{rs}{\underset{ws}{\to}} (Array\ \tau) \\
quicksort! &:: (Array\ Int) \overset{ws}{\to} (Array\ Int)
\end{aligned}
$$

$$
\begin{aligned}
swap!\ a\ i\ j &= let\!*\ x = (lookup\ a\ i) \\
&\qquad\quad y = (lookup\ a\ j) \\
&\quad in\ update!\ (update!\ a\ i\ y)\ j\ x
\end{aligned}
$$

$$
\begin{aligned}
split!\ a\ i\ j\ p\ \{i == j{+}1\} &= (a, i) \\
split!\ a\ i\ j\ p\ \{a[i] \leq a[p]\} &= split!\ a\ (i{+}1)\ j\ p \\
\{a[i] > a[p]\} &= let\!*\ \ a' = swap!\ a\ i\ j \\
&\quad in\ split!\ a'\ i\ (j{-}1)\ p
\end{aligned}
$$

$$
\begin{aligned}
qs!\ a\ i\ j\ \{i == j{+}1\} &= a \\
qs!\ a\ i\ j\ \{i == j\} &= a \\
qs!\ a\ i\ j &= let\!*\ (a', k) = split!\ a\ i\ j\ i \\
&\qquad\quad k' = copy\ k \\
&\qquad\quad a'' = qs!\ a'\ i\ k' \\
&\quad in\ qs!\ a''\ (k'{+}1)\ j \\
quicksort!\ a &= let\!*\ \ n = size\ a \\
&\quad in\ qs!\ a\ 1\ n
\end{aligned}
$$

Figure 5.1: Implementation of Quicksort In-Place

Program

The program appears in Figure 5.1. It is assumed that arrays always begin on index 1 and cannot be split. Thus, in order to sort sections of the array, the indices for the beginning and ending of the array must be given. Note that in order to make the program typable, *copy* was used within $qs!$ to dissociate k from the pair (a', k). Given that k is of type Int, the copy operation would be trivial in this case, since integers are usually copied in most language implementations anyway. The type system has no knowledge of these pragmatic considerations, however. This copy arises from the fact that (a', k) come from the value of $split!$, and thus are implemented as a single bound variable of quicksort from which a', and k are retrieved. Therefore, a', and k share the same use. Since a' is mutated, it must do it in a single-threaded environment, thus, the value of k must be retrieved *before* a' is updated.

5.3 Gaussian Elimination

It can be seen in this example how arrays can be updated single-threadedly in higher-order contexts. The algorithm presented here is iteratively in nature. Each iteration is performed by a $fold$-like operator named $folda$. The program operates on matrices, i.e., two-dimensional arrays. All elements of the matrix are directly accessible. In this solution, it is assumed the existence of the creator function mka_2, selector function $lookup_2$, and mutator functions $update_2!$ and $swap_2!$ that are two-dimentional versions of $mkarray$, $lookup$, $update!$, and $swap!$—they behave similarly to their one dimensional counterparts, but need two indices to operate.

$$
\begin{aligned}
mkarray_2 &: \; Int \xrightarrow[cs]{rs} Int \xrightarrow{rs} Int \xrightarrow{rs} \alpha \xrightarrow{cs} (Array_2\ \alpha) \\
lookup_2 &: \; (Array_2\ \alpha) \xrightarrow{cs} Int \xrightarrow{rs} Int \xrightarrow{rs} \alpha \\
update_2! &: \; (Array_2\ \alpha) \xrightarrow{ws} Int \xrightarrow{rs} Int \xrightarrow{rs} \alpha \xrightarrow{cs} (Array_2\ \alpha) \\
swap_2! &: \; (Array_2\ \alpha) \xrightarrow{ws} Int \xrightarrow{rs} Int \xrightarrow{rs} Int \xrightarrow{rs} Int \xrightarrow{rs} (Array_2\ \alpha)
\end{aligned}
$$

Problem

Solve the equation system $Ax = b$, where A is a non-singular $n \times n$ matrix, x is a vector of n-variables, and b is a vector of n elements. This system is represented as a matrix of $n \times (n+1)$ elements, with the extra elements holding the vector b.

Method

Gaussian Elimination Method. This is done iteratively (and in imperative style!). There are two passes: forward, and backward. The forward pass has the effect of transforming the matrix into an equivalent upper triangular one. The backward pass transforms the upper triangular matrix into a diagonal matrix, thus solving the system of equations.

Forward Pass. At iteration i, the following invariant holds: *all elements* $A[k,j] = 0$, *and* $A[j,j] = 1$ *for all* $1 \leq k < j < i$. At iteration i (*forward!*), row i is transformed so that $A[i,i]$ becomes 1, and rows $i+1, \ldots, n$ are also transformed so that elements $A[j,i]$, $j > i$ become 0. More specifically, the following takes place at step i:

1. (*findrow!*) Row j is selected, $j \geq i$, such that $A[j,i] \neq 0$, and rows i and j are interchanged (the existence of such a row is guaranteed by the non-singularity assumption on A).

2. (*scalerow!*) Row i is divided by $A[i,i]$, thus making $A[i,i] = 1$, and

3. (*addtorow!*) Row i is appropriately added to the remaining rows in order to transform A[j,i], $j > i$ to 0.

After step n, the transformed matrix is upper triangular.

Backward Pass. At iteration i, it is true that A is upper triangular ($A[k,j] = 0$ for all $1 \leq k < j \leq n$, and $A[j,j] = 1$ for $1 \leq j \leq n$), *and* $A[k,j] = 0$ for all $n \geq k > j > i$.

$$
\begin{aligned}
inc &:: Int \xrightarrow{rs} Int \\
dec &:: Int \xrightarrow{rs} Int \\
folda &:: \tau_1 \xrightarrow{ws} \tau_2 \xrightarrow{rm} \tau_2 \xrightarrow{rs} (\tau_2 \xrightarrow{rs} \tau_2) \xrightarrow{rm} (\tau_2 \xrightarrow{rm} \tau_1 \xrightarrow{ws} \tau_1) \xrightarrow{wm} \tau_1
\end{aligned}
$$

$$
\begin{aligned}
inc\ x &= x+1 \\
dec\ x &= x-1
\end{aligned}
$$

$$
\begin{aligned}
folda\ v\ n\ n\ inc\ f &= v \\
folda\ v\ i\ n\ inc\ f &= folda\ (f\ i\ v)\ (inc\ i)\ n\ inc\ f
\end{aligned}
$$

Figure 5.2: Auxiliary Functions for Gaussian Elimination In-Place

In this iteration, rows $i-1,\ldots,1$ are transformed so that the elements $A[j,i]$, $j<i$ become 0. This is just the application of function *addtorow*! with the counters controlling the loop running backwards (i.e., from i to 0). After step n the matrix $A[i,j]$, $1 \leq i,j \leq n$ is diagonal.

Putting it all together. The function *gauss*! just iterates forward $(1..(n+1))$ the forward pass, and then iterates backward $(n..0)$ the backward pass. The elements $A[i,n+1]$ have the solutions to the equation system.

Auxiliary Functions

The algorithm just presented performs iterative operations on the matrix. Iterative constructs are devised in functional languages by abstracting the actual operations performed in an iteration from the control that guides iterative process. That control is implemented as a a higher-order function that receives—among its parameters—the function that implements one iteration. The most common of such constructs is the

higher-order function map, which receives a list and a fuction f that operates on an element of the list, and returns the list of results of applying f to each element of the list. Another such operator is $foldl$, which successively applies a binary function to all elements of the list until a result is obtained for the whole data structure. In this case, the implementation of the gaussian elimination algorithm will be using a version of $foldl$ targeted for arrays: $folda$. It receives an array, initial and final indices, an increment function for the indices, and a function that operates on the array at the current index and returns a possibly modified array. The implementation of $folda$, as well as those of *inc*, *dec* (increment, and decrement functions), and their type signatures appear in Figure 5.2. For pedagogical purposes, the type signature shown for $folda$ is not its principal type, but an instantiation that is compatible with all uses of the function within the program. I will present its principal type in Section 5.4.

Program

The program appears in Figure 5.3. The function $folda$ implemented all loops of the program. Note the nesting of *let*'s and *let*$*$'s in *scalerow* and *addtorow*. Two calls to function *copy* appear within $swap_2$!, called from $findrow$!. These could be avoided with a more elaborate type system. See Chapter 8 for details.

5.4 Higher-Order Types

The higher-order function $folda$ was introduced in the previous section, and, at that time, was given the type

$$folda \;::\; \tau_1 \xrightarrow{ws} \tau_2 \xrightarrow{rm} \tau_2 \xrightarrow{rs} (\tau_2 \xrightarrow{rs} \tau_2) \xrightarrow{rm} (\tau_2 \xrightarrow{rm} \tau_1 \xrightarrow{ws} \tau_1) \xrightarrow{ws} \tau_1$$

which is general enough not to interfere with the inference process of the other functions. However, the type the reconstructor will find for $folda$—its principal type—appears in Figure 5.4. This is rather daunting! However, according to Section 6.2, when full type reconstruction is performed, then the use information of higher-order

$$
\begin{aligned}
scalerow! &:: (Array_2\ Int) \xrightarrow{ws} Int \xrightarrow{rs} Int \xrightarrow[ws]{rs} (Array_2\ Int)\\
addtorow! &:: (Array_2\ Int) \xrightarrow{ws} Int \xrightarrow{rs} Int \xrightarrow[ws]{rs} (Array_2\ Int)\\
findrow! &:: (Array_2\ Int) \xrightarrow{ws} Int \xrightarrow{rs} Int \xrightarrow[ws]{rs} (Array_2\ Int)\\
forward! &:: (Array_2\ Int) \xrightarrow{ws} Int \xrightarrow{rs} Int \xrightarrow[ws]{rs} (Array_2\ Int)\\
backward! &:: (Array_2\ Int) \xrightarrow{ws} Int \xrightarrow{rs} Int \xrightarrow[ws]{rs} (Array_2\ Int)\\
gauss! &:: (Array_2\ Int) \xrightarrow{ws} Int \xrightarrow[ws]{rs} (Array_2\ Int)
\end{aligned}
$$

$$
\begin{aligned}
scalerow!\ a\ i\ n\ =\ & let{*}\ p = lookup\ a\ i\ i\\
& in\ let\ f = \lambda j\ a.let{*}\ x = lookup\ a\ i\ j\\
& \qquad in\ update_2!\ a\ i\ j\ x/p\\
& \quad in\ folda\ a\ i\ (n{+}1)\ inc\ f
\end{aligned}
$$

$$
\begin{aligned}
addtorow!\ i\ j\ n\ =\ & let{*}\ p = (lookup\ a\ j\ i)\\
& in\ let\ f = \lambda k\ a.let{*}\ x = lookup\ a\ i\ k\\
& \qquad y = lookup\ a\ j\ k\\
& \qquad in\ update_2!\ a\ j\ k\ (x{-}p{*}y)\\
& \quad in\ folda\ a\ i\ (n{+}1)\ inc\ f
\end{aligned}
$$

$$
\begin{aligned}
& findrow!\ a\ i\ j\ \{(lookup_2\ a\ j\ i) \neq 0\}\ =\\
& \ let\ f = (\lambda k\ a.(swap_2!\ a\ i\ k\ j\ k))\\
& \ in\ \ if\ (i == j)\ then\ a\\
& \qquad else\ folda\ a\ i\ n\ inc\ f\\
& \qquad\qquad findrow!\ a\ i\ j\ =\ findrow\ a\ i\ (j{+}1)
\end{aligned}
$$

$$
\begin{aligned}
forward!\ a\ i\ n\ =\ & let\ a'' = scalerow!\ (findrow\ a\ i)\ i\ n\\
& in\ folda\ a''\ (i{+}1)\ n\ inc\ (\lambda j\ a.addtorow!\ a\ i\ j\ n)
\end{aligned}
$$

$$
backward!\ a\ i\ n\ =\ folda\ a\ (i{-}1)\ 1\ dec\ (\lambda j\ a.addtorow!\ a\ i\ j\ n)
$$

$$
\begin{aligned}
gauss!\ a\ n\ =\ & let{*}\ a' = folda\ a\ 1\ (n{+}1)\ inc\ (\lambda i\ a.(forward!\ a\ i\ n))\\
& in\ folda\ a'\ n\ 0\ dec\ (\lambda i\ a.(backward!\ a\ i\ n))
\end{aligned}
$$

Figure 5.3: Implementation of Gaussian Elimination In-Place

$$
\begin{aligned}
folda \;::\; & \tau_1 \xrightarrow{\nu_7} \tau_2 \xrightarrow{\nu_8} \tau_2 \xrightarrow{rs} (\tau_2 \underset{\nu_2}{\overset{\nu_1}{\rightarrow}} \tau_2) \xrightarrow{\nu_9} (\tau_2 \underset{\nu_4}{\overset{\nu_3}{\rightarrow}} \tau_1 \underset{\nu_5}{\overset{\nu_6}{\rightarrow}} \tau_1) \underset{\nu_{11}}{\overset{\nu_{10}}{\rightarrow}} \tau_1 \\
& \textit{where} \\
& \nu_7 = cs \overset{alt}{\odot} (\nu_7\ (\nu_6\ cs)) \\
& \nu_8 = rs \overset{seq}{\odot} ((\nu_7\ (\nu_3\ cs)) \overset{par}{\odot} (\nu_8\ (\nu_1\ cs))) \\
& \nu_9 = (\nu_8\ cs) \overset{par}{\odot} (\nu_9\ cs) \\
& \nu_{10} = (\nu_7\ cs) \overset{par}{\odot} (\nu_{10}\ cs) \\
& \nu_{11} = (\nu_7\ (\nu_4 \overset{alt}{\odot} \nu_5)) \overset{alt}{\odot} (\nu_8\ \nu_2) \overset{alt}{\odot} \nu_{11}
\end{aligned}
$$

Figure 5.4: Principal Type of $folda$

functions cannot have any restrictions whatsoever. In fact, by analyzing the implementation of $folda$, there is no clue on the behavior of f—how f uses its arguments, and global variables—on the call $(f\ i\ v)$. Granted that the former type of $folda$ looks much simpler, and more likely to be understood, but it is just an instance of its principal type when the instantiations from Figure 5.5 are made.

It is certainly possible to allow the user to (partially) specify the extended types of the *functional* arguments. Given those extended types, the system can infer a type where all uses are completely specified. The user could even be allowed to provide alternative type signatures for functional types, and the type inferencer could choose the less restrictive signature in each use of the function. In the extreme, the system

$$
\begin{aligned}
\nu_1 &= rs & \nu_2 &= \bot \\
\nu_3 &= rm & \nu_4 &= \bot \\
\nu_6 &= ws & \nu_5 &= \bot
\end{aligned}
$$

Figure 5.5: Instantiations of Principal Type of $folda$

$$\begin{array}{lcl} \nu_7 & = & ws \\ \nu_8 & = & rm \\ \nu_9 & = & rm \\ \nu_{10} & = & cs \end{array} \qquad \nu_{11} = \bot$$

Figure 5.6: Solution to Fixpoint Equations of Type of *folda*

may find all possible instantiation to the higher order uses that would result in a legal type (i.e., that does not map any use expression to *wm*. The number of assignments to higher-order use variables grows combinatorialy with the number of higher-order use variables, but the number of "reasonable " assignments may not grow so rapidly.

Chapter 6

Type/Liability Reconstruction

6.1 Introduction

In this chapter, an algorithm for extended type reconstruction is presented. The algorithm computes a principal extended type *in the absence of any type information provided by the user.* An interesting feature of the reconstruction algorithm is that it benefits from subtyping, but there is no need to manipulate coercion sets.

Even though the Type/Liability inference rules were presented as a unity (Section 4.4), where information on types is used on liabilities, and vice versa, the algorithm presented in this Chapter performs a standard type reconstruction algorithm followed by an *extended* type reconstruction algorithm:

Type Reconstruction, à la Hindley-Milner

This part does a straightforward type reconstruction—there is no involvement with abstract uses or liabilities; and

Extended Type and Liability Reconstruction

This part takes type information collected by the Hindley-Milner algorithm to reconstruct the extended type of expressions by reconstructing all use and liability information.

An interesting result from this chapter is that even though the typing rules make use of a subtyping model on types, the typing algorithm does not need subtyping information at all in the case that total type reconstruction is performed. This is important because the system benefits from the expressiveness of subtyping without making the type reconstruction algorithm any more complex.

Section 6.2 presents a more restrictive language of extended types, which can be used under the assumption that total extended type reconstruction is performed. Sections 6.3 and 6.4 present a type reconstruction algorithm, and Section 6.5 shows that the type computed by the algorithm is guaranteed to be principal.

6.2 Reconstructible Extended Types and Type Schemes

The inference rules presented in the previous chapter make no assumptions on the form of extended type expressions that can be inferred. In fact, expressions like

$$\lambda \mathtt{f}.\lambda \mathtt{x}.(\mathtt{f}\ \mathtt{x})$$

may have, among others, any of the following types

$$(\tau_1 \xrightarrow{\nu} \tau_2) \xrightarrow{cs} \tau_1 \xrightarrow{\nu} \tau_2$$
$$(\tau_1 \xrightarrow{rs} \tau_2) \xrightarrow{cs} \tau_1 \xrightarrow{rs} \tau_2$$
$$(\tau_1 \xrightarrow{rs} \tau_2) \xrightarrow{cs} \tau_1 \xrightarrow{ws} \tau_2$$

All these types are *extended* types of the same standard type

$$(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$$

which turns out to be the principal standard type for the expression. As happens in standard types, it is reasonable to expect for extended types a principal type property, and that only principal types be computed by the reconstruction algorithm—the first type from the preceding example.

However, when full type reconstruction is performed (i.e., all type information is inferred from the program; the user does not provide any type information) it will always be the case that use information associated with higher-order functions cannot have any restriction whatsoever; i.e., all use information associated with higher-order functions must consist entirely of variables. Intuitively, the use information for a graph is gathered in the scope where the graph is created. In the definition of a function, the use of the bound variable to the function becomes apparent, but, in the case the bound variable is of a functional type, there cannot be any restriction on how the actual functional uses its arguments. In terms of types, there is a noticeable distinction between a *first-order* function type—a function type not inside the argument type of another function type—and a *higher-order* function type (otherwise, corresponing to functional arguments). A higher-order function type must have its uses uninstantiated—plain use variables. This is formally captured by the language of reconstructible types and type schemes, shown in Figure 6.1. Further, a use variable cannot appear more than once in higher-order position within a type.

The following theorem states that if all assumptions made on the types of variables and constants are from $TypeSch_{Ext}^{Assume}$, then a reconstructible type can be found for any graph that has a standard type. This actually implies that $TypeSch_{Ext}^{Assume}$ is a simpler subset than $TypeSch_{Ext}$, and that it is "closed".

Theorem 6.2.1 *For all graphs g, and assumption environments T, if the types of all constants are in $TypeSch_{Ext}^{Assume}$, and all the types in the assumption environment are also in $TypeSch_{Ext}^{Rec}$ (Figure 6.1), and g can be given standard type σ, then it is possible to infer a reconstructible type for the graph.*

Proof The proof is by induction on the structure of g, and following the inference rules from Section 4.4.

1. Constants

   $g = c$

   $C, T \vdash k : \mathcal{K}(k)$

$$
\begin{array}{rcll}
\hat{\sigma} \in TypeSch^{Rec}_{Ext} & ::= & \hat{\mu} \ where \ P & \\
 & & \mid \forall \alpha \hat{\sigma} & \\
 & & \mid \forall \nu \hat{\sigma} & \\
 & & & \\
\hat{\mu} \in MutType^{Rec}_{Ext} & ::= & \mid Array \ \hat{\tau} & Arrays \\
 & & \mid \hat{\tau} & \\
 & & & \\
\hat{\tau} \in Type^{Rec}_{Ext} & ::= & Int \mid Bool \mid \ldots & Basic \ Types \\
 & & \mid Pair \ \hat{\tau}_1 \ \hat{\tau}_2 & Pairs \\
 & & \mid \hat{\psi}_1 \stackrel{u_1}{\underset{u_2}{\rightarrow}} \hat{\mu}_2 & First\text{-}Order \ Functions \\
 & & \mid \alpha & Type \ Variables \\
 & & & \\
\hat{\psi} \in MutType^{Rec}_{Ext} & ::= & Array \ \hat{\phi} & Second\text{-}Order \ Mut. \ Types \\
 & & \mid \hat{\phi} & \\
 & & & \\
\hat{\phi} \in Type^{hi}_{Ext} & ::= & Int \mid Bool \mid \ldots & Second\text{-}Order \ Types \\
 & & \mid Pair \ \hat{\phi}_1 \ \hat{\phi}_2 & \\
 & & \mid \hat{\psi}_1 \stackrel{\nu_1}{\underset{\nu_2}{\rightarrow}} \hat{\psi}_2 & \\
 & & \mid \alpha &
\end{array}
$$

Note: use variables cannot appear more than once in higher-order position within a type.

Figure 6.1: Syntax for Reconstructible Types and Type Schemes

The theorem trivially holds for g, since the types of all constants are already in $TypeSch_{Ext}^{Assume} \subset TypeSch_{Ext}^{Rec}$.

2. Variables

   $g = v$

   $C, T \vdash v : \langle T(v), \bot_{Liab}[v \mapsto cs] \rangle$

   Like in the previous case, trivial, since $T(v)$ is in $TypeSch_{Ext}^{Assume} \subset TypeSch_{Ext}^{Rec}$.

3. Abstractions

   $g = (\lambda v.g')$

   $C, T[v \mapsto \hat{\mu}_a] \vdash g' : \langle \hat{\mu}_b, \bot_{Liab}[v_1 \mapsto u_1, \ldots, v_n \mapsto u_n, v \mapsto u_a, \xi \mapsto u_b] \rangle$

   Note that in order for the inductive hypothesis to be valid, $\hat{\mu}_a \in TypeSch_{Ext}^{Assume}$. It follows that $(\hat{\mu}_a \xrightarrow[u_b]{u_a} \hat{\mu}_b) \in TypeSch_{Ext}^{Rec}$.

4. Applications

   $g = (g_a\ g_b)$

   $C, T \vdash g_a : \langle \hat{\mu}_a \xrightarrow[u_b]{u_a} \hat{\mu}_b, L_a \rangle$

   $C, T \vdash g_b : \langle \hat{\mu}_a, L_b \rangle$

   By inductive hypothesis,

   $$C, T \vdash g_a : \langle \hat{\psi}_a \xrightarrow[u'_b]{u'_a} \hat{\mu}'_b, L_a \rangle$$

   with $\psi_a = \mu_a$ (i.e.,the standard types coincide, but the extended may not, due to differences on uses). Note that $\hat{\psi}_a \trianglerighteq \hat{\mu}_a$, since they just differ in their uses, and $\hat{\psi}_a$ does not have any instantiation in its uses, since it is a higher-order type. Then there exists S such that $S\ \hat{\psi}_a = \hat{\mu}_a$. Take $\hat{\mu}_b = S\ \hat{\mu}'_b$. It must be the case that $\hat{\mu}_b \in MutType_{Ext}^{Rec}$ because S cannot bind any use in higher-order position within $\hat{\mu}'_b$. This follows because

   $$\hat{\psi}_a \xrightarrow[u'_b]{u'_a} \hat{\mu}'_b \in MutType_{Ext}^{Rec}$$

   and, therefore its higher order use variables are different from any use variable in $\hat{\psi}_a$.

5. Strict Applications

   Same argument as previous item.

6. If-then-else

   $g = (if\ g_a\ g_b\ g_c)$

   $C, T \vdash g_a : \langle Bool, L_a \rangle$

   $C, T \vdash g_b : \langle \hat{\mu}, L_b \rangle$

   $C, T \vdash g_c : \langle \hat{\mu}, L_c \rangle$

   By inductive hypothesis, $\hat{\mu} \in MutType_{Ext}^{Rec}$, and this is precisely the type of the construction.

7. Let

   $g = (let\ v = g_a\ in\ g_b)$

   $C, T \vdash g_a : \langle \hat{\sigma}, L_a \rangle$

   $C, T[v \mapsto \sigma] \vdash g_b : \langle \hat{\mu}, L_b \rangle$

   By inductive hypothesis, $\hat{\sigma} \in TypeSch_{Ext}^{Rec}$, and thus, the environment T can be extended to map v to such a type. It also follows that $\hat{\mu} \in MutType_{Ext}^{Rec}$. This last type is the type of the construction.

8. Let$*$

   Same argument as previous item.

9. Fixpoint

   $g = (fix\ g_a)$

   $C, T \vdash g_a : \langle \hat{\mu} \xrightarrow[u_b]{u_a} \hat{\mu}, L \rangle$

   By inductive hypothesis,

   $$C, T \vdash g_a : \langle \hat{\psi} \xrightarrow[u'_b]{u'_a} \hat{\mu}', L \rangle$$

   where $(\hat{\psi} \xrightarrow[u'_b]{u'_a} \hat{\mu}') \in MutType_{Ext}^{Rec}$. Then $\hat{\mu}$ is the fixpoint of $\hat{\mu}'$ where each use variable of $\hat{\psi}$ has been replaced by the fixpoint of the corresponding use expression in $\hat{\mu}'$. $\hat{\mu} \in MutType_{Ext}^{Rec}$ since every use variable in higher-order position

in $\hat{\mu}$ was instantiated to a use variable in higher-order position in $\hat{\psi}$, hence, it continues with no restriction.

□

Maintaining the inferred types within $TypeSch_{Ext}^{Rec}$ is important since it will later be proved that there exists a principal type property for graphs that can be given a type in $TypeSch_{Ext}^{Rec}$. Also, reconstruction is possible for this simple language, but the algorithm presented in Section 6.4 does not generalize for arbitrary extended types.

6.3 Standard Type Reconstruction

The algorithm for type reconstruction is typical of a Hindley-Milner type system. Therefore, its presentation is omitted.

6.4 Extended Type and Liability Reconstruction

The type inference and coercion rules can be combined with an effective type reconstruction algorithm to yield a decidable type system. As mentioned before, an outstanding feature of this algorithm is that even though the typing rules were based on a subtyping structure of the domain, the algorithm is implemented without the standard techniques for subtyping, like the ones found in [Mitchell, 1984,Fuh and Mishra, 1989]. This is possible because a total liability reconstruction is performed, and thus there is no knowledge of higher-order behavior.

6.4.1 Computation of Abstract Uses and Liabilities

Two operations are performed on abstract uses:

- binary operations specified on Chapter 3, like $u_1 \overset{par}{\odot} u_2$; and

- fixpoint operations, introduced by *fix*-nodes, which impose recursive constraints on the uses of their functional argument.

Binary operations are solved by applying the corresponding operator to the argument uses if they are known. If either of the uses is not known then this computation is carried out symbolically (i.e., it is left expressed, but not computed), and can be resolved as soon as the missing value becomes available. Due to a lack of a better computational model, the solution that has been adopted is to keep a set of unresolved use expressions along with the liabilities. For that purpose, the domain *Liab* is introduced:

$$L \in Liab \;=\; Liab \times (UseVar \rightarrow AbsUse)$$

Operations on liabilities are extended to this domain:

$$\begin{aligned} \_ \odot \_ :&\; Liab \rightarrow Liab \rightarrow Liab \\ u :&\; Liab \rightarrow Liab \end{aligned}$$

$$\begin{aligned} \langle L_1, C_1 \rangle \odot \langle L_2, C_2 \rangle &= \langle L_1 \odot L_2, C_1 \sqcup C_2 \rangle \\ u\ \langle L, C \rangle &= \langle u\ L, C \rangle \end{aligned}$$

Fixpoint constraints are solved by the *pumping from bottom* technique [Young, 1988]. The current set of constraints is first broken into sets of mutually recursive constraints. Each set that is closed (the value of the elements of the set depend only on their own values or on previously computed values) is solved using the previously mentioned fixpoint iteration tecnique. The specifics on how these operations are eventually evaluated is discussed in Chapter 7.

6.4.2 Auxiliary Functions

For the algorithm, $S \in Subst$ denotes the substitution environment. It binds type variables to type expressions, and use variables to use expressions (Id is the identity substitution). The following auxiliary functions are assumed:

- Extended Type/Use Environment for constants:

$$\hat{\mathcal{K}} : Kon \rightarrow \langle MutType^{Rec}_{Ext}, AbsUse \rangle$$

This is a typical type environment for constants, except that it also contains the use of the anonymous object implied by the use of the constant.

- Pattern-Matching:

$$\begin{aligned} \mathcal{PM} : &\quad Type \rightarrow Type \rightarrow Subst \\ \widehat{\mathcal{PM}} : &\quad (MutType^{Rec}_{Ext} + Use) \rightarrow (MutType^{Rec}_{Ext} + Use) \rightarrow Subst \end{aligned}$$

This is a standard pattern-matching algorithm, or one-way unification which provides the most general unifier (a substitution) of two types or two uses, where no variable occurring in the first argument has been instantiated. The first argument is called the *expression*, and the second one, the *pattern*. $\widehat{\mathcal{PM}}$ acts on extended type, while $\mathcal{PM}$ ignores any information on uses.

- Extended Type Abstraction:

$$\hat{\_} : Type \rightarrow MutType^{Rec}_{Ext}$$

This function is the trivial abstraction function from standard types to extended types. As such, it makes no assumption on uses, thus mapping to the most general extended type that can be concretized to the given standard type.

- Generalization:

$$gen : MutType^{Rec}_{Ext} \rightarrow (Var \rightarrow MutType^{Rec}_{Ext}) \rightarrow MutType^{Rec}_{Ext}$$

$(gen\ \mu\ T)$ generalizes all use and type variables that occur free in μ, but not in T.

- Specialization:

$$spec : MutType^{Rec}_{Ext} \rightarrow MutType^{Rec}_{Ext}$$

This function produces the extended type instantiation; i.e., it instantiates all bound type, and use variables.

6.4.3 Algorithm

The main algorithm, $\mathcal{T}$ has as signature

$$\mathcal{T} : Graph_{Ann} \rightarrow (Var \rightarrow MutType^{Rec}_{Ext}) \rightarrow (MutType^{Rec}_{Ext} \times Liab \times Subst)$$

with the following inputs

- A λ_{st}-graph g, for which the extended type is to be inferred. $Graph_{Ann}$ stands for any λ_{st}-tree annotated with the (reconstructed) standard type of all constant nodes (k^{σ}) and bound variables (v^{σ}); and
- A type environment (T), which holds the extended types of bound variables.

and outputs

- the extended type of the graph,
- the liability of the graph, and
- a substitution environment.

$$\begin{array}{lcl} \mathcal{T}\ g\ T_i & = & case\ g\ of \\ & & \quad BODY \end{array}$$

To type the program, proceed by cases dispatching on the form of the expression being typed.

1. Constants

$$\begin{array}{l} g\ =\ k^{\mu} \rightarrow \\ \quad let\ \langle \hat{\mu}_1, u_1 \rangle = (\hat{\mathcal{K}}\ k) \\ \qquad S_1 = \widehat{\mathcal{PM}}\ \hat{\hat{\mu}}\ \hat{\mu}_1 \\ \qquad S_2 = \widehat{\widehat{\mathcal{PM}}}\ S_1\hat{\mu}_1\ \hat{\hat{\mu}} \\ \quad in\ \ (S_2\hat{\hat{\mu}}, \perp_{Liab}[\xi \mapsto Su_1], S_2) \end{array}$$

Constants do not affect any *named* object. However, they may capture anonymous objects, as the liability $\perp_{Liab}[\xi \mapsto u_1]$ indicates. Note how the actual standard type (μ) is manipulated to get the extended type $(S_2\hat{\mu})$.

2. Identifiers

$$
\begin{aligned}
g \;=\; & v \;\rightarrow \\
& \mathit{let}\ \hat{\mu}_1 = \mathit{spec}\ (T_i\ v) \\
& \mathit{in}\ \ (\hat{\mu}_1, \perp_{Liab}[v \mapsto cs], Id)
\end{aligned}
$$

This follows directly from the inference rule. The liability of an identifier is $\perp_{Liab}[v \mapsto cs]$. Referencing an identifier does not involve any manipulation to the anonymous objects. Note that the actual type is instantiated (specialized) using *spec*.

3. Lambda Abstractions

$$
\begin{aligned}
g \;=\; & (\lambda v^{\mu}.e_1) \;\rightarrow \\
& \mathit{let}\ \langle \hat{\mu}_1, L_1, S_1 \rangle = \mathcal{T}\ e_1\ T_i[v \mapsto \hat{\mu}] \\
& \mathit{in}\ \ \langle (S_1\mu)\ \overset{(Lv)}{(L\ \xi)}\ \hat{\mu}_1, S_1 L_1^{v,\xi}, S_1 \rangle
\end{aligned}
$$

The uses of the bound variable v, and anonymous objects are computed recursively by typing the body. Then use of v, and ξ must be removed from the resulting liability, as they become part of the extended type. The type of the bound variable (μ) can be trivially extended since $\hat{\mu}$ is a second order type (appears within the domain of a function type).

4. Applications

$$
\begin{array}{l}
g \;=\; (e_1\ e_2) \;\rightarrow \\
\quad \mathit{let}\ \langle \hat{\mu}_1, L_1, S_1 \rangle = \mathcal{T}\ e_1\ T_i \\
\qquad \langle \hat{\mu}_2, L_2, S_2 \rangle = \mathcal{T}\ e_2\ S_1 T_i \\
\qquad S_3 = \widehat{\mathcal{PM}}\ S_2\hat{\mu}_1\ (\hat{\mu}_2 \xrightarrow[\nu_2]{\nu_1} \alpha) \\
\qquad\qquad \mathit{where}\ \alpha,\ \nu_1,\ \nu_2\ \mathit{are\ new\ vars} \\
\quad \mathit{in}\ \ \langle S_3\alpha, ((\nu_1 \cdot S_3 L_2) \overset{par}{\odot} S_3 S_2 L_1) \sqcup \bot_{Liab}[\xi \mapsto (S_3\ \nu_2)], S_3 S_2 S_1 \rangle
\end{array}
$$

The computation of uses $(u_1 \overset{par}{\odot} (u \cdot u_2))$ cannot always be solved, at this stage, since it can be the case that either of these uses is unknown, but the expression can nevertheless be left expressed, and will be solved at any super-expression of first-order type. Also note that the correct use for the anonymous objects is

$$(\nu_1 \cdot (L_2\xi)) \sqcup (L_1\xi) \sqcup \bot_{Liab}[\xi \mapsto \nu_2]$$

In the case of a functional argument, the higher-order part of the function's extended type does not restrict the argument's behavior in any way. A pattern match of the instantiated domain part of the function to the argument type is powerful enough for the task—no need for full-fleshed unification.

5. Strict Applications

$$
\begin{array}{l}
g \;=\; \langle e_1\ e_2 \rangle \;\rightarrow \\
\quad \mathit{let}\ \langle \hat{\mu}_1, L_1, S_1 \rangle = \mathcal{T}\ e_1\ T_i \\
\qquad \langle \tilde{\mu}_2, L_2, S_2 \rangle = \mathcal{T}\ e_2\ S_1 T_i \\
\qquad S_3 = \widehat{\mathcal{PM}}\ S_2\hat{\mu}_1\ (\tilde{\mu}_2 \xrightarrow[\nu_2]{\nu_1} \alpha) \\
\qquad\qquad \mathit{where}\ \alpha,\ \nu_1,\ \nu_2\ \mathit{are\ new\ vars} \\
\quad \mathit{in}\ \ \langle S_3\alpha, ((\nu \cdot S_3 L_2) \overset{seq}{\odot} S_3 S_2 L_{11}) \sqcup \bot_{Liab}[\xi \mapsto (S_3\ \nu_2)], S_3 S_2 S_1 \rangle
\end{array}
$$

This case is essentially the same as the previous one, with the exception that the type of the argument is unified to a non-functional term, acording to the

corresponding rule for strict applications, and the resulting liability is more liberal (use of $\overset{seq}{\odot}$ instead of $\overset{alt}{\odot}$).

6. If-then-else

$$
\begin{array}{l}
g \;=\; (if\ e_1\ e_2\ e_3) \rightarrow \\
\quad let\ \langle \hat{\mu}_1, L_1, u_1, S_1 \rangle = \mathcal{T}\ e_1\ T_i \\
\qquad \langle \hat{\mu}_2, L_2, u_2, S_2 \rangle = \mathcal{T}\ e_2\ S_1 T_i \\
\qquad \langle \hat{\mu}_3, L_3, u_3, S_3 \rangle = \mathcal{T}\ e_3\ S_2 S_1 T_i \\
\qquad S_4 = \widehat{\mathcal{PM}}\ S_3 \hat{\mu}_2\ \hat{\mu}_3 \\
\quad in\ \ \langle S_4 S_3 \hat{\mu}_2 \sqcup S_4 \hat{\mu}_3, S_4 S_3 S_2 L_1 \overset{seq}{\odot} (S_4 S_3 L_2 \overset{alt}{\odot} S_4 L_3), S_4 S_3 S_2 S_1 \rangle
\end{array}
$$

Note that the restriction on the types of the expressions of both branches of if is that they both be coercible to the same type, but their types need not unify.

7. Let

$$
\begin{array}{l}
g \;=\; (let\ v = e_1\ in\ e_2) \rightarrow \\
\quad let\ \langle \hat{\mu}_1, L_1, u_1, S_1 \rangle = \mathcal{T}\ e_1\ T_i \\
\qquad \langle \hat{\mu}_2, L_2, u_2, S_2 \rangle = \mathcal{T}\ e_2\ S_1 T_i [v \mapsto (gen\ \mu_1\ S_1 T_i)] \\
\quad in\ \ \langle S_2 \hat{\mu}_2, ((L_2\ v) \cdot S_1 L_1) \overset{par}{\odot} L_2, S_2 \rangle
\end{array}
$$

Note the generalization step to transform μ_1 into an extended type scheme. Otherwise, it is a straightforward implementation of the inference rule for let.

8. Let*

$$
\begin{array}{l}
g \;=\; (let{*}\ v = e_1\ in\ e_2) \rightarrow \\
\quad let\ \langle \hat{\mu}_1, L_1, u_1, S_1 \rangle = \mathcal{T}\ e_1\ T_i \\
\qquad \langle \hat{\mu}_2, L_2, u_2, S_2 \rangle = \mathcal{T}\ e_2\ S_1 T_i [v \mapsto (gen\ \mu_1\ S_1 T_i)] \\
\quad in\ \ \langle S_2 \hat{\mu}_2, ((L_2\ v) \cdot S_1 L_1) \overset{seq}{\odot} L_2, S_2 \rangle
\end{array}
$$

Note the generalization step to transform μ_1 into an extended type scheme. Otherwise, it is a straightforward implementation of the inference rule for let.

9. Fix

$$
\begin{array}{l}
g \;=\; (fix\ e_1) \;\rightarrow \\
\quad let\ \langle \hat{\mu}_1, L_1, S_1 \rangle = \mathcal{T}\ e_1\ T_i \\
\qquad S_2 = \widehat{PM}\ S_1\hat{\mu}_1\ (\alpha \xrightarrow[\nu_2]{\nu_1} \alpha) \\
\qquad\qquad where\ \alpha,\ \nu_1,\ \nu_2\ are\ new\ vars \\
\quad in\ \ \langle S_2\alpha, ((S_2\nu_1) \cdot S_2L_1) \sqcup (\perp_{Liab}[\xi \mapsto S_2\nu_2], S_2S_1 \rangle
\end{array}
$$

The fix-operator is handled as a constant of type $(\alpha \rightarrow \alpha) \rightarrow \alpha$ in traditional Hindley-Milner type systems. However, fix cannot be correctly typed in the extended type system because in order to do so, the assumption has to be made on the functional argument it takes that the extended type of its domain be identical to that of its range, including any higher order behavior that the argument may have. This is not expressible in the restriction to the extended type system adopted. Instead, the domain's and range's types are matched. From the restrictions imposed on the extended type language, the type of e_1 is of the form $\hat{\psi} \xrightarrow[u_1]{u_2} \hat{\mu}$. The pattern-match then has the effect of matching use variables of ψ to corresponding uses of μ. From this matching, (possibly recursive) constraints are generated. These capture the intended behavior of fix.

6.5 Principal Extended Types

The algorithm presented in the previous section computes reconstructible types for graphs under the restrictions of Theorem 6.2.1. Assuming that programs have empty type assumptions, the results of that theorem will hold for any program provided that the constants have reconstructible types—a quick glance at Figure 4.4 shows that constants do satisfy this restriction.

In this section, it will be proven that in case any reconstructible type can be derived for a graph g, then there exists a (reconstructible) type $\hat{\sigma}$ for g also derivable from the rules such that any other type that can be deduced for g is a logic conse-

quence of $\hat{\sigma}$; i.e., $\hat{\sigma}$ is a *principal type* of g. Further, it will be proven that if g has a reconstructible type, then $\mathcal{T}\ g\ \perp_{Env}$ will find a principal type for g.

Lemma 6.5.1 *For all contexts $\overline{C}$, graphs g, coercion sets C, assumption environments T and T', and extended types $\hat{\sigma}_1$, and $\hat{\sigma}_2$ such that $\hat{\sigma}_1 \trianglerighteq \hat{\sigma}_2$, if*

$$C, T' \vdash \overline{C}[g] : \langle \hat{\sigma}_4, L_4 \rangle$$

by using the fact that

$$C, T \vdash g : \langle \hat{\sigma}_2, L_2 \rangle$$

then there exist $\hat{\sigma}_3$, L_3 such that

$$C, T' \vdash \overline{C}[g] : \langle \hat{\sigma}_3, L_3 \rangle$$

by using the fact

$$C, T \vdash g : \langle \hat{\sigma}_1, L_1 \rangle$$

and $\hat{\sigma}_3 \trianglerighteq \hat{\sigma}_4$

Proof The proof is by induction on the structure of $\overline{C}$. It will be observed for each construct that a weaker premise necessarily leads to a weaker conclusion. □

Definition 6.5.2 *A* canonical derivation *is one such that the premises to instantiation rules can only be conclusions of constant rules, or variable rules.*

Lemma 6.5.3 *If there is a proof for $T \vdash g : \mu$, then there exists a canonical derivation with the same conclusion.*

Proof By induction on the structure of the derivation of the type.

- All substitution nodes can be moved from conclusions to *all* premises by applying the substitution to the conclusion.

- Two instantiation nodes in chain—one being the conclusion of another—can be collapsed into an instantiation node that does both substitutions.
- After the above two transformations are done, all instantiation nodes will be on conclusions of constant rules, or on conclusions of variable rules.

□

Lemma 6.5.4 *If* $C,\ T\ \vdash g : \langle \hat{\mu}, L \rangle$, *then there exists a canonical derivation with the same conclusion.*

Proof Similar to previous lemma's. Just separate coercions from instantiations. □

Lemma 6.5.5 *If* $C,\ T\ \vdash g : \langle \hat{\mu}, L \rangle$, *then* $T\ \vdash\ \mu$.

Proof Just eliminate coercion steps in the derivation. All standard types coincide, since coercions only deal with use information (relevant to extended type, but with no counterpart in the standard type system). □

Lemma 6.5.6 *If* $T\ \vdash\ \mu$, *then* $C,\ T\ \vdash g : \langle \hat{\mu}, L \rangle$.

Proof The proof of Theorem 6.2.1 shows how to build the proof of the extended type. The intuitive idea is to add a coercion step in between any two steps of the standard type proof. □

Lemma 6.5.7 *For all graphs* g, *and assumption environments* T, *if*

$$(\mathcal{T} g\ T) = \langle \hat{\mu}, L, S \rangle$$

then there exists C *such that*

$$C, T\ \vdash g : \langle \hat{\mu}, L \rangle$$

Proof By induction, the algorithm *is* a canonical proof. For identifiers and constants, the algorithm infers the type of the identifier/constant, and immediately does an instantiation step. For all other graph structures, the algorithm first does coercion steps in order to get all conditions required to apply the rule corresponding to the structure of the graph. □

Lemma 6.5.8 *For all graphs g, coercion sets, C, and assumption environments T, if $C,T \vdash g : \langle \hat{\mu}_1, L_1 \rangle$, then $(\mathcal{T}\, g\ T) = \langle \hat{\mu}_2, L_2, S \rangle$, and $C \Vdash \{\hat{\mu}_2 \trianglerighteq \hat{\mu}_1\}$.*

Proof The proof is by induction on the structure of g, and following the inference rules from Section 4.4.

1. Constants

   $g = c^{\mu}$

   $C,T \vdash k : \mathcal{K}(k) = \langle \mu_1, L_1 \rangle$

   $\hat{\mu}_1 \trianglerighteq \hat{\mu}_2$, and $\hat{\hat{\mu}} \trianglerighteq \mu_2$

   $(\mathcal{T}\ c^{\mu}\ T) = \langle \hat{\hat{\mu}}, L, \perp_{Subst} \rangle$

   Note that in the derivation to find the principal standard type for g, the type for c was μ. Therefore, by Lemma 6.5.3, there exists a canonical derivation, by Lemma 6.5.6 there exists the corresponding derivation for the extended type. For that derivation, there also exists a canonical extended derivation. In that derivation, c is necessarily instantiated to $\hat{\mu}_2$ with $\hat{\hat{\mu}} \trianglerighteq \hat{\mu}_2$.

2. Variables

   $g = v$

   $C,T \vdash v : \langle T(v), \perp_{Liab}[v \mapsto cs] \rangle$

   $(\mathcal{T}\ v^{\mu}\ T) = \langle T(v), \perp_{Liab}[v \mapsto cs], \perp_{Subst} \rangle$

   Similar to previous case.

3. Abstractions

$g = (\lambda v^{\mu}.g_1)$

$C, T[v \mapsto \hat{\hat{\mu}}] \vdash g' : \langle \hat{\mu}_b, L[v \mapsto u_a, \xi \mapsto u_b] \rangle$

$(\mathcal{T}\ g\ T) = \langle \hat{\mu}_d, L[v \mapsto u_c, \xi \mapsto u_d], S \rangle$

Note that the inductive hypothesis implies that $\hat{\mu}_d \trianglerighteq \hat{\mu}_b$, $u_c \trianglerighteq u_a$, and $u_d \trianglerighteq u_b$. Under these conditions, it is immediate that

$$(\hat{\hat{\mu}} \overset{u_c}{\underset{u_d}{\rightarrow}} \hat{\mu}_d) \trianglerighteq (\hat{\hat{\mu}} \overset{u_a}{\underset{u_b}{\rightarrow}} \hat{\mu}_b)$$

4. Applications

$g = (g_a\ g_b)$

$C, T \vdash g_a : \langle \hat{\psi}_a \overset{u_a}{\underset{u_b}{\rightarrow}} \hat{\mu}'_a, L_a \rangle$

$C, T \vdash g_b : \langle \hat{\mu}_a, L_b \rangle$

$(\mathcal{T}\ g_a\ T) = \langle \hat{\psi}_c \overset{u_c}{\underset{u_d}{\rightarrow}} \hat{\mu}'_c, L_c, S_c \rangle$

$(\mathcal{T}\ g_b\ T) = \langle \hat{\mu}_c, L_d, S_d \rangle$

By inductive hypothesis, $(\hat{\psi}_c \overset{u_c}{\underset{u_d}{\rightarrow}} \hat{\mu}'_c) \trianglerighteq (\hat{\psi}_a \overset{u_a}{\underset{u_b}{\rightarrow}} \hat{\mu}'_a)$, and $\hat{\mu}_c \trianglerighteq \hat{\mu}_a$. Coercion steps will be taken to coerce $\hat{\psi}_c$ to $\hat{\mu}_c$, and $\hat{\psi}_a$ to $\hat{\mu}_a$. By the coercion rules on function types, it is implied from the premises that $\hat{\mu}_c \trianglerighteq \hat{\mu}_a$.

5. Strict Applications

Same argument as previous item.

6. If-then-else

$g = (if\ g_a\ g_b\ g_c)$

$C, T \vdash g_a : \langle Bool, L_a \rangle$

$C, T \vdash g_b : \langle \hat{\mu}, L_b \rangle$

$C, T \vdash g_c : \langle \hat{\mu}, L_c \rangle$

$(\mathcal{T}\ g_a\ T) = \langle Bool, L_d, S_d \rangle$

$(\mathcal{T}\ g_b\ T) = \langle \hat{\mu}', L_e, S_e \rangle$

$(\mathcal{T}\ g_c\ T) = \langle \hat{\mu}', L_f, S_f \rangle$

By inductive hypothesis, $\hat{\mu}' \trianglerighteq \hat{\mu}$.

7. Let

   $g = (let\ v = g_a\ in\ g_b)$

   $C, T \vdash g_a : \langle \hat{\sigma}, L_a \rangle$

   $C, T[v \mapsto \sigma] \vdash g_b : \langle \hat{\mu}, L_b \rangle$

   $(\mathcal{T}\ g_a\ T) = \langle \hat{\sigma}', L_c, S_c \rangle$

   $(\mathcal{T}\ g_b\ T) = \langle \hat{\mu}', L_d, S_d \rangle$

   By inductive hypothesis, $\hat{\sigma} \trianglerighteq \hat{\sigma}'$, and $\hat{\mu} \trianglerighteq \hat{\mu}'$. $\hat{\mu}$ is the extended type of the construct.

8. Let*

   Same argument as previous item.

9. Fixpoint

   $g = (fix\ g_a)$

   $C, T \vdash g_a : \langle \hat{\psi}_a \xrightarrow[u_b]{u_a} \hat{\mu}_a, L \rangle$

   $(\mathcal{T}\ g_a\ T) = \langle \hat{\psi}_c \xrightarrow[u_d]{u_c} \hat{\mu}_c, L_c, S_c \rangle$

   By inductive hypothesis, $(\hat{\psi}_c \xrightarrow[u_d]{u_c} \hat{\mu}_c) \trianglerighteq (\hat{\psi}_a \xrightarrow[u_b]{u_a} \hat{\mu}_a)$. Since the fixpoint calculation perfomed on both sides preserves the inequality, it is concluded that the lemma also holds for this case.

□

Theorem 6.5.9 *1. The language of reconstructible types and type schemes has the principal type property.*

2. For any graph g, and type assumptions T, the algorithm $\mathcal{T}$ finds a principal type for g if g can be given a standard type.

Proof Lemma 6.5.7 ensures that the algorithm $\mathcal{T}$ only computes valid types. By Lemma 6.5.8, it is ensured that the type inferred by the algorithm is coercible to any other type produced by any valid derivation, therefore it must be principal. □

Chapter 7

Implementation

7.1 Introduction

In this chapter I present some experimental results taken from an implementation of the type inferencer for the Poly-λ_{st}-calculus. This implementation is faithful to the type reconstruction algorithm presented in Section 6.4, and is written in the T language [Rees *et al.*, 1984,Slade, 1987]. The concrete syntax for this implementation is presented, as well as how it relates to λ_{st}-calculus, and how the reconstructor can be run. The syntax of expressions and types are introduced in Sections 7.2, and 7.3 respectively. Section 7.4 provides some insight on implementation details, and Section 7.5 shows some actual examples of graphs typed by the implementation.

7.2 Implementation

The type reconstruction program follows the guidelines set forth by the algorithm presented in section 6.4.

7.2.1 Concrete Syntax

The input language for the type-checker is an enhanced version of the λ_{st}-calculus with Lisp-like format. Its concrete syntax is

$$
\begin{array}{rcll}
f, v & \in & Ide & \textit{identifiers (T symbols)} \\
k & \in & Kon & \textit{constants} \\
i & \in & Int \subset Kon & \textit{integers (T numbers)} \\
e & \in & Expr & \textit{expressions} \\
\multicolumn{4}{l}{where\ e ::= n} \\
 & & \mid v & \\
 & & \mid \texttt{(}\ \texttt{lambda}\ v_1\ \ldots\ v_n\ e\ \texttt{)} & \\
 & & \mid \texttt{(}\ \texttt{if}\ e_1\ e_2\ e_3\ \texttt{)} & \\
 & & \mid \texttt{(}\ \texttt{let}\ v\ e_1\ e_2\ \texttt{)} & \\
 & & \mid \texttt{(}\ \texttt{let*}\ v\ e_1\ e_2\ \texttt{)} & \\
 & & \mid \texttt{(}\ \texttt{fix}\ e\ \texttt{)} & \\
 & & \mid (e_1\ \ldots\ e_n) &
\end{array}
$$

Note that unlike the abstract syntax for λ_{st}-calculus, labels are not associated to expressions in the concrete syntax. A consequence is that no sharing is expressed for arbitrary expressions. The only sharing mechanism provided is through the `let`-, and `let*`-bound variables. Both versions of `let` allow type polymorphism. The distinction between them is that `let` translates to a function application while `let*` translates to a strict application. The fixpoint operator is implemented by `fix`. There are also constants, λ-abstractions, and (uncurried) applications.

This language is translated to the λ_{st}-calculus in a straight-forward way, with the exception of abstractions and applications which need to be curried.

7.3 Types

Compared to the Abstract Syntax, the concrete syntax of types has the usual simplification, inherited from the Hindley-Milner type system, that quantifiers on type variables are not used. Instead, a sophisticated method is used to determine whether or not any specific type variable is quantified. Similarly, there are no quantifiers for use variables either.

The concrete syntax for types is:

$$\begin{array}{lll} \nu & \in & \{\hat{}\}{+}{+}Ide \quad \textit{Use Variables}^1 \\ \alpha & \in & \{\texttt{?}\}{+}{+}Ide \quad \textit{Type Variables} \end{array}$$

$$\begin{array}{lll} u \in AbsUse ::= & \texttt{u-bot} \mid \texttt{u-rs} \mid \texttt{u-rm} \mid \texttt{u-cs} & \\ & \mid \texttt{u-cm} \mid \texttt{u-ws} \mid \texttt{u-wc} \mid \texttt{u-wm} & \textit{Abstract uses} \\ & \mid u_1 \overset{alt}{\odot} u_2 \mid u_1 \overset{seq}{\odot} u_2 \mid u_1 \overset{par}{\odot} u_2 & \textit{Binary operators} \\ & \mid u_1 \cdot u_2 & \textit{Projections} \\ & \mid \nu & \textit{Use Variables} \end{array}$$

$$\begin{array}{lll} \hat{\sigma}, \hat{\chi}, \hat{\tau} \in Type_{Ext} ::= & \texttt{Number} \mid \texttt{Bool} \mid \ldots & \textit{Basic Types} \\ & \mid (\ \texttt{Array}\ \hat{\tau}\ ) & \textit{Arrays} \\ & \mid (\ \texttt{List}\ \hat{\tau}\ ) & \textit{Lists} \\ & \mid (\ \texttt{Pair}\ \hat{\tau}_1\ \hat{\tau}_2\ ) & \textit{Pairs} \\ & \mid (\ \hat{\tau}_1\ (\ u_1\ u_2\ )\ \hat{\tau}_2\ ) & \textit{Functions} \\ & \mid \alpha & \textit{Type Variables} \end{array}$$

Where `u-bot`, `u-rs`, `u-rm`, `u-cs`, `u-cm`, `u-ws`, `u-wc`, and `u-wm` correspond to $\perp$, rs, rm, cs, cm, ws, $ws \vee cm$, and wm, respectively; `Number` corresponds to the Int domain; $(\ \hat{\tau}_1\ (\ u_1\ u_2\ )\ \hat{\tau}_2\ )$ is the concrete syntax for $\hat{\chi} \overset{u_1}{\underset{u_2}{\rightarrow}} \hat{\tau}$; and type, and use variables are just any T symbol prefixed with `?`, and `^` (caret character) respectively.

[1]The operator _++_ has signature

$$(Set\ \alpha) \rightarrow (Set\ \alpha) \rightarrow (Set\ \alpha)$$

Elements of $A{+}{+}B$ are formed from the concatenation of one element of A to one element of B

7.4 Issues on Implementation

The type reconstructor is implemented in the T language [Rees *et al.*, 1984,Slade, 1987]. The novel part of the implementation is the manipulation of uses and liabilities, and therefore, are the only issues discussed.

As pointed out in the previous Chapter, the extended type reconstructor has the information on the standard type of expressions available. In this particular implementation, the standard type reconstructor actually builds an extended type template with the characteristic that its mapping to the standard type domain is the principal standard type for the expression. In particular, no restrictions are placed on uses. The extended type inferencer then specifies the restrictions that the uses embedded in the type have to satisfy.

Binary operations are sometimes applied to uses that are not yet known, as when an object that involves a functional argument. It is enough for the reconstruction algorithm to specify the operation to be performed. In the actual implementation, however, it is important that all operations that can be computed eventually be done. In this way, these operations share the same principle of a logical variable: their existence is first specified, but their value may not be given until much later. The implementation computes all binary operations which can be reduced. It is guaranteed that a first-order expression will have enough restrictions in its use facet that all restrictions on uses will be solved. In expressions yielding higher-order types, however, not all use variables are given enough restrictions to be able to be evaluated to an abstract use. This variables are given an equation dependent on other unknowns which precisely state their value. For expressions that also involve fixpoints, recursive constraints may arise. On solving fixpoint operations, standard graph techniques were used to recognize data dependencies among equations, and to classify them into sets of mutually recursive constraints.

7.5 Sample Runs

The type reconstruction routines are loaded into the T environment by loading the file `reconstructor.t`. The extended type reconstructor function is bound to function `etr`. It receives a λ_{st}-program, and returns its extended type with the corresponding fixpoint constraints on uses, and its liability. Also, there is the possibility of running only the standard type reconstructor, bound to the function `tr`. The following examples were run on the reconstructor:

Example 7.5.1 *This example types the identity function:* $\lambda x.x$*:*

```
> (etr '(lambda x x))

;multiple values:
 [0] (-> ?0 (CS BOT) ?0)
 [1] ()
 [2] ()
```

The answer is, as expected, the extended type $?0 \xrightarrow{cs} ?0$*. Its has empty liability and no unresolved fixpoint constraints.*

Example 7.5.2 $\lambda f.\lambda g.\lambda x.(f\ (g\ x))$

```
> (etr '(lambda f (lambda g (lambda x (f (g x))))))

;multiple values:
 [0] (-> (-> ?3 (^4 ^5) ?6)
         (CS BOT)
         (-> (-> ?0 (^1 ^2) ?3)
             ((^4 CS) BOT)
             (-> ?0 ((^4 (^1 CS)) (ALT ^5 (^4 ^2))) ?6)))
 [1] ()
 [2] ()
```

As expected, the answer has empty liability, and no recursive constraints. In a more familiar syntax, the type is

$$(?3\ \hat{}5\ ?6) \xrightarrow{cs} (?0\ \hat{}2\ ?3) \xrightarrow{\hat{}4\,cs} ?0\ \hat{}5 \overset{alt}{\odot} (\hat{}4\ \hat{}2)\ ?6$$

or, by doing α-renaming:

$$(\beta\ \xi_2\ \gamma) \xrightarrow{cs} (\alpha\ \xi_1\ \beta) \xrightarrow{\nu_2 cs} \alpha\ \xi_2 \overset{alt}{\odot} (\nu_2 \xi_1)\ \gamma$$

which is the principal extended type for the expression. Note how different the reconstructor manipulates the use of the bound variable, from the use of the anonymous objects. Further simplification on the type (not afecting its principal property) can be done if $(\nu\ cs)$ are replaced by ν. Although these are easy simplifications, and would highly improve readability, they are dependent on the particular structure of the domain of uses, and the operations defined among them. Therefore I restrain from performing such simplifications in this chapter.

Example 7.5.3 *iterate*

This example involves the fixpoint operator. The function **iter** *takes a functional $f : \alpha \to \alpha$, an object $x : \alpha$ and a number n, and iterates the function f on x n times. In order to show better the effect of the fixpoint operator,*

```
(define iter-fn
  '(lambda iter
     (lambda f x n
       (if (= n 0)
           x
           (iter f (f x) (- n 1))))))

(define iterate
  `(fix ,iter-fn))
```

```
> (etr iter-fn)

;multiple values:
 [0] (-> (-> (-> ?9 (^2 ^3) ?4)
             (^0 ^1)
             (-> ?4 (^5 ^6) (-> NUMBER (^7 ^8) ?9)))
         (CS BOT)
         (-> (-> ?9 (^2 ^3) ?4)
             ((PAR (^0 CS) (^5 CS)) BOT)
             (-> ?9
                 ((PAR CS (^5 (^2 CS))) BOT)
                 (-> NUMBER
                     ((SEQ CS (^7 CS))
                      (ALT BOT (ALT (ALT (ALT (ALT ^1 ^6)
                                              (^5 ^3)) ^8) (^7 BOT))))
                     ?9))))
 [1] ()
 [2] ()
```

It can observed that the uses of the functional argument **(lambda iter ...)** *do not correspond to the uses of the range of the function. The fixpoint operator creates these restrictions, namely, that*

```
^0 = (PAR (^0 CS) (^5 CS))
^1 = BOT
^5 = (PAR CS (^5 (^2 CS))
^6 = BOT
^7 = (SEQ CS (^7 CS))
^8 = (ALT BOT
          (ALT (ALT (ALT (ALT ^1 ^6) (^5 ^3)) ^8) (^7 BOT)))
```

```
> (etr iterate)

 [0] (-> (-> ?9 (^2 ^3) ?9)
         ((PAR (^0 CS) (^5 CS)) BOT)
         (-> ?9
             ((PAR CS (^5 (^2 CS))) BOT)
             (-> NUMBER
                 (CM (ALT BOT (ALT (ALT (ALT BOT (^5 ^3)) ^8) BOT)))
                 ?9)))
 [1] ()
 [2] ((^8 (ALT BOT (ALT (ALT (ALT BOT (^5 ^3)) ^8) BOT)))
      (^5 (PAR CS (^5 (^2 CS))))
      (^0 (PAR (^0 CS) (^5 CS))))
```

These fixpoint equations cannot be solved at this stage because they depend on values `^2` *and* `^3`*, which are not known. When this function is applied to a functional argument,* `(+ 1)`*, for example, then all its fixpoint equations can be solved:*

```
> (etr '(,iterate (+ 1)))

;multiple values:
 [0] (-> NUMBER (CM BOT) (-> NUMBER (CM BOT) NUMBER))
 [1] ()
 [2] ()
```

Many other examples have been run with the type reconstructor, including all those presented in Chapter 5.

Chapter 8

Conclusions

As this dissertation shows, the expressiveness of functional languages can be extended with the ability to destructively manipulate state without losing referential transparency. The degree to which the state can be destructively manipulated depends on many factors involved in the design of the system as a whole. Many trade-offs have been resolved in favor of a simpler type system, rather than a more expressive one. Exceptions were made in cases where added complexity would mean theoretical robustness, like in the case of reconstructible extended types—their introduction was instrumental for proving the *principal type property* for extended types. The experience gained here proves the feasibility of such extended type systems. Section 8.1 surveys several trade-offs that were resolved, and hints how the system changes when other choices are taken. Finally, Section 8.2 provides some concluding remarks on this dissertation.

8.1 Complexity vs. Expressiveness

The degree to which the state can be destructively manipulated depends on

- the actual structure of the domain of abstract uses—the operational properties that are analyzed, and

- the objects on which these properties are analyzed.

In this work, the domain of abstract uses contains 8 elements, resulting from the conjunctive evaluation of 3 operational properties; and the properties are controlled only on free variables and arguments to functions. The resulting system is very primitive, yet powerful enough to demonstrate the validity of this approach. In fact, although I have shown several programs that satisfy the type requirements, and as such, are guaranteed confluence under λ_{st}-calculus, limitations of the system are responsible for the fact that many programs that are indeed single-threaded, as per Definition 3.4.7, do not possess a valid type. Some of these programs can indeed be given a type if the system is suitably extended.

One limiting characteristic of the system (due to its simplicity) is the fact that the extended part of the type (i.e., all the use information) is structured homomorphically to the function constructor on types, i.e., the extension only deals with the usage of function arguments. For example, the extended type of *update*! is

$$\textbf{\textit{update!}} : \langle (Array\ \tau) \overset{ws}{\rightarrow} Int \overset{rs}{\rightarrow} \tau \overset{cs}{\rightarrow} (Array\ \tau), \bot_{Liab} \rangle$$

where the actual extension to the Hindley-Milner type system is just the annotations above the arrows:

$$\_ \overset{ws}{\rightarrow} \_ \overset{rs}{\rightarrow} \_ \overset{cs}{\rightarrow} \_$$

which indicate that the first argument is mutated, etc. The notation adopted in this thesis stresses the point of view that the functions are the entities that manipulate their arguments, and as such, the extension is to control on *how* the functions use their arguments.

In the case of a structured argument to a function, it is usually the case that the different parts of the structure are manipulated in different ways. In the *update*! example, *the system cannot distinguish that the structure of the array is mutated, but the elements are not.*

The type hierarchy from which the standard type system is based reflects this deficiency when it discriminates mutable data structures, e.g., arrays, from immutable

$$
\begin{array}{lll}
\sigma \in TypeSch & ::= & \mu \\
 & & \mid \forall \alpha \sigma \\
 & & \\
\mu \in MutType & ::= & Int \mid Bool \mid \ldots \quad \textit{Basic Types} \\
 & & \mid Array\ \mu \quad \textit{Arrays} \\
 & & \mid Pair\ \mu_1\ \mu_2 \quad \textit{Pairs} \\
 & & \mid \mu_1 \rightarrow \mu_2 \quad \textit{Functions} \\
 & & \mid \alpha \quad \textit{Type Variables}
\end{array}
$$

Figure 8.1: Syntax of Collapsed Type Expressions

ones. There cannot be arrays of arrays, or other more complex data structures. If the hierarchy is collapsed, as in Figure 8.1, the arrays become "first-class objects". In this case, all objects become mutable (since any object can contain an array within its structure), and the types of primitive rules, like lookup must reflect that generality

$$\textbf{\textit{lookup}} : \langle (Array\ \alpha) \xrightarrow{cs} Int \xrightarrow{rs} \alpha, \perp_{Liab} \rangle$$

since the assumption that the elements of an array are immutable no longer holds. Hence a potentially mutable part of the array is returned, and therefore, the array argument is captured.

A type system resulting from this modification would fail to type *swap*!

```
swap! a i j =
    let x = lookup a i
        y = lookup a j
    in update (update a i y) j x
```

forcing it to be written as

$$
\begin{array}{rcll}
\sigma \in \mathit{TypeSch} & ::= & \mu & \\
 & & \mid \forall \alpha \sigma & \\
 & & & \\
\mu \in \mathit{MutType} & ::= & \mid \mathit{Array!}\ \mu & \mathit{Mutable\ Arrays} \\
 & & \mid \mathit{Pair}\ \mu_1\ \mu_2 & \mathit{Pairs} \\
 & & \mid \tau & \mathit{Immutable\ Types} \\
 & & & \\
\tau \in \mathit{ImType} & ::= & \mathit{Int} \mid \mathit{Bool} \mid \ldots & \mathit{Basic\ Types} \\
 & & \mid \mathit{Array}\ \tau & \mathit{Immutable\ Arrays} \\
 & & \mid \mathit{Pair}\ \tau_1\ \tau_2 & \mathit{Pairs} \\
 & & \mid \mu_1 \rightarrow \mu_2 & \mathit{Functions} \\
 & & \mid \alpha & \mathit{Type\ Variables}
\end{array}
$$

Figure 8.2: Syntax for Mutable and Immutable Arrays

```
swap'! a i j =
    let x = copy (lookup a i)
        y = copy (lookup a j)
    in update (update a i y) j x
```

For small data structures, like integers, these **copy**'s might be avoided, or would happen anyway. However, for large data structures, these copies would be clearly unacceptable. Even if **swap!** could be provided as a primitive with its less constrained type, the basic deficiency is there, and it will show up in other computations as well (just try to exchange *three* elements of the array!)—fixing the type of **swap!** does not attack the source of the problem, but only its symptoms.

A more elaborate definition of the type system—one involving mutable and immutable versions of data structures, like the one in Figure 8.2, may ease this prob-

lem, but such a type system will surely need reasoning about subtyping in order to allow mutable and immutable versions of the structures behave similar on certain operations—*lookup*, and *update* for instance—but behave differently on mutators like *update*!. Although feasible, the resulting type system would certainly be more complex. In fact, such a type system was introduced in [Guzmán and Hudak, 1990].

When distinctions are made on how different parts of the structure of an object are manipulated by the function, then the important elements are the different substructures of the argument, and how they are manipulated by the function. The notation would then shift to annotate arguments, rather than functions. The type of *update*! would be under this extension

$$\textbf{\textit{update!}} : (Array^{ws}\tau^{cs}) \rightarrow Int^{rs} \rightarrow \tau^{cs} \rightarrow (Array\ \tau)$$

This signature makes the distinction between the use of the array (ws), and the use of its elements (cs). Pragmatically, this kind of knowledge is indeed necessary, as the examples presented in Chapter 5 show. Without it, the ability to manipulate complex data structures is impaired.

Annotating each type expression looks simple enough for first-order expressions, but it becomes unwieldly complicated for higher-order expressions. An attempt to create a sensible type system appears in [Guzmán and Hudak, 1991], where the capturing notion presented in this thesis is studied independently, and is transformed into a full fledged liveness property. A liveness analyzer for first-order languages based on extended types that incorporate liveness information is presented there.

Another limitation of the system hereby presented is the absence of "interprocedural" information. Information lost at the λ-abstraction level, such as relationship among formal parameters, could be of use in estimating a more precise type. The `if` function had to be introduced as a primitive in order to alleviate the interprocedural information loss. The function

```
if' a b c = if a then b else c
```

has the reasonable type

$$if' : Bool \xrightarrow{rs} \alpha \xrightarrow{cs} \alpha \xrightarrow{cs} \alpha$$

since it reads its boolean argument, and returns either the second, or the third argument, depending on the result of the test. However,

```
if' (lookup a i x) (swap! a i j) (swap! a i k)
```

will be ill-typed—a being multiple-threaded. Assuming a sequential implementation of a functional language, this would be distressing. Any strictness analyzer would recognize `if'` as a non-strict function on its second and third arguments. Thus, it is assumed that their evaluations will not be done until demanded inside `if'`. Further, any partial evaluator will replace the former expression by the latter. Then, why cannot `if'` have the same semantics as `if` in λ_{st}-calculus?. Why is λ-abstraction referentially opaque? Why is `if` a special type of expression? The answer to these questions is *simplicity!* The primitive `if` cannot be given a reasonable type under the type system—the system does not know about strictness.

If a non-sequential implementation was provided, then the strictness property would not be enough to guarantee that the evaluation of `if`'s second and third arguments are mutually exclusive, because, in fact, they are *not.* A speculative order reduction strategy may choose to evaluate `b` and `c` simultaneously with `a`, even though only one of them would be needed. This would be disastrous in presence of mutation.

The language of types is not expressive enough to provide a type for a nonstrict function that would exploit its nonstrictness when normal order reduction is fixed. But even if the expressiveness was available, there are other issues pertaining evaluation control in presence of mutation that need to be addressed in order for that expressiveness to be of practical use.

Additionally, guaranteeing sequentiality between the evaluation of two expressions is a very complex problem in contexts where functions are manipulated. Even when a function is in normal form—fully evaluated—its application may trigger further

reductions within the function. Thus, the use of any object captured by the function may be latent until the function is fully applied—when it returns an object of first-order type. In this dissertation, no attempt has been made to study the sequentiality properties of functions. Sequential restrictions have been used only in cases where objects of first-order type guarantee the feasability of enforcement of such a restriction.

8.2 Concluding Remarks

There are several achievements in this thesis, all of them contributing to the solution of the extended type system:

1. the Single-Threaded Lambda-Calculus, a calculus close to the Lambda-Calculus, but which is a more faithful abstraction of graph-reduction, the de-facto operational semantics of many functional languages;

2. the notion of Liabilities as a means to accumulate, and operate with the uses of all free variables.

3. the Extended Type System, which graciously combines the Hindley-Milner type system, with a special case of subtyping; and

4. the Type Reconstruction Algorithm, designed to be a post-processing to the standard type reconstruction algorithm, and which only involves pattern-matching instead of unification, and fixpoints in the finite abstract use domain for recursive definitions arising from fixpoint expressions.

The first two items were first presented in [Guzmán and Hudak, 1990]. The Single-Threaded Lambda-Calculus is a step forward in reasoning at an abstract level on how the implementation of a functional language operates. On that respect, and without recurring to the introduction of a store, the model makes provision for structure

mutation, a concept notably absent from the Lambda-Calculus. This concept, introduced at the abstract level, is available to the user, which translates in a calculus as powerful as the Lambda Calculus, and yet closer to pragmatic concerns as sharing of computation, and space reutilization. The calculus is not guaranteed to be confluent for arbitrary expressions that make use of mutators, but only for those that satisfy the type discipline. Earlier versions of items 3 and 4 were also introduced in [Guzmán and Hudak, 1990], but were substantially revised. The versions presented here are based on the experience gained in [Guzmán and Hudak, 1991]. The interrelation between liabilities and the extended type system has proven to be a useful one when analyzing properties that are of a denotational flavor—the property on the expression is the combination of the properties of the constituent subexpressions. The extension to the types has been done in such a way that

- the principal type property is satisfied—any well-typed expression will have a valid type that is more general than any other valid types for the expression, and
- the extended type system structure is able to absorb the lattice structure of uses, thereby introducing subtyping.

Restrictions to the richness of the extended type language based on pragmatic reasons, allowed the type system to benefit from the generality of the above items without introducing extra machinery in the actual process of type reconstruction. Due to the nature of the properties controlled by the abstract uses, unification was not required to actually compute the required uses. However, a delayed computation of uses is required for functions dependent on higher-order functionals, and fixpoint operations on the finite domain of uses is required to correctly solve fixpoint operations expressed in the calculus.

Finally, this thesis was made with the pragmatic intuition that type systems are widely used because of its simplicity, and theoretical robustness, properties that make them ideal for specifying program interfaces. Interesting trade-offs arise in extending

such systems, such as simplicity vs. expressiveness. Throughout the dissertation the emphasis has been done on simplicity over expressiveness, with the pragmatic intuition that a less expressive, but simpler type system is more likely to be used at all than a more powerful, but rather complex system.

Bibliography

[Barendregt, 1984] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[Bird and Wadler, 1988] Richard Bird and Philip Wadler. *An Introduction to Functional Programming.* Prentice-Hall, Inc., 1988.

[Bloss, 1989] Adrienne Bloss. *Path Analysis and the Optimization of Non-Strict Functional Languages.* PhD thesis, Yale University, May 1989.

[Eekelen, 1988] M.C.J.D. van Eekelen. *Parallel Graph Rewritting — Some Contribution to its Theory, its Implementation and its Applications.* PhD thesis, University of Nijmegen, December 1988.

[Felleisen, 1988] Matthias Felleisen. λ-v-CS: An extended λ-calculus for scheme. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 72–84, Snowbird, Utah, July 1988.

[Field and Harrison, 1988] Anthony Field and Peter Harrison. *Functional Programming.* Addison-Wesley Publishing Company, 1988.

[Fuh and Mishra, 1989] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proceedings of TAPSOFT 89.* Springer-Verlag LNCS, 1989.

[Gifford and Lucassen, 1986] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *Proceedings of 13th ACM Symposium on Principles of Programming Languages.* ACM, 1986.

[Girard, 1987] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Guzmán and Hudak, 1990] Juan Carlos Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science.* IEEE, 1990.

[Guzmán and Hudak, 1991] Juan Carlos Guzmán and Paul Hudak. Liveness analysis via type inference. In *Proceedings of the XVII Latin American Informatics Conference*, 1991.

[Hindley, 1978] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1978.

[Hudak *et al.*, 1992] Paul Hudak, Simon Peyton Jones, Philip L. Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: a non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[Hudak, 1986] Paul Hudak. A semantic model of reference counts and its abstraction (detailed abstract). In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programmming*, pages 351–363. ACM, August 1986.

[Lucassen and Gifford, 1988] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proceedings of 15th ACM Symposium on Principles of Programming Languages.* ACM, 1988.

[Milner, 1978] R.A. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mitchell, 1984] J.C. Mitchell. Coercion and type inference. In *11th ACM Symposium on Principles of Programming Languages*, pages 175–185. ACM, January 1984.

[Odersky, 1991] Martin Odersky. How to make destructive updates less destructive. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 25–26. ACM Press, January 1991.

[Peyton Jones, 1987] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, Inc., 1987.

[Rees *et al.*, 1984] Jonathan Rees, Norman Adams, and J. Meehan. *The T Manual.* New Haven, Conn., fourth edition, 1984.

[Slade, 1987] Stephen Slade. *The T Programming Language: A Dialect of Lisp.* Prentice-Hall, Inc., 1987.

[Wadler, 1989] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1989. M. Broy and C Jones, Editors.

[Wadsworth, 1971] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, Oxford University, 1971.

[Young, 1988] Jonathan Young. *Theory and Practice of Semantics-Directed Compiling for Functional Languages.* PhD thesis, Yale University, December 1988.

Appendix A

λ_{st}-Calculus Programs

A.1 Quicksort

```
;;; QuickSort:

;;; All arrays are passed along with lower, and upper bounds.

;;; swap two elements of the array
(define swap
  '(lambda a i j
     (let* x (copy (lookup a i))
       (let* y (copy (lookup a j))
         (update! (update! a i y) j x)))))


;;; rearranges the array according to pivot p,
;;; all elements less than p go to lower indices
;;; all elements greater than p go to higher indices
;;; returns the rearranged array and the border index
```

```
(define split
  `(let swap ,swap
     (fix (lambda split a i j p
            (if (> i j)
                (Pair a (copy j))
                (if (>= (lookup a p) (lookup a i))
                    (split a (+ i 1) j p)
                    (let* a' (swap a i j)
                      (split a' i (- j 1) p))))))))


;;; performs quicksort on the array
(define qs
  `(let split ,split
     (let qs (fix (lambda qs a i j
                    (if (>= i j)
                        a
                        (let* arr-idx-pair (split a i j i)
                          (let* k (copy (snd arr-idx-pair))
                            (let* a' (fst arr-idx-pair)
                              (let* a'' (qs a' i k)
                                (qs a'' (+ k 1) j))))))))
       (lambda a (lambda n (qs a 0 n))))))
```

A.2 Higher-Order Functions for Arrays

```
;;; Higher Order Functions on Arrays.

(define mapa!
  '(fix (lambda mapa! v i n inc f
```

```
          (if (= i n)
              v
              (let* x (copy (f i (lookup v i)))
                (mapa! (update! v i x) (inc i) n inc f))))))

(define mapafn!
  '(fix (lambda mapafn! v i n inc f
          (if (= i n)
              v
              (mapafn! (updfn! v i (f i)) (inc i) n inc f)))))

(define folda!
  `(fix (lambda folda! a i n inc f
          (if (= i n)
              a
              (folda! (f i a) (inc i)   n inc f)))))
```

A.3 Gauss Elimination

```
;;; increment function
(define inc
  '(lambda x (+ x 1)))

;;; decrement function
(define dec
  '(lambda x (- x 1)))

;;; scales v[i..n] so that v[i] becomes 1
;;; (it is assumed that v[j]=0 for j<i)
```

```
                (let backward! ,backward!
                  ,gauss!))))))))))

;;; Perform (etr g!)
```