# Lazy Specialization

Michael Jonathan Thyer

Submitted for the degree of Doctor of Philosophy

The University of York

Department of Computer Science

September 1999

# Abstract

This thesis describes a scheme to combine the benefits of lazy evaluation with partial evaluation. By performing specializations only when needed (lazily), the specialize-residualize decision is changed from being semantic to operational. It is demonstrated that a completely lazy evaluator is capable of eliminating towers of interpreters. The scheme is generalised, devising a new implementation of optimal evaluation, and it is demonstrated that optimal evaluators do not eliminate towers of interpreters.

It is argued that the concept of *scope* has too often been overlooked in the lambda-calculus. A new system of *depths* is introduced in order handle the issue of scope. The new approach leads to a much richer understanding of the issue of *sharing* in the lambda-calculus. Although optimal evaluators are well known, it is argued that less well understood degrees of sharing, in between the sharing of conventional functional languages and optimal evaluators, are of more practical use. A new classification of possible function body reduction strategies is shown to be analogous and orthogonal to argument reduction strategies.

# Contents

# List of Figures

# List of Tables

# Acknowledgements

I am grateful to Colin Runciman, my supervisor, for giving me the opportunity to pursue this research, for introducing me to many exciting areas of computer science research, for comments on drafts of this thesis and other work and for giving me the freedom to explore as I pleased. Thanks also to Alan Wood and Kevin Hammond for examining this thesis and suggesting many improvements. I would also like to thank my parents for their continued support and encouragement, and for instilling in me the belief that with enough determination, just about anything is possible. Thanks to the members of Programming Languages and Systems group at York for a conducive working environment and feedback on my talks. Thanks also to Graeme, Hazel, Nick and Tony for their contribution to my enjoyable time at York.

# Chapter 1

# Introduction

This chapter explains the motivation for the research, and the new approach taken. A summary of the contributions of this thesis is given and the remaining chapters are outlined. In order to give the reader a feel for the motivation and appeal of the research, this chapter deliberately glosses over some of the detail. This detail is left for later chapters.

## 1.1 Motivation

This research is motivated by the desire to make programming in more abstract and interpretive styles more practical and less costly, and to to do so in as unobtrusive a way as possible.

Two programming paradigms that are suited to abstract styles of programming are lazy evaluation [39] and partial evaluation [46]. By not evaluating arguments to functions before they are known to be needed lazy evaluation frees the programmer from deciding when evaluations should be performed, making it possible to abstract over the order of evaluation. This enables programs to compute with infinite data structures, such as the set of all possible chess games, or things which are not fully known yet, such as data received over a network connection, in just the same way that programs compute with things which are fully known. Lazy evaluation can help disentangle the separate concerns of control and data. For example a function used to build an infinite tree representation of chess games, is kept separate from the function(s) that will traverse (some finite part of) this tree. Lazy evaluation makes it very natural and trivial to write in styles that would require added support for concurrency in a strict language.

Partial evaluation enables programmers to write a generic program and have it transformed into a specialized one. Thus less code needs to be written and a programmer can more accurately commit themselves to just those design decisions they wish to. They don't have to write reams of tedious specialized code that may all need to be changed if one earlier design decision is changed.

## 1.2 Partial evaluation and sharing

Partial evaluation can be thought of as a form of sharing. Given a function `f` taking two arguments, that is going to be applied repeatedly with the same first argument, `s`, but with different second arguments, $\mathtt{d}_1, \mathtt{d}_2 ... \mathtt{d}_n$, then by partially evaluating `f` with respect to `s`, a specialized version of `f`, $\mathtt{f}_\mathtt{s}$, is obtained. The function $\mathtt{f}_\mathtt{s}$ can then be applied to each of $\mathtt{d}_1, \mathtt{d}_2 ... \mathtt{d}_n$, and the work which is independent of the second argument is shared, and not repeated. This sharing is achieved by pre-computation during the specializing phase.

Sharing work that is dependent only on the earlier arguments to a function, so that the work is not repeated if the partially applied function is then fully applied multiple times, is a characteristic of a fully lazy language [37]. This similarity was first noted by Holst [35]. However the work that is shared by full laziness is only that which is syntactically independent of later arguments, not computationally independent, this hinders the extent to which full laziness shares work. To increase the amount of work that could be shared Holst devised the syntactic transformation *improved full laziness*. This transforms the program so that more work can be shared by making more expressions syntactically independent of variables they are not computationally dependent on. However this transformation is inherently limited to first order programs, and in general no transformation can achieve the sharing effect that a higher-order partial evaluator can.

Further work was carried out by Holst and Gomard [36]. They identified the degree of sharing that would be required to achieve the same specializing effect as a higher-order partial evaluator, and called this degree of sharing *complete laziness*. They gave a number of transformations that could make a program completely lazy, but such transformations can only work in special cases.

Since this early work, no further work has been reported on combining partial evaluation with lazy evaluation.

```
type Var = String
type Label = String
data Exp = ENum Integer
         | EOp (Integer->Integer->Integer) Exp Exp
         | EVar Var
data Stmt = SAssign Var Exp
          | SIf Exp [Stmt] [Stmt]
          | SGoto Label
type Block = (Label,[Stmt])
type Program = [Block]
```

Figure 1.1: Abstract parse tree definition for a flowchart language.

```
lookup a ((key,val):table) = if a==key then val else lookup a table
update av@(a,v) (kv@(key,val):table) =
 if a==key then av:table else kv:update av table

evalExp (ENum n)    env = n
evalExp (EOp f a b) env = f (evalExp env a) (evalExp env b)
evalExp (EVar v)    env = lookup v env

evalProg prog env = evalStmt (lookup "main" prog) env where
 evalStmt [] env = env
 evalStmt (stmt:stmts) env = case stmt of
  SAssign var exp -> evalStmt stmts (update (var,evalExp exp env) env)
  SGoto label -> evalStmt (lookup label prog) env
  SIf cond yes no -> if evalExp cond env /= 0
                     then evalStmt yes env
                     else evalStmt no  env
```

Figure 1.2: An interpreter for a flowchart language.

## 1.3 Partial evaluation of a lazy language

To understand the appeal of combining partial evaluation with lazy evaluation, it helps to understand a little more about how a partial evaluator works.

Consider an interpreter for an imperative flowchart language, the abstract syntax tree for which is defined by the data type in Figure 1.1. A partial evaluator could specialize this interpreter with respect to a given flowchart program, such that the resulting residual program would take an initial environment and return the final environment. The partial evaluator works much like an interpreter except the values of some variables will be unknown. When the partial evaluator meets an expression it first tries to reduce any subexpressions, and then if it cannot reduce the current expression, it rebuilds the expression with the results from trying to reduce its subexpressions. The final expression constitutes the residual program, which will be run later when values for the unknown variables are known.

When partially evaluating the flowchart interpreter with respect to a flowchart

```
evalProg prog env = (lookup "main" prog') env where
 prog' = map (\(label,stmts)->(label,evalStmt stmts)) prog
 evalStmt [] env = env
 evalStmt (stmt:stmts) env = case stmt of
  SAssign var exp -> evalStmt stmts (update (var,evalExp exp env) env)
  SGoto label -> (lookup label prog') env
  SIf cond yes no -> if evalExp cond env /= 0
                     then evalStmt yes env
                     else evalStmt no  env
```

Figure 1.3: A *knot-tying* interpreter for a flowchart language.

program, the partial evaluator will know the first argument to `evalStmt`, the current list of statements to interpret, but not the second argument, the environment. The partial evaluator will produce a specialized version of `evalStmt` for each time `evalStmt` is applied to a different known argument, that is, there will be a specialized version of `evalStmt` for each statement in the flowchart program. When the partial evaluator meets an unreducible `if` expression it evaluates both branches of the `if` expression. This would result in non-termination except that the partial evaluator remembers (memoizes) which values it has specialized `evalStmt` to, and if it finds itself trying to specialize `evalStmt` to the same value again, then it just uses the result of the previous specialization. This memoization is essential to ensure termination of the partial evaluator if there are going to be cycles in the residual code.

Some specialization systems, intended for use in dynamic code generation, omit the memoing. This enables them to specialize faster, but the generated code will never contain cycles (§2.14.1).

The exciting thing about specializing a non-strict language, is the knot-tying that is possible. This makes it possible to generate specialized code with cycles in without requiring the memoing.

This can be demonstrated by rewriting the flowchart interpreter in a knot-tying way as shown in Figure 1.3. Here `evalStmt` is applied a bounded number of times to all the statements that it will ever be applied to, once and for all at the start, so creating `prog'`. In the process of creating `prog'`, `prog'` is actually used. This style of programming is only possible in a non-strict language.

## 1.4 Lazy specialization

Complete laziness cannot be achieved by transforming a program to be executed in a conventional language. But it can be achieved by implementing a completely lazy evaluator. This requires non-standard reductions under lambdas. This effectively partially evaluates a program concurrently with the full evaluation of the program. The program is specialized lazily. This has the interesting implication that deciding whether to specialize or residualize changes from being a semantic issue to an operational issue.

Specializing code dynamically makes it easier for programmers to integrate the benefits of partial evaluation into their programs. For example it may be worth specializing a string matcher in a text editor to a given string.

## 1.5 Towers of interpreters

Throughout this research a primary objective was to implement an evaluator that could pass the *tower of interpreters test* (§6.1). A tower of two interpreters is an interpreter interpreting an interpreter interpreting some program. An evaluator passing this test should be able to run the program at the top of this tower at the same speed regardless of the number of interpreters in the tower, excepting a bounded amount of time for the one-off specializing overhead. The purpose of constructing programs like this is that new languages more suited to the domain they are to be used in can be implemented easily in an interpretive style in another language. This language in turn may be either the specializing language or some layered interpreter on top of it. The language developed for all these experiments is deliberately minimal, with the intention that any features making the language easier to use should be implemented in an interpretive layer.

## 1.6 Chapter outline

**Chapter 2: Background** A review of the relevant background material is given.

**Chapter 3: Degrees of Sharing** Explanations are given of the notation used, the way scoped functions work, and how variables can be represented by integers. An explanation is given of how by losing sharing, conventional reduction

strategies duplicate work. Different ways of maintaining sharing are presented, culminating in optimal evaluation. The partial ordering that exists between different degrees of sharing is explained.

**Chapter 4: Reduction Rules** A definition is given of cyclic scoped reverse de Bruijn notation. This notation is new, and is and used to define the reduction rules used in the implementations of complete laziness and optimal evaluation. Examples are given of the reduction rules in action.

**Chapter 5: Implementation** The implementations are presented and some of the design decisions are discussed.

**Chapter 6: Results** First a demonstration of complete laziness eliminating a towers of lazy interpreters is given. Next two heterogeneous towers are used to demonstrate that new language features can be introduced in the tower, and to explore which language features are required. An example is given showing how complete laziness can specialize away the interpretive layer of an imperative language, and how the specializing effect is inherited by the imperative language. The tower of interpreters experiment is conducted with two optimal evaluators, neither pass. A number of examples previously presented with the BOHM optimal evaluator are repeated using interpreters with various degrees of sharing. This shows that many of them are tamed by a lesser degree of sharing than optimal evaluation.

**Chapter 7: Conclusions** Finally the conclusion discusses the significance of the contributions of the thesis and how these relate to the objectives. Current limitations and possible avenues for future research are discussed.

## 1.7 Contributions

**Chapter 2: Background**

- The identification of a difference between static full laziness and dynamic full laziness is original

**Chapter 3: Degrees of Sharing**

- The use of depth to delimit the scope of a function is original.

- The identification of a partial ordering between some degrees of sharing is original.
- The classification of reduction strategies in terms of: substitute-by-name, substitute-by-value and substitute-by-need is original. Their analogy to, and orthogonality with, call-by-name, call-by-value and call-by-need is original.
- The use of memo-tables to achieve call-by-need and substitute-by-need is original.
- Explaining optimal evaluation in terms of the simultaneous specialization of a function to multiple arguments is original.
- The generalization of a completely lazy evaluator to an optimal evaluator is original.

**Chapter 4: Reduction Rules**

- The notation used to represent graphs with memo-tables in a term-like fashion is original.
- The reduction rules for complete laziness and optimal evaluation are original.

**Chapter 5: Implementation**

- The implementation of full laziness by graph transformation is original.
- The implementation of complete laziness is the first ever.
- The implementation of optimal evaluation with memo-tables is original.

**Chapter 6: Results**

- The first ever evaluator to pass the tower of interpreters test is demonstrated.
- Examples demonstrate that interpreters used in a tower of interpreters may introduce additional features and the interpretive overhead is still eliminated.
- Existing implementations of optimal evaluators are shown to not pass the tower of interpreters test.
- The specializing effect of complete laziness is shown to be inheritable by an imperative language as well as functional languages.

- Numerous test programs are executed with lazy, fully lazy, completely lazy and optimal evaluators, and the results analysed.

# Chapter 2

# Background

This chapter reviews the background material related to the work conducted in this thesis.

## 2.1 Origins of the lambda calculus

The lambda calculus was devised in the 1930s by Alonzo Church [24] as a technique to model the intuitive concept of the *effectively calculable function*. It was Church's belief that any function that could be computed in a systematic fashion could also be computed via the lambda-calculus. Numerous other models of computation have been proposed, including most notably the Turing machine [79]. The set of functions computable by the lambda-calculus, Turing machines and various other models of computation has been proved to be the same. The belief that no more general model of computation can be devised is known as the *Church-Turing thesis* [51]. While previous claims about lesser models of computation have been proved false, no more general model of computation than the lambda-calculus or Turing machine have been discovered and the Church-Turing thesis is not seriously in question.

As a model of computation, the Turing machine approach has the advantage that it is immediately clear how such a machine could be made in reality, (excepting that unboundedly long paper tapes are hard to come by).

A Turing machine is however somewhat tedious to program, where as the lambda-calculus immediately supports the power of abstraction which makes functional programs more concise and reusable.

It is not so immediately clear how a lambda-calculus "machine" should be constructed. The more abstract nature of the lambda-calculus makes it more amenable

to alternative implementations.

From a mathematical viewpoint *how* a lambda-calculus "machine", is constructed is irrelevant, so long as such a machine *can* be constructed. But from a practical point of view, for programming languages based on the lambda-calculus, understanding of the lambda-calculus can lead to insights leading to better compilers and interpreters.

Many decades after the invention of the lambda-calculus, discoveries are still being made which will lead to further lambda-calculus inspired improvements in the programming of computers.

## 2.2 Lambda calculus definition

A term in the lambda-calculus is defined as either a variable, abstraction or application:

**Definition 2.1 (lambda terms)**

| | | | |
|---|---|---|---|
| Term | ::= | x | (variable) |
| | \| | ($\lambda$x.M) | (abstraction) |
| | \| | (M N) | (application) |

where x and y range over the infinite set of variables, and M, N, A, B and C range over the set of terms.

□

The syntactic overhead of the notation may be reduced by adopting the following conventions:

**Definition 2.2 (shorthand)**

| | | |
|---|---|---|
| ((A B) C) | = | (A B C) |
| ($\lambda$x. ($\lambda$y. A)) | = | ($\lambda$x. $\lambda$y. A) |
| ($\lambda$x. (A B)) | = | ($\lambda$x. A B) |
| (A ($\lambda$x. B)) | = | (A $\lambda$x. B) |
| ($\lambda$x. ($\lambda$$y_1$ .. $y_n$. A)) | = | ($\lambda$ x $y_1$ .. $y_n$. A) |

□

The same variable may appear in multiple places in a term. Each occurrence may be either a binding occurrence, a bound occurrence or a free occurrence. A variable occurrence is binding if it occurs as the variable part of an abstraction. A variable occurrence is bound if it occurs (directly or more deeply) within the term part of an abstraction which binds that variable. A variable occurrence is free otherwise.

The free variables of a term are computed by FV:

**Definition 2.3 (free variables)**

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} \\ \mathrm{FV}(A\ B) &= \mathrm{FV}(A) \cup \mathrm{FV}(B) \\ \mathrm{FV}(\lambda x.\ A) &= \mathrm{FV}(A) \setminus \{x\} \end{aligned}$$

□

For example FV(λx.x y z) = {y,z}. A term is said to be *closed* if it has no free variables, and *open* otherwise.

A lambda-term can be given a meaning by defining the beta-reduction rule which operates on lambda-terms.

**Definition 2.4 (beta-reduction)**

$(\lambda x.A)\ B \rightarrow_\beta A[x := B]$

□

The notation A[x := B] denotes substitution, that is the replacement of all free occurrences of variable x within the lambda-term A with the lambda-term B. How this is achieved is an implementation issue. The substitution is not entirely trivial and there are many alternative ways of achieving it.

Barendregt [18] gives the following definition of substitution:

**Definition 2.5 (substitution)**

$$\begin{aligned} x[x := A] &= A \\ y[x := A] &= y, \text{ if } x \neq y \\ (M\ N)[x := A] &= (M[x := A])\ (N[x := A]) \\ (\lambda x.M)[x := A] &= \lambda x.M \\ (\lambda y.M)[x := A] &= \lambda y.(M[x := A]), \text{ if } y \notin \mathrm{FV}(A) \text{ or } x \notin \mathrm{FV}(M) \\ (\lambda y.M)[x := A] &= \lambda z.(M[y := z][x := A]), \quad \text{if } y \in \mathrm{FV}(A) \text{ and } x \in \mathrm{FV}(M), \\ &\qquad\qquad \text{where } z \notin \mathrm{FV}(M) \cup \mathrm{FV}(A) \end{aligned}$$

□

The apparent complexity of this substitution definition obscures the minimalist elegance of the lambda-calculus. The complication arises from the use of variable *names* to associate variables with an abstraction. An example is given in §2.8 demonstrating how these names can *clash*. Alternative techniques can be used, for example, a variable can indicate which abstraction it is associated with by use of

a pointer simply pointing to the abstraction. The inelegance of this substitution definition is really a symptom of the fact that mathematicians like to formulate concepts such as the lambda-calculus on paper using letters of the alphabet, rather than a symptom of any conceptual untidiness in the lambda-calculus itself.

Note the inner substitution in the term (M[y := z][x := A]), this renames all free occurrences of variable y to z to prevent name clash. Such renaming is also known as alpha-conversion.

The beta-reduction rule states that a term of the form (($\lambda$x.A) B) can be reduced. Such a term is known as a reducible expression, or *redex* for short. When a term contains multiple redexes, the beta-reduction rule does not state which order they should be reduced in.

A number of alternative reduction orders have been defined. Those employed by programming languages based on the lambda-calculus typically do not reduce *all* redexes. The following definitions are useful in explaining some of the alternative reduction orders:

**Definition 2.6 (normal form)**

A term is in *normal form* (NF) if it contains no redexes.

□

**Definition 2.7 (head normal form)**

A term is in *head* normal form (HNF) if it is a variable, or an abstraction whose body is in HNF, or an application whose function part cannot be reduced to an abstraction. Equivalently, a term of the form ($\lambda x_1 ... x_n$.y $A_1 ... A_m$) where $n, m \geq 0$, is in HNF.

□

**Definition 2.8 (weak head norm form)**

A term is in *weak* head normal form (WHNF) if it is a variable or an abstraction, or an application whose function part cannot be reduced to be an abstraction. Equivalently, a term of the form (x $A_1 ... A_n$) where $n \geq 0$, or of the form ($\lambda$x.A) is in WHNF.

□

Programming languages based on the lambda calculus typically only perform reduction to WHNF.

Not all terms can be reduced to normal form. For example the lambda-term ((λx.x x) (λx.x x)), is not already in normal form as a beta-reduction is possible, however the term can only be reduced to itself so can never be reduced to normal form.

A reduction strategy which guarantees to reduce a term to normal form if a normal form exists is the *normal order* reduction strategy. By always postponing the reduction of the argument of an application until after that application has been performed, the possibility that the reduction of the argument will fail to terminate is avoided. The normal order reduction strategy is also known as call-by-name.

An alternative reduction strategy is the *applicative* order reduction strategy, where the arguments of applications are reduced to WHNF before the application is performed. Applicative order reduction has the advantage that if the argument of an application is required more than once, then reducing the argument before copies of it are made results in less work. For example the lambda-term ((λx.x x) A), where A is expensive to reduce. It would be cheaper to reduce A once before application instead of twice after. The applicative order reduction strategy is also known as call-by-value.

A function is termed non-strict if it does not evaluate its argument. A function is termed strict if it does evaluate its argument.

Ideally only arguments to strict functions should be evaluated before the function is applied. Such a reduction strategy would achieve the best of both worlds: the normalizing capabilities of normal order reduction, and the reduction sharing of applicative order reduction. However determining whether a function is strict or not is undecidable in general.

## 2.3 Graph reduction

Although deciding whether a function is strict or non-strict is, in general, undecidable, reducing the argument to a function before the function is called is not the only way to ensure the argument is only reduced once. The decision as to whether to reduce the argument can be postponed. A reduction mechanism that operates on graphs instead of lambda-terms, can copy pointers to the argument rather than copy the argument itself. The reduction of the argument can be postponed until the first time the argument is found to be needed, if ever, and the result of this reduction

shared by all occurrences of the variable within the function. If the argument turns out not to be needed, then it will not be evaluated at all.

The reductions performed by this graph reduction technique are equivalent to the same reductions being performed, but in a different order, on a lambda-term by an evaluator with the *impossible foresight* to know which functions are strict and which non-strict.

Wadsworth [82] describes such a graph reduction technique and calls its evaluation strategy call-by-need. This reduction technique works on acyclic lambda-graphs. Wadsworth also describes a substitution process which maintains sharing while copying them. This substitution process avoids copying graph which doesn't need to be copied. The substitution process described in Definition 2.5 will propagate substitutions right the way through a term. In contrast Wadsworth's substitution process only propagates a substitution as far as is necessary for the substitution to reach all the variables it binds[1]. There is no point substituting a term whose set of free variables does not include the variable the substitution binds.

Although it has been helpful to discuss the scope of substitutions in Wadsworth's technique as if they were performed using the substitution rules in Definition 2.5, Wadsworth actually performs the substitutions in one step, this makes it possible to preserve the structure of the graph being substituted.

Wadsworth describes a fairly expensive process for dynamically determining before the body of each function is copied, just how much of the graph need be copied.

He notes:

> *Clearly, this is one area awaiting innovative suggestions. One possibility is that there will be a simple decision procedure using connectivity matrices, or some other graph-theoretic concept ...*

There is a particular circumstance where this optimization helps, which is worth distinguishing. A term of the form ((λx.λy.A) B C) will reduce to ((λz.A') C) where A' = A[y:=z][x:=B], which in turn will reduce to A" where A" = A'[z:=C]. The important point here is that in the process of performing the substitution [z:=C] though A', there is no point in substituting [z:=C] though any occurrences of B which may exist in A', as there is no possibility of an occurrence of the variable z existing free in B.

[1]To be technically accurate the variables bound by any abstractions substituted must also substituted.

Arvind, Kathail and Pingali [6] describe a variant of Wadsworth's substitution process which, in the above example, prevents any attempt to substitute [z:=C] through occurrences of B in A'. However their technique is only applicable if reduction stops at WHNF. When performing a top-level beta reduction, the argument is tagged to indicate it is a closed-term. In the above example the term B will be tagged in this way, so the substitution [z:=C] will know that there cannot be any free variables in B.

Wadsworth's reduction mechanism reduces terms to NF, in contrast, all the other implementations discussed stop at WHNF. However Wadsworth's evaluator only starts reducing under lambdas once it has reduced the whole term to WHNF, since cyclic graphs are prohibited, there is no possibility that a lambda under which reductions have been performed will ever be applied subsequently.

It is worth noting how the reductions performed by a call-by-need reduction strategy can be related to those performed by a call-by-value strategy. The reductions performed by call-by-need with graph sharing are a reordering (and subset of), the reductions performed by a call-by-value without the benefit of graph sharing. The point of spelling this out, is that the same cannot be said for some of the sharing mechanisms described later.

A novel approach that can be considered to be what Wadsworth describes as an *innovative suggestion*, is described in §3.2.

## 2.4 Environment machines

Performing multiple substitutions through a lambda-term is expensive. An alternative is to build up an environment of variable bindings and evaluate open terms with reference to an environment.

This is the approach adopted by the Landin's SECD machine [55]. The SECD machine uses an applicative order reduction strategy (call-by-value). The SECD machine can be adapted to use call-by-need [23, 33], by replacing values with pointers to either values or terms. In this way only the pointer is duplicated and not a value or term. The first time an argument is found to be needed, the term is found by dereferencing the pointer and reducing the term in-place to a value, so that the next time the pointer, or another copy of the pointer is dereferenced, the value is found directly. The degree of sharing achieved in this way is called lazy [34].

The SECD machine achieves the same degree of sharing as Arvind et al's [6] technique. It does not achieve the full benefit of Wadsworth's minimal graph copying technique. If the body A of a function ($\lambda$x.A) contains subterms in which x does not occur, then Wadsworth's evaluator will reduce these terms no more that once, regardless of how often the function is applied. Whereas a SECD machine will reduce the term repeatedly in different environments.

## 2.5 Combinators

Combinators provide an alternative model of computation, equivalent in power to the lambda-calculus.

**Definition 2.9 (combinator-term)**

| | | | |
|---|---|---|---|
| Term | ::= | x | (variable) |
| | \| | ($\kappa$ $x_1$...$x_n$.M) | (combinator) |
| | \| | (M N) | (application) |

where x,y range over variables, and M,N range over terms. All variable occurrences must be either binding, or bound by the nearest enclosing combinator.

□

A lambda-term ($\lambda$x.M) with free variables $y_1$ ... $y_n$ can be translated into the combinator term (($\kappa$ $y_1$ ... $y_n$ x.M') $y_1$ ... $y_n$), where M' is the result of applying this translation to all terms in M.

This technique can be improved upon by converting a series of nested lambdas into a single combinator term, rather than converting them separately into several combinators. Alternative techniques for converting lambda-terms to combinator-terms are discussed in the next two sections.

Combinators are often written as a series of recursion equations instead of a single combinator-term. This is a simple translation to perform as combinator-terms by definition do not contain free variables.

For example, the recursive equations:

Zero f x = x

One f x = f x

Two f x = f (One f x)

Three f x = f (Two f x)

Main = Three Zero

are equivalent to the combinator-term:

($\kappa$ f x.f ($\kappa$ f x.f ($\kappa$ f x.f x))) ($\kappa$ f x.x)

where Main is a combinator of arity zero which corresponds to the root of the term.

**Definition 2.10 (combinator reduction)**

Combinator reduction can be defined as

($\kappa$ $x_1$...$x_n$.M) $N_1$...$N_n$ $\rightarrow_\kappa$ M[$x_1$ := $N_1$,...,$x_n$ := $N_n$]

□

This reduction rule implies that combinators can only be applied when *saturated*, i.e. there must be sufficient arguments for every variable to be bound. A practical advantage in using combinators is that unlike lambda-terms, the bodies of combinators are only substituted once. A disadvantage is that the opportunity to perform reductions between substitutions is lost, so it is not possible to share the benefits of any reductions which could otherwise be performed on a partially applied function.

Combinators have been exploited by functional languages implementers using both program independent (or *fixed*) combinators [81] and custom program specific combinators [37].

## 2.6 Fixed combinators

Techniques have long be known to exist to enable lambda-terms to be converted into combinator-terms utilizing a fixed set of combinators. Turner [80] noticed that the lazy SECD machine spent most of its time looking up variables in environments and discovered that combinator reduction performs computations more quickly. Turner describes a simple translation from lambda-terms to a combinator-term using the set of combinators S, K, I.

**Definition 2.11 (SKI combinators)**

S = ($\kappa$ a b c.(a c)(b c))

K = ($\kappa$ a b.a)

I = ($\kappa$ a.a)

□

**Definition 2.12 (lambda - SKI translation)**

| | | |
|---|---|---|
| $\lambda$x.A | $\rightarrow$ | [x]A |
| [x](A B) | $\rightarrow$ | S ([x]A) ([x]B) |
| [x]x | $\rightarrow$ | I |
| [x]y | $\rightarrow$ | K y, if x $\neq$ y |

□

Turner also describes combinator optimization rules. One significant optimization is:

S (K a) (K b) $\rightarrow$ K (a b)

Where a and b range over arbitrary combinator terms.

Turner observed that this optimisation led to remarkable "self-optimizing" properties. The expression ($\lambda$x.1+2) evaluated using Turner's fixed combinators, with the optimization rule above, reduces to the equivalent of ($\lambda$x.3), The evaluation of 1+2 need only be performed the first time the function is applied. This optimization achieves the same benefits as Wadsworth's implementation by avoiding unnecessary copying. However in this case it has been achieved by a compile-time optimization and avoids the run-time overhead of Wadsworth's technique. Wadsworth's technique still has the advantage that it admits the possibility of reductions being performed anywhere in a lambda-term, whereas the combinator-term equivalent is more restrictive.

## 2.7 Super combinators and lambda lifting

Evaluation based on small sets of fixed combinators results in computation progressing in very small steps. Although techniques to generalise the small set of fixed combinators to larger or even infinite sets have been developed [52], more success has been had with using program specific combinators.

Turning lambda-terms into program specific combinators has been called *lambda-lifting* by Johnsson [42], as nested lambdas are lifted out resulting in a set of recursive equations.

Hughes [37] and Johnsson [41] note how if lambda-terms are turned into custom combinators using the simple approach discussed in §2.5, the sharing benefit achieved by Wadsworth's and Turner's techniques is lost.

In order to regain this benefit, improved techniques to extract combinators from lambda-terms have been devised. These improved combinators have been named

*super*-combinators by Hughes who also coins the term *full*-laziness to describe the degree of sharing achieved by these super-combinators (and Turner's and Wadsworth's techniques).

Where the simple combinator extraction technique abstracts out free variables to turn open lambda-terms into combinator-terms, super-combinators are generated by abstracting out so called *maximally free expressions* (MFE) from open lambda-terms.

A free expression of a term is a proper sub-term whose free variables are also free variables of the term. A *maximally* free expression of a term is a free expression of the term which is not also a proper sub-term of another free expression of the term. For example the MFEs of the term ($\lambda$x.(x a) (b b)) are a and (b b), but not b.

By abstracting out MFEs instead of just free variables additional sharing is present in the arguments to the super-combinators.

Super-combinator based implementations achieve the same degree of sharing as Wadsworth's implementation, i.e. full-laziness. Where Wadsworth uses expensive run-time techniques, super-combinators use cheap compile-time techniques. However the technique Wadsworth uses to maintain full-laziness are more powerful than those used by super-combinators or Turner's fixed combinators. The identification of maximally free expressions only remains valid as long as reductions beneath lambdas are not performed. The following abstraction contains (essentially[2]) no free expressions:

($\lambda$ a.(($\lambda$ b.1) a) * 2 + 3)

However once the term (($\lambda$ b.1) a) is reduced to 1, the whole term (1*2+3) becomes a free expression.

As Wadsworth's reduction order never performs reductions under lambdas that will subsequently be applied, the extra power of his full laziness technique is never exploited. However it is worth noting its existence, particularly because no one has made note of it before. Its additional capabilities will be referred to as dynamic full laziness. Traditional full laziness will be referred to as static full laziness, when the difference is being highlighted.

Precisely what degree of sharing dynamic full laziness achieves will depend on the reduction order used. An example of a reduction order capable of exploiting the benefits of dynamic full laziness, is one that reduces functions to HNF before

[2]Technically 1, 2 and 3 are free expressions, but they are not worth abstracting.

applying them. Such a reduction order is discussed further in §3.7.

Peyton Jones and Lester [70] point out that full-laziness achieved by supercombinators can actually be performed independently of a combinator implementation. They present a translation from programs to full lazy programs. This enables full laziness to be achieved by other implementation techniques such as environment machines. However this translation isn't entirely independent of the implementation technique, the copying of free expressions is only avoided if no reductions are performed under lambdas. Takeichi [75, 49] coins the term *lambda hoisting* to describe an equivalent technique for achieving full laziness by program transformation.

In general no static translation can achieve the benefits of dynamic full laziness, as to do so, the translation would need to determine the strictness of a function such as ($\lambda$ b.1) above, which is general is undecidable.

## 2.8 de Bruijn notation

To understand the issue of variable name clashes consider the following $\lambda$-calculus reduction sequence (the underlines indicate the redex being reduced in each case):

$\underline{(\lambda \text{a.a a})\ \lambda \text{f.}\lambda \text{x.f (f x)}}\ \rightarrow$

$\underline{(\lambda \text{f.}\lambda \text{x.f (f x)})\ \lambda \text{f.}\lambda \text{x.f (f x)}}\ \rightarrow$

$\lambda \text{x.}\lambda \text{f.}\lambda \text{x.f (f x)})\ \underline{((\lambda \text{f.}\lambda \text{x.f (f x)})\ \text{x})}\ \not\rightarrow$

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

Even though all the variables are unique before reduction begins, there is still a danger of the variables becoming mixed up as this reduction sequence illustrates.

If reduction is performed only with closed terms, as is the case for most functional language implementations, then the situation above doesn't arise as no reduction are performed below lambdas.

The substitution rule in Definition 2.5 would catch this case and rename the x occurrences in the term ($\lambda$x.f(f x)) to a fresh variable, say y, resulting in ($\lambda$y.f(f y)), before substituting [f := x] into the term.

As an alternative to associating abstractions and variables by use of *names*, de Bruijn suggested using positive integers as *nameless dummies*.

For example the lambda-term ($\lambda$x.x($\lambda$y$\lambda$z.z x y)) is denoted ($\lambda \underline{1}(\lambda\,\lambda\,\underline{1}\ \underline{3}\ \underline{2})$) in de Bruijn notation. A variable $\underline{n}$ and its abstraction are associated by counting $n$ lambdas out from the occurrence of the variable. This notation has the advantage that where two lambda-terms may be considered equivalent *modulo renaming*, such

terms have a unique representation in de Bruijn notation.

However use of de Bruijn's notation appears to be inherently unsuited to sharing. Note how the two occurrences of the variable x in the lambda-term are denoted by $\underline{1}$ and $\underline{3}$ in the de Bruijn equivalent. Further examples of such issues are shown in the next section.

## 2.9 Explicit substitution

The substitution rule as specified by Church [24] or Barendregt [18] is a meta-operation. Although (($\lambda$ x.x x) A) and (A A) are both lambda-terms, the intermediate stages are not. This can hinder proving the correctness of implementations of programming languages based on the lambda-calculus. Implementations typically do not perform substitutions at the same time that a beta-reduction is performed as the beta rule would suggest.

In order to help bridge the gap between theory and practice, Abadi et al [1] study the $\lambda\sigma$-calculus where substitutions are formally part of the notation. They define reduction rules, which unlike the beta rule in the $\lambda$-calculus, are all atomic in nature. Just as with the $\lambda$-calculus the order in which these reduction rules should be applied is not specified, so the $\lambda\sigma$-calculus can be used to reason about a number of different reduction strategies.

The $\lambda\sigma$-calculus uses de Bruijn notation. When substituting an argument into a new context, free variables in the arguments may need to be *renumbered*. If the same argument is substituted into two different contexts then this renumbering will be different. Consider for example this lambda-term: ($\lambda$x.($\lambda$y.y$\lambda$z.y) ($\lambda$w.w) x). Reduction in de Bruijn notation can be performed as follows (where the long underline indicates the redex being reduced):

$\lambda\,\underline{(\lambda\,\underline{1}\,\lambda\,\underline{2})\ ((\lambda\,\underline{1})\underline{1})}\ \rightarrow$

$\lambda((\lambda\,\underline{1})\underline{1})\,\lambda((\lambda\,\underline{1})\underline{2})$

The free variable in the term $((\lambda\,\underline{1})\underline{1})$ has been renumbered as it is has been substituted into a context where it is now one lambda further away from its binding lambda.

This renumbering is not a problem if no reductions beneath lambdas are performed, as there will be no free variables to renumber. When reductions under-lambdas are performed, this renumbering may duplicate work.

A novel alternative to this scheme where it is the function body that is renum-

bered instead of the argument is explained in §3.2. Since the body of the function is copied during substitution anyway, renumbering the copy of the function body is preferable as it saves duplicating the argument.

The $\lambda\sigma$-calculus does not posses the Preservation of Strong Normalization (PSN) property. A calculus which extends the $\lambda$-calculus is said to posses PSN if terms which are strongly normalizing in the $\lambda$-calculus, are also strongly normalizing in the extended calculus. A term is strongly normalizing if it only admits reduction sequences of finite length.

To overcome this short comming in the $\lambda\sigma$-calculus, Kamareddine and Ríos devised the $\lambda$s-calculus with a more refined handling of substitution [47, 48].

## 2.10 Call-by-need lambda calculus

Proving a functional language implementation is correct with respect to the lambda-calculus is not sufficient to ensure the implementation is not grossly inefficient. An implementation which performed call-by-name reduction would be essentially unusable, although technically correct. Proving the correctness of a call-by-value evaluator is not difficult, as a call-by-value reduction sequence can be represented as entirely as a sequence of lambda-terms. However for the call-by-need reduction order, issues get more complicated. The sharing mechanisms employed by call-by-need implementations have no direct counterpart in a lambda-term, (or in Abadi et al's $\lambda\sigma$-terms).

To overcome this limitation various attempts have been made at providing a calculus which can accurately capture the issue of sharing. Launchbury [57] defines a natural semantics for lazy evaluation which precisely defines which reductions are shared by a lazy implementation. The definition is however in terms of a global heap mapping variable names to closed terms. This approach is of a too low-level nature to be useful for reasoning about alternative reduction orders or proving the correctness of syntactic transformations.

Independently Ariola and Felleisen [4] and Maraist, Odersky and Wadler [62] devised very similar definitions of a call-by-need lambda calculus. These approaches provide a semantics directly in terms of a linear representation of lambda-terms. This makes them more amenable to source-level reasoning about the behaviour of programs and the correctness of syntactic transformations.

Both these definitions of a call-by-need lambda calculus have the deficiency that they do not the include the recursive bindings (i.e. letrec) found in most functional languages. In fact attempts to include such a construct results in confluence being lost. A reduction system is said to be confluent if it is the case that when two reduction sequences beginning with the same term diverge (by reducing different redexes), they are always able to reconverge (by reducing appropriate redexes).

Arvind et al [7] give the following example of a reduction system with recursive bindings losing confluence:

```
odd  n = if n==0 then False else even (n-1)
even n = if n==0 then True  else odd  (n-1)
```

If the application of `even` within the definition of `odd` is reduced, then `odd` becomes directly recursive and there is no longer mutual recursion. Similarly if the application of `odd` within the definition of `even` is reduced. But once one of these redexes is reduced there is no way to make the recursive definition *stop* being recursive, or to make the non-recursive definition *become* recursive. So it will never be possible for these diverging reduction sequences to reconverge.

From an implementors viewpoint, this loss of confluence is not as serious as a loss of confluence could be. The correct values will still be computed. From a mathematical viewpoint, loss of confluence makes reasoning about the correctness of program transformations more difficult.

To overcome these difficulties, Ariola and Blom study *infinite* normal forms [3] and give a new definition of confluence, *skew* confluence [22]. Informally, in the even/odd example above, if reduction of the applications of `even` and `odd` are continued forever, in the limit, the definitions in the initially diverging reduction sequences converge.

The techniques they used were original and significant. They used lambda-*graphs* rather than lambda-terms, much like Wadsworth did. However their graphs are cyclic and so naturally support the recursion introduced by a letrec construct. In comparison, previous equivalences between the recursive bindings in functional languages and the lambda-calculus, were achieved via the rather clumsy use of a Y-combinator. Most significantly Ariola and Blom introduce the idea of *scope* in a lambda-graph. Where Wadsworth's technique recomputes the minimum scope of a function every time the the function is applied, Ariola and Blom's technique remembers the scope as part of the graph.

The issue of scope does not feature at all prominently in mathematical accounts of the lambda-calculus or functional languages implementation.

The graph reduction performed by typical implementations of functional languages [38, 17, 68, 71] is only concerned with two types of graph: The graphs which are syntactically introduced by use of letrecs in the source code, and the top-level graphs which are dynamically manipulated and which enable closed-terms to be shared.

Neither of these uses of graph require the concept of scope: the source-level graphs are substituted in their entirety in one go and so the issue of scope is unimportant; the top-level graphs are only concerned with closed-terms, and not the bodies of functions, so again the issue of scope is unimportant.

From a mathematical stance, scope can be seen as something of an implementation issue to save the cost of needless (but otherwise harmless) substitution of closed terms. But for most implementations, the issue of scope has been essentially optimized away as the implementations do not repeatedly substitute terms. Source-level graphs are substituted in their entirety to become top-level graphs, which will not themselves be substituted.

The re-introduction of scope helps bridge the gap between mathematical accounts of the lambda calculus, and practical implementations.

This idea of scope will be seen to play an important role in the new reduction mechanisms explained in the next chapter.

## 2.11 Optimal evaluation

Wadsworth noticed the normal order graph reduction performed by his implementation did not always reduce a term to normal form in the minimal number of beta-reductions. He raised the question [82]:

> *Are there other orders of graph reduction which further improve the performance of graph reduction ? Is there even a minimal reduction algorithm, i.e. one for which there is never a shorter reduction to normal form ?*

A problem arises with open-terms with two (or more) free variables. Examples of such terms are given in §3.8. Alternative reduction sequences, result in the variables being bound in different orders. These different reduction sequences are not just

re-orderings or subsets of each other (in contrast to the relationship between call-by-need and call-by-value). It is not in general possible to know beforehand which order to perform reductions so as achieve the shortest reduction sequence.

Perhaps it would be possible to somehow postpone deciding which reduction sequence to use, in an analogous way to postponing the reduction of arguments. However it seems unlikely that such an approach could be used, as it provides no answer to the question of which reduction to perform instead. In contrast when call-by-need postpones a decision, there is always a reduction it can be getting on with instead.

Lévy [59] studied this problem and found a way to relate the redexes in the alternate reduction sequences. He noted that a redex, with free variables, in one reduction sequence may appear as several copies, with some of the variables bound, in another sequence. Such copies are known as *residuals*. So in one reduction sequence this redex is reduced in a single step, but in another reduction sequence all the residuals of this redex must each be reduced. Lévy definition of an optimal evaluator requires it to reduce all these residuals in one step, analogously to call-by-need reducing the copies of redexes made by call-by-name in one step. Lévy's other requirement for an optimal evaluator is that it not perform unneeded reductions, the issues of avoiding the duplication of work and avoiding unneeded work being orthogonal. Lévy called the simultaneous reduction of copies or residuals of a redex a *parallel*-reduction. He called a parallel-reduction which reduces *all* the residuals of a redex a *complete* reduction.

The relationship which exists between redexes in one reduction sequence, and residuals of redexes in another, is such that, in general, there will not be a reduction sequence which contains the redexes but not the residuals. This means that the reductions performed by an optimal evaluator with some sharing mechanism will not correspond to the reductions found in any *one* reduction sequence without that sharing mechanism. (In contrast to call-by-need).

It was not until about a decade after Lévy's work that Lamping [54] and Kathail [50] independently came up with sharing mechanisms, similar to each other's, which made the implementation of an optimal evaluator possible. The key insight which their approaches share is the concept of a *shared context*. A context is a term with a hole in it, where the hole can be denoted by [ ].

Consider the following reduction sequence:

($\lambda$ x.(x 1) (x 2)) ($\lambda$ y.($\lambda$ z.z) y)

(($\lambda$ y.($\lambda$ z.z) y ) 1) (($\lambda$ y.($\lambda$ z.z) y) 2)

(($\lambda$ z.z) 1) (($\lambda$ y.($\lambda$ z.z) y) 2)

Both of 1 and y occur in the same context, namely (($\lambda$ z.z) [ ]), and this context can be reduced to ([ ]), independently of how the hole is filed. A sharing mechanism which shares contexts makes it possible for (($\lambda$ z.z) 1) and (($\lambda$ z.z) y) to be reduced in the same beta-reduction.

Both Lamping's and Kathail's techniques for sharing contexts involve the use of *sharing* and *unsharing* nodes, and a graph representation of lambda-terms. Where as sub-terms in Wadsworth's graph representation could be reached and shared by any number of other terms, sharing nodes impose the restriction that all sharing must be via sharing nodes.

The sharing and unsharing nodes make it possible for a context to be shared. The route through the sharing nodes through which a shared graph is reached, determines the route through unsharing nodes through which the context's hole is filled. This use of sharing nodes to capture the common structure in two terms when this common structure in not a simple shared sub-term is known as *superposition*.

Sharing and unsharing nodes are used in beta-reduction. Instead of copying the entire body of a function body when applied, the sharing nodes are used to superimpose the body of the function with its original variable still in-place, and the body of the function with the new argument bound.

These sharing and unsharing nodes reduce the graph in between them as they propagate through the graph, when corresponding sharing and unsharing nodes meet they cancel out, and the superposition is over. When sharing and unsharing nodes which do not correspond meet, they propagate through each other, duplicating each other in the process.

Determining whether sharing and unsharing nodes correspond or not is the most complicated part of the technique. The complication arises as contexts can interact in non-trivial ways, they do not form a simple hierarchy, as they can overlap with each other, Lamping handles this complexity by adding a number of control nodes.

The overhead of processing these control nodes is very high. Although the number of beta-reductions might be minimal, the cost of each beta-reduction is not bounded as the cost of handling the control nodes is not bounded.

Gonthier, Abadi and Lévy [32] significantly simplified Lamping's reduction rules, by noticing connections to Girard's [31] linear logic and geometry of interaction, they demonstrated that Lamping's technique could be considered an example of Lafont's [53] interaction nets.

None of Lamping, Kathail or Gonthier et al published details of any implementations they may have had for their designs, nor any experiments conducted using such implementations.

Asperti and Laneve showed how previous optimal evaluation algorithms could be generalised into a new class of higher-order rewriting system called Interaction Systems [12]. They used this new system to show how primitives such as arithmetic and conditional operations could be handled by an optimal evaluation algorithm. Asperti [9] went on to devise a system of *safe operators* to reduce particular 'safe' sequences of control operators to simpler configurations. These optimizations, in many typical cases, change an exponential reduction time to a polynomial or even linear reduction time.

Asperti describes his implementation of an optimal evaluator, provides results of experiments conducted [2], and makes the implementation publicly available [8]. Further experiments with this optimal evaluator are conducted in Chapter 6.

## 2.12 Incremental computation

The term *partial evaluation* can be traced back to Lombardi [60]. Lombardi also coined the term *incremental computation*, and was concerned with programming on-line interactive computer systems. He wanted to move away from a model of computation where a computer was given a complete program and data to perform batch-processing, to a model where *holes* could exist in both the program and data, which would be filled in interactively by the user later.

More recently Field [29] was similarly motivated to study techniques to aid in the incremental use of computers. He was concerned with the repeated application of a program, (such as a theorem proving system, programming environment or compiler), to input that differs only slightly from the previous. Ideally the computations which are unchanged from one application to another should be shared and not recomputed.

Field devises a system of delayed substitutions in order to increase the sharing

achieved in the reduction of lambda-terms. However his system is not optimal, he makes the remark [29]:

> *[complete parallel-reduction] requires that existing sharing in a graph be respected during the process of substitution, ... In order to implement optimal reduction, it thus seems that more powerful reduction systems (i.e. non-left-linear systems that can "test" for equality of sub-terms) or graph rewriting systems with capability beyond those of term graph rewriting are required.*

The use of memo-tables described in §3.8, is a novel solution to this problem, it provides a way to test for the equality of sub-terms.

## 2.13 Partial evaluation

A partial evaluator is a program that takes as arguments a program and data for some of the arguments of the program and returns the result of specializing the program to that data.

If the program to be specialized, `F`, takes two arguments `s` and `d`, then a partial evaluator, `PE`, will specialized `F` to a value of `s` such that the following holds:

`PE(F,s)(d)` = `F(s,d)`

The arguments to a program are classified as either static or dynamic, in the example above, there is one static argument and one dynamic. The partial evaluator is able to reduce the parts of `F` which are dependent only on the static arguments. However in general the partial evaluator must not reduce all such terms, as this would result in non-termination. In practice partial evaluators must be conservative in how much they specialize in order to ensure termination.

The use of a partial evaluator is beneficial if the speed-up achieved by the specialized program more than compensates the cost of performing the specialization. This is particularly likely to be the case if the resulting specialized program is used many times.

Programs that have been shown to benefit from partial evaluation include: parsers, pattern matchers, computer graphics and neural network training [46] amongst others.

One of the first areas identified where use of a partial evaluator would be beneficial is the specialization of interpreters [30]. Futamura noted that given an

interpreter, `int`, which takes two arguments `prog` and `data` corresponding to the program to evaluate and the data which should be supplied, the following relation holds:

$$\underbrace{\texttt{PE}(\texttt{int}, \texttt{prog})}_{\text{compiled program}}(\texttt{data}) = \texttt{int}(\texttt{prog}, \texttt{data})$$

If the language the interpreter `int` is written in is different from the language it interprets then the above use of a partial evaluator has the effect of translating the program `prog` from one language to another, so achieving a compilation effect.

If a partial evaluator were to be used frequently in this way, with the same interpreter but with different programs, then the partial evaluator itself would be a prime candidate for specializing. Futamura noted that the following relationship holds:

$$\underbrace{\underbrace{\texttt{PE}(\texttt{PE}, \texttt{int})}_{\text{compiler}}(\texttt{prog})}_{\text{compiled program}}(\texttt{data}) = \texttt{int}(\texttt{prog}, \texttt{data})$$

If this technique is used with a program other than an interpreter, then the result is known as a *generating extension*, rather than a compiler. For example the generating extension of a general purpose string-matching program would take strings and generate a fast specialized string matching program.

This line of thinking was extended on last step further by Beckman et al [21]. They noted that if several interpreters are to be converted into compilers by use of a partial evaluator, then it may be worth specializing a partial evaluator to itself. So the following relationship holds:

$$\underbrace{\underbrace{\underbrace{\texttt{PE}(\texttt{PE}, \texttt{PE})}_{\text{compiler generator}}(\texttt{int})}_{\text{compiler}}(\texttt{prog})}_{\text{compiled program}}(\texttt{data}) = \texttt{int}(\texttt{prog}, \texttt{data})$$

The *compiler generator* is also referred to as *cogen*. However a cogen could be used to generate programs other than compilers. If instead of applying a cogen to an interpreter, it was applied to a general-purpose parser, the result would be a program that took grammars and produced a specialized parser. In fact some specialization systems are hand written cogens, and don't bother with the partial evaluation stage at all.

These three uses of a partial evaluator to effect compilation have subsequently

become known as the first, second and third Futamura projections [3] .

At the time at which such ideas were put forward however, no partial evaluator existed which was both sophisticated enough to be able to specialize itself, and simple enough to be specialized by itself. Whether it was even possible to write such a partial evaluator was an open question, until more than a decade after Futamura's work [30] Jones et al succeeded in writing such a partial evaluator [44].

## 2.14 Staged computation

Staged computation is a generalisation of partial evaluation where the concepts of static and dynamic data are generalised to any number of stages, so allowing the specialized result of one stage to be specialized further by the next stage.

### 2.14.1 MetaML

MetaML [74] is a system that combines staged computation with dynamic code generation. It uses annotations to determine what to specialize. Since what will be specialized is fully determined at compile-time, the system cannot pass the interpretive towers test (§1.5). Knowing statically what you will specialize however enables much faster specializing to take place. Generating extensions can be created statically which when applied to dynamic data will very quickly generate the corresponding specialized code.

Partial evaluation ensures termination when specializing programs that will result in cyclic residual programs, by memoizing the specialization of functions with respect to the values they are specialized with. Something similar can be achieved with the knot-tying that is only possible in a non-strict language. MetaML doesn't use either of these techniques and so cannot create residual programs with cycles in. This limits its applicability to removing interpretive layers.

### 2.14.2 PGG

Unlike MetaML, PGG (program generator generator) [77] is used statically. PGG generates generators which generates generators which ... and so on, with the last

[3]The descriptions given here gloss over the important detail of the difference between a program and the function denoted by that program, just as Beckman et al do [21]. Jones et al give a more rigorous treatment [46].

one generating programs. However the decisions regarding what will be specialized are all made before the first generator is created. This makes the generators much faster, but not flexible enough to pass the interpretive towers test (§1.5).

## 2.15 Partial evaluation is fuller laziness

The similarity between the sharing achieved by full laziness and partial evaluation was first noted by Holst [35]. He devised the transformation *Improved Full Laziness*, briefly discussed in §3.6. This was shown to be equivalent to a first order partial evaluator. Subsequent work by Holst and Gomard [36] to study ways to further this specializing effect led to the term *complete laziness*[4]. They showed a number of transformations that could maintain the degree of sharing achieved by a completely lazy evaluator, and explained that it would be impossible for any such static transformation scheme to maintain that degree of sharing for arbitrary expressions.

Holst and Gomard describe a sequence of normal order reduction strategies, call-by-name, lazy, fully lazy and completely lazy. They define completely lazy reduction as that achieved by a higher-order partial evaluator, and claim it achieves a strictly greater degree of sharing than full laziness, and speculate that complete laziness is the last step in such a sequence.

They go on to give an algorithm to implement complete laziness.

However their algorithm loses sharing in a serious way. In the process of performing substitutions though graphs, it does not maintain any sharing already present in the graph. Although their algorithm is defined to work on terms and not graphs, graphs will be created the first time a function with more than one occurrence of its bound variable is applied. This loss of sharing is similar to that resulting from Field's approach [29].

Regarding complete laziness achieving a strictly greater degree of sharing than full laziness, Holst and Gomard do not fully explain how this should be achieved. They claim their completely lazy algorithm must be evaluated by a fully lazy evaluator. However evaluating their interpreter with a fully lazy interpreter would not make their interpreter maintain full laziness.

As explained later in §3.8, an evaluator that performs reductions under lambdas

[4]The use of the same word *complete* in the term complete laziness [36] and in Lévy's term complete parallel-reduction [59] (see §2.11) is coincidental.

only maintains the full laziness property if dynamic full laziness is used. Maintaining dynamic full laziness is expensive. Since complete laziness without full laziness has remarkable properties in its own right, complete laziness is best not defined as achieving a strictly greater degree of sharing than full laziness.

As for complete laziness being the end of a sequence of normal order evaluators where each maintains more sharing than the previous, optimal evaluation [58] has been shown to not lose any sharing, and so should be considered the end of such a sequence of reduction orders. In fact as shown in §3.4 this *sequence* is really a partial ordering.

## 2.16 Small

Augustsson's [14] language named *small* is worth mentioning, not because of any specializing effect but because of the unusual trick it performs using fixed combinators. A progression of combinators can be described:

- fixed combinators, language implementation specific,
- super combinators, program specific,
- dynamic combinators [78], computation specific.

The use of dynamic combinators can be motivated by arguing that super combinators are more suited to running less abstract styles of programming. The more higher order functions are used the less suited super combinators are, so if combinators are specialized according to the actual values they are being used with the are suited for use in more abstract styles of programming, including (potentially) passing the tower of interpreters test. However, fixed combinators, by virtue of being independent of the program being run, can pass a variant of the tower of interpreters test, what could be called the tower of *compilers* test.

The complier for small is written in small and it compiles the functional language into fixed combinators. The unusual part is that instead of building the compiled result as a data structure to be printed out or interpreted, it builds the new combinator tree as applications of the combinators in the underlying system. Thus the expression that was compiled at run-time will run just as fast as the code that was initially present in the system. This could be extended to a tower of compilers.

Recursive bindings are not supported by *small*. Instead the Y combinator is used, just as with explicit substitution (2.9), the call-by-need lambda-calculus (2.10) and Wadsworth's graph reduction (2.3).

Unlike optimal evaluation (2.11), staged computation (2.14) and partial evaulation (2.13), *small* does not perform reductions under lambdas and so has no specializing effect.

## 2.17 Summary

The research reviewed in this chapter can be broadly categorized into four areas:

- lambda calculus,
- functional language implementation,
- sharing, and
- specialization.

None of the existing research in these areas provides a solution to the towers of interpreters problem. The work presented in the next chapter resulted from the pursuit of this problem. The techniques developed in this pursuit also lead to greater understanding of each of the four research areas listed above.

# Chapter 3

# Degrees of Sharing

In this chapter first an explanation is given of the syntax and semantics of the language to be used. Then a new representation for graphs to be used in graph reduction is presented. Conventional call-by-need evaluation is demonstrated using this new graph representation. Examples are given of how different degrees of sharing maintain and lose sharing. The partial ordering that exists between these degrees of sharing is presented.

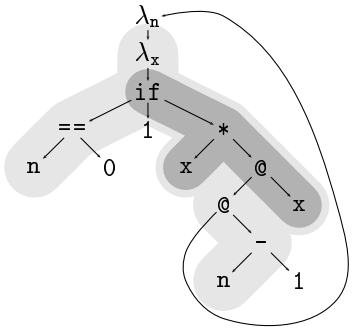
## 3.1 Syntax and semantics

The language used is called *Ef* and has a syntax much like Haskell's [69]. *Ef*'s semantics are non-strict and untyped; it supports integers and standard operations upon them, it also has booleans, Scheme-like [25] type-less pairs and the empty list.

The program in Figure 3.1 illustrates a number of points regarding *Ef*'s concrete syntax. *Ef* doesn't perform eager pattern matching [13]. The apparent uses of patterns in the `fibs` example are just syntactic sugar to save excessive use of `head` and `tail`. *Ef* uses Haskell's syntax for lists and tuples as syntactic sugar for type-less pairs. The empty list, `[]`, is the same as the empty tuple, `()`. Other than that everything else is just as in Haskell.

```
take 15 fibs
where
fiblist a b = a:fiblist b (a+b)
fibs = fiblist 1 1
take n (l:ls) = if n==0 then [] else l:take (n-1) ls
```

Figure 3.1: An *Ef* program to list the first 15 Fibonacci numbers.




```
power where
power n x =
 if n==0 then 1 else x*power (n-1) x
```

Figure 3.2: Graph and text representation of `power`.

## 3.2 Graph representation

The `power` function in Figure 3.2 serves as an example to explain the graph representation used. The important things to learn from this diagram are that the graphs used can be *cyclic* and that each node in the graph has a *depth* associated with it. The depth is represented by the darkness of the shading.

This depth shading is used to delimit the scope of a function. Nodes at depth zero are free from variables and are shown on a white background. Nodes at depth one are within the scope of a lambda at depth zero, and may be able to reach the free variables at depth one within the body of the lambda abstraction. These nodes are shaded in the lightest grey. Likewise for nodes at greater depths, a node at depth $n$ is within the scope of a lambda at depth $n$-1 and may be able to reach the free variables at depth $n$ to depth one within the body of the lambda abstraction.

The depth of an argument node is unaffected by function application. Even through a node may become shared during the reduction process, the node still exists within the same scope.

The names of variables become irrelevant, which lambda a variable is bound by is determined by depths. Replacing variable names with integers is a technique attributed to de Bruijn [26] (see §2.8). The new technique presented here is similar but in a sense reversed. This gives the advantage that it can naturally handle cycles and sharing, but adds the burden that scope of functions must be maintained (the bounds to the shaded regions). Even though the names are strictly redundant they will continue to be used to name bindings and variables in diagrams to make the

diagrams more readable.

The use of depths to delimit the scope of a function is original, and can be seen as a solutions to Wadsworth's [82] (see §2.3) appeal for an *innovative suggestion*. They can also be seen as a generalization of Arvind et al's [6] (see §2.3) proposed solution, with the advantage that this solution works for reduction under lambdas also. All the reduction mechanisms in this chapter are presented in terms of this new technique, and for that and other reasons they are also all original.

The graph notation presented side-steps the need to handle let bindings, but introduces the concept of graph sharing, and equality between node addresses. A graph can be considered an expression with let bindings by use of a single top-level let expression and additional let expression placed under each lambda abstraction. The address of each node can be used for the name of the corresponding let binding.

Some transformations on expression with let bindings, such as full laziness [70], have corresponding transformations in the graph representation, examples of this can be seen in §3.5. Other transformations only correspond to the identity transformation in the graph representation, for examples let-floating [67].

In a sense expressions with let bindings can be seen as a lower level representation than graphs. Ultimately functional language compilers transform programs into a sequential series of instructions with jumps. Transformations on linear representations of programs are useful in reaching this goal in a more optimal way. However such transformations are of a lower level and more implementation specific nature than the work discussed in this thesis.

## 3.3 Top-level reduction

There are a number of reduction orders which can be applied. First considered are conventional *weak head normal form* (WHNF) reduction of closed terms where no reductions occur beneath lambda abstractions.

The diagrams in Figures 3.3 and 3.4 illustrate WHNF reduction. Cycles are supported directly, there is no need for the Y-combinator. The diagrams show each of the 17 stages of reduction in applying `power` to `2` and `7`, computing $7^2$.

At each reduction stage any new nodes created are shown in bold. Whenever a function is applied, all the graph shaded is copied, and all occurrences of the bound variable replaced by a single shared occurrence of the argument. The depth of the

newly created graph is one less than that being copied, nodes shaded dark grey are copied lighter and nodes shaded light grey are copied without shading.

This process of graph copying with variable replacement is called substitution.

In reducing a graph to WHNF, reductions never occur beneath lambdas. This can be seen in the reduction sequence: the `power` function never changes and even though lambdas are substituted, creating new functions (see Figures 3.3(b), 3.3(f) and 3.4(k)), these functions never change once formed.

## 3.4 Degrees of sharing

The term *lazy* implies non-strict semantics, and a certain degree of *sharing*. Non-strictness ensures that terms are only reduced when needed. Sharing tries to avoid repeating reductions.

In terms of function arguments a strict reduction scheme will result in arguments being evaluated exactly once. A simplistic non-strict reduction scheme will result in arguments being evaluated never, once, twice, or any number of times. A lazy implementation evaluates arguments at most once.

The degree of sharing achieved can be anything from none with call-by-name evaluation to optimal with optimal evaluation. Figure 3.5 shows the degrees of sharing which will be examined, and the partial ordering that exists between them. This identification of such a partial ordering is original.

The use of the word *lazy* in the names of some of these degrees of sharing is misleading as this sharing makes just as much sense in a strict language. The issues of avoiding repeating computations (maintaining sharing) and not evaluating unneeded computations are orthogonal. The strict version of the degrees of sharing have been mostly unnamed (and the existing names cannot be used as many of them include the term lazy).

Call-by-value is the strict equivalent of call-by-need, it has the same degree of sharing, but not the same termination properties. Call-by-name has the same termination properties as call-by-need, but not the same sharing properties. Conceivably a reduction mechanism could have the termination properties of call-by-value and the sharing properties of call-by-name, but this would be the worst of both worlds.

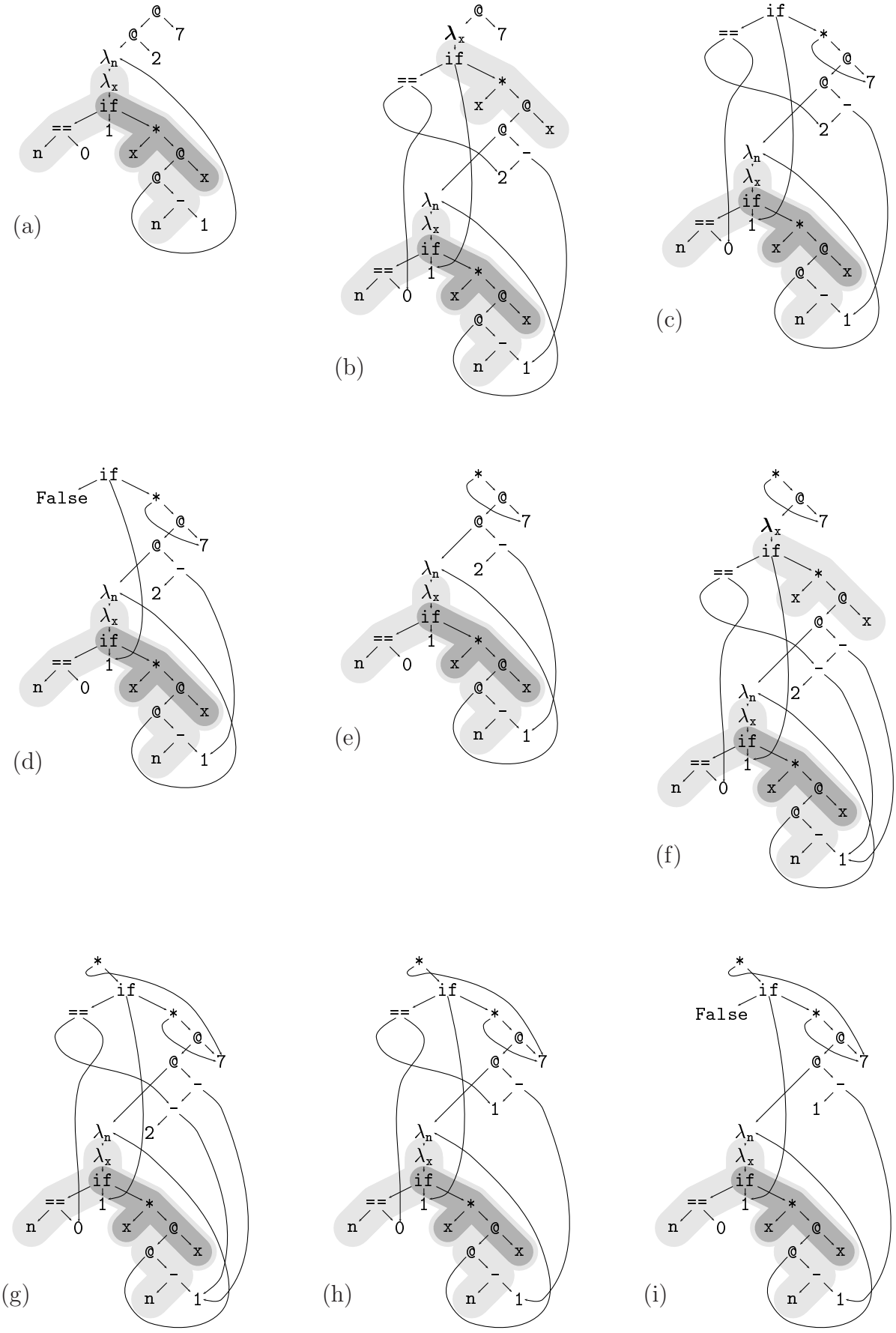



Figure 3.3: WHNF reduction of `power 2 7`.

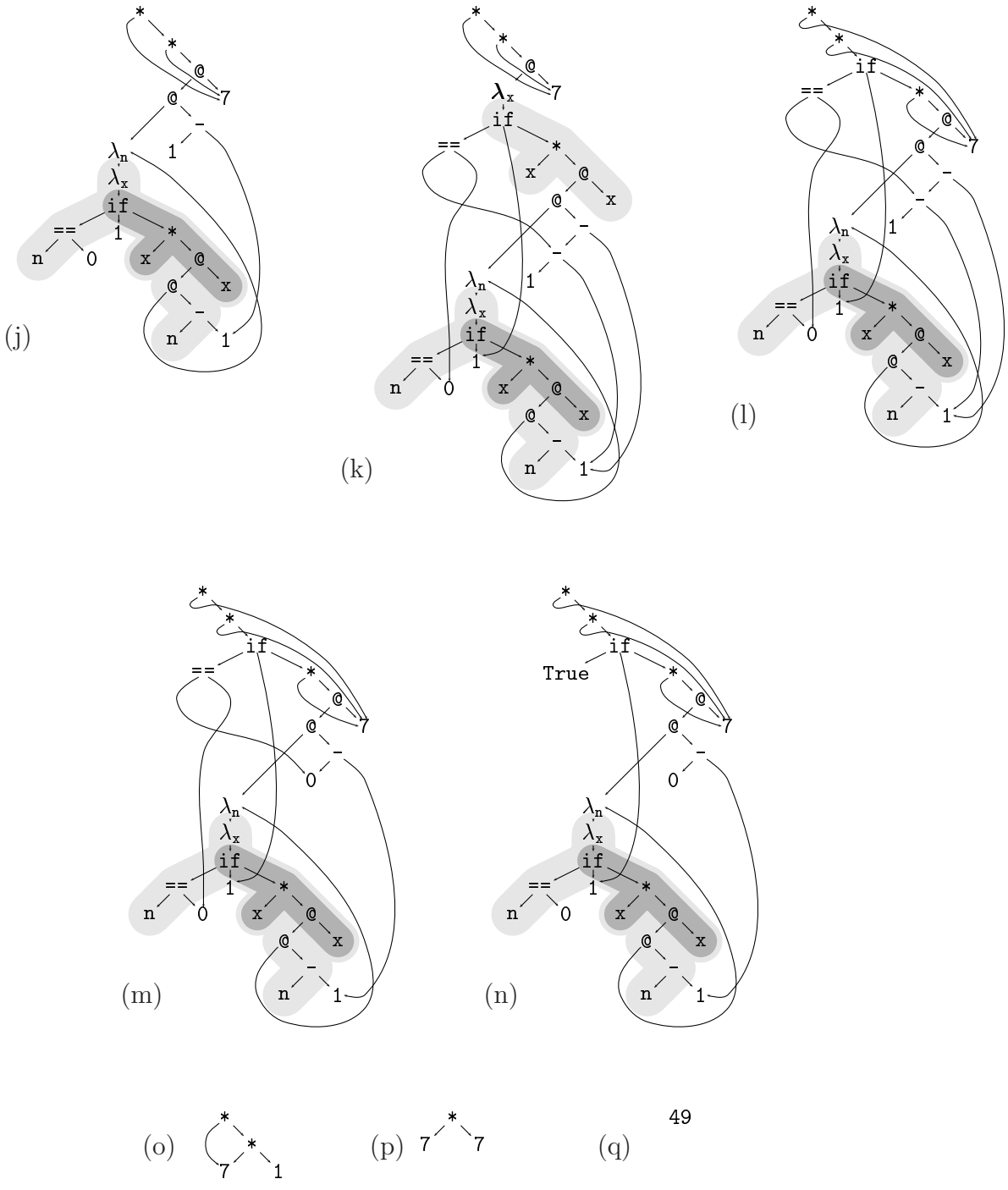



Figure 3.4: WHNF reduction of `power 2 7` continued.

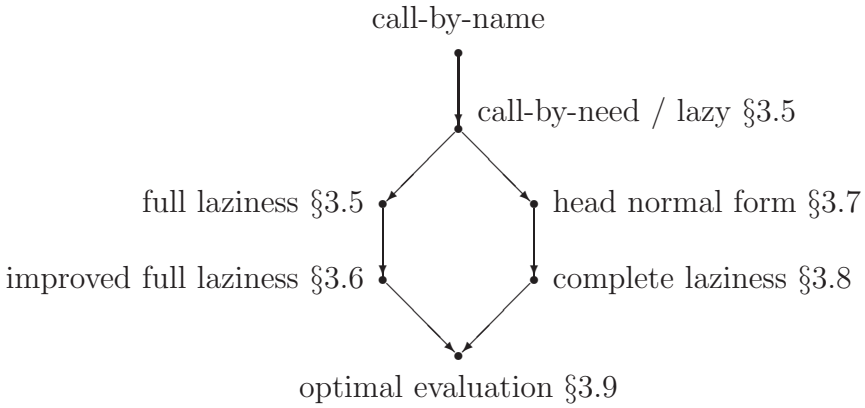



Figure 3.5: Partial ordering of degrees of sharing.

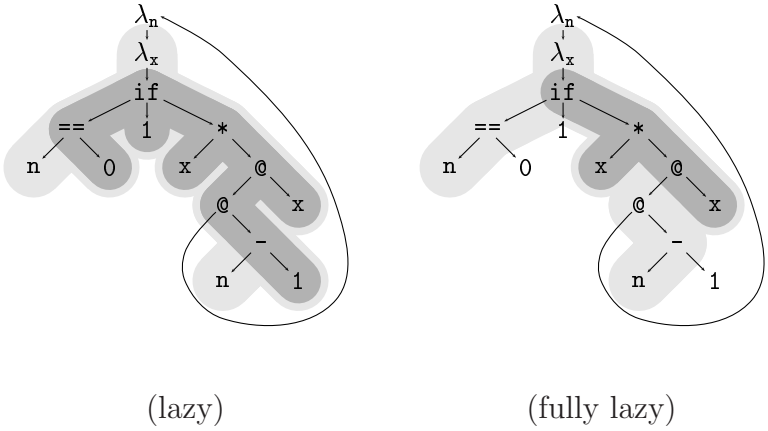


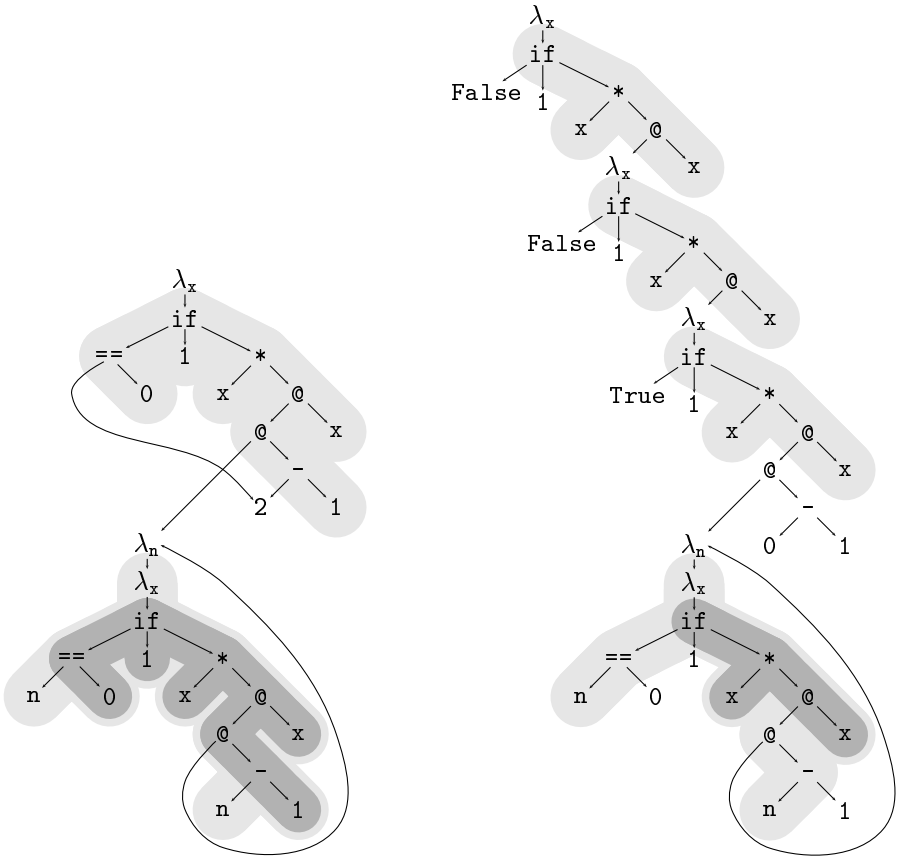

Figure 3.6: Lazy and fully lazy graph representations of `power`.

## 3.5 Laziness and full laziness

There are different ways to translate a syntactic representation of a function to its graphical representation. The way shown in Figure 3.2, is actually the fully lazy way. Contemporary implementations of functional languages however default to lazy, not fully lazy [68, 15, 71, 43]. The difference lies in where the scope of a function is drawn. A fully lazy translation gives every function the minimum scope it can, a lazy translation uses the scope that is present in the syntactic representation of the function. It is possible to achieve full laziness in a lazy language by transforming the syntactic representations of functions so that the scope of every function is minimal [70].

Full laziness ensures that any graph that is syntactically independent of an enclosing lambda is shared and not rebuilt when that lambda is applied. Figure 3.6 illustrates the difference between the lazy and fully lazy versions of `power`, and Figure 3.7 shows the implications of this.

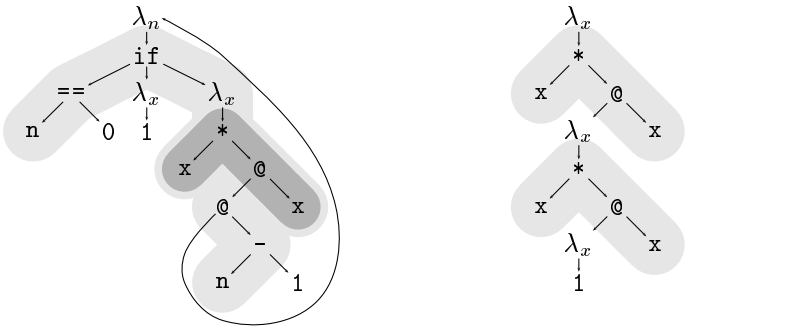
Figure 3.7 illustrates what `power 2` reduces to. For fully lazy reduction this is




```
\x->if 2==0 then 1
     else x*power (2-1) x
where
power n x =
 if n==0 then 1
 else x*power (n-1) x
```

```
\x->if False then 1 else x*
 (\x->if False then 1 else x*
  (\x->if True then 1 else
   power (0-1) x) x) x
where
power n x =
 if n==0 then 1
 else x*power (n-1) x
```

Figure 3.7: Lazy and fully lazy graph representations of `power 2`.




```
power where
power n =
 if n==0 then \x->1
 else \x->x*power (n-1) x
```

```
\x->x*(\x->x*(\x->1) x) x
```

Figure 3.8: Improved fully lazy `power` and `power 2`.

what was shown in Figures 3.3 and 3.4, although it was not shown in a single frame. Figure 3.7 shows the result of *holding onto* the application node applying `power` to `2` in Figure 3.3(a). The lazy version achieves almost no sharing at all. The fully lazy version can be seen to share all the reductions of `n==0`, and `power n`. After the first application of `power 2`, no more instances of `==` or `power` are encountered, thus greater sharing is achieved.

## 3.6 Improved full laziness

The fully lazy reduction of `power` clearly saves repeating work, but it also looks like more work could be saved. The `if` expressions will be repeatedly reduced, and this is undesirable. One way to solve this would be to reduce the `if` expressions where they are. However, this means reducing under lambdas which is costly as it prohibits the use of a fixed-program implementation. Performing reductions within the scope of a lambda would amount to changing the machine-code of a function at run-time. But there is another way to solve the problem: it is possible to float the `if` expression out of the enclosing lambda expression.

Figure 3.8 shows the effects of a source-to-source transformation known as *improved full laziness* [35]. Here the `if` expression has been floated up through the enclosing lambda expression so that it can be evaluated once and for all before the value to be squared is known. This results in two lambda expressions being required, one for the then clause (the terminal case) and one for the else clause (the recursive case). Once `power 2` has been fully applied, no more `if` expressions will be encountered.

It is not possible to eliminate the applications still to be performed if a fixed-program implementation is used. The transformed `power 2` is still made up of functions that were present in the power expression. No reductions have been performed under lambdas hence there are no new representations of function bodies, just an efficient composition of fragments of the original program.

Improved full laziness is the most that can be achieved without evaluating under lambdas. It provides a relatively high degree of sharing. If used with a memo-table to achieve the coincidental sharing achieved by partial evaluation it is equivalent to a first-order partial evaluator [46, Chapter 5].

The specializing effect achieved so far has relied upon the order in which arguments have been supplied. The arguments to `power` are first the power to which to raise something, and then the thing to be raised to that power. Without performing reductions under lambdas this argument ordering is critical as it determines the order in which computations become top-level and reducible.

This argument ordering requirement also carries through to any functions called by a function to be specialized. This can result in a limited amount of code explosion, as it may be necessary to have different versions of functions taking their arguments in different orders. In the power example, if there had been any sharing between the `then` and `else` clauses, this would have to be duplicated, resulting in further code explosion.

## 3.7 Head normal form reduction

Now the issue of what more can be achieved if reductions under lambdas are permitted is explored. It may seem odd to reduce expressions under lambdas. Surely no programmer writes functions which obviously contains expensive reducible expressions. However it is not so much the functions that the programmer writes whose bodies may benefit from reductions, it is the functions that are created dynamically by substitution, whose bodies may contain worthwhile reductions.

When only top-level reductions are performed, the bodies of functions are always copied at one depth less (one grey shade lighter). When reductions under lambdas are performed, it will be necessary to be able to *shift* the graph as it is copied by differing amounts. This occurs when the argument and the lambda expression are at different depths. For example in the application of a function `\x->x:x` to a variable

```
power 2
\x->if 2==0 then 1 else x*power (2-1) x
\x->if False then 1 else x*power (2-1) x
\x->x*power (2-1) x
\x->x*if (2-1)==0 then 1 else x*power ((2-1)-1) x
\x->x*if 1==0 then 1 else x*power (1-1) x
\x->x*if False then 1 else x*power (1-1) x
\x->x*x*power (1-1) x
\x->x*x*if (1-1)==0 then 1 else x*power ((1-1)-1) x
\x->x*x*if 0==0 then 1 else x*power (0-1) x
\x->x*x*if True then 1 else x*power (0-1) x
\x->x*x*1
```

Figure 3.9: HNF reduction sequence for `power 2`.

```
map (\x->power' x 2) [1..10]
where
power' x n =
 if n==0 then 1 else x*power' x (n-1)
```

Figure 3.10: `power'` taking its arguments in a different order.

at depth n, the the copied : should be at depth n. In general all substitutions shift the graph by the difference between the depth of the application node, and the depth of the lambda node plus one. Thus since all top-level reductions take place between application nodes at depth zero and lambda nodes at depth zero, all top-level substitutions are performed with a shift of minus one. This *shift* can be thought of as renaming, the function body is renamed as it is copied, unlike Abadi et al's $\lambda\sigma$-calculus [1] (see §2.9) where it is the argument which is renamed.

Reducing arbitrary expressions under lambdas may risk non-termination, so which reductions are perform under lambdas is important. One solution is to reduce any computations under a lambda that will be performed immediately after applying that lambda. That is lambda expressions are reduced to head normal form just before they are applied.

Using the running example `power 2`, HNF evaluation performs the reduction steps shown in Figure 3.9. Thus HNF reduction results in an even better function than was achieved by improved full laziness. Moreover if the `power` example is rewritten to take its arguments in the more natural order, the same degree of sharing is still achieved. For example with the program in Figure 3.10, the function `\x->power' x 2` will again reduce to `\x->x*x*1`.

The degree of sharing achieved is enough for entire layers of interpretation to be eliminated. For example if an interpreter is written for *Ef* parse trees in *Ef* and this interpreter is used to evaluate the parse tree for `power` rather than run it

```
map square_cube [1,2,3] where
square_cube x = (power' x 2,power' x 3)
power' x n =
 if n==0 then 1 else x*power' x (n-1)
```

Figure 3.11: `square_cube` mapped over `[1,2,3]`.

directly the result is the same: `\x->x*x*1`. (See §6.1 for further discussion on such an interpreter).

Consider a lambda expression that contains reducible expressions, if these expressions will not necessarily be evaluated immediately after application, then it is unsafe to reduce them before copying the body of the lambda expression. This results in copying them before reducing them, thus they will be reduced many times after being copied instead of once before.

A simple example of HNF reduction failing to achieve the desired degree of sharing is illustrated in Figure 3.11. This program computes a list of pairs of the squares and cubes from `1` to `3`. However since the pairing function is non-strict, the terms `power' x 2` and `power' x 3` cannot be reduced beneath the lambda as it is not known that the actual squares and cubes will ever be needed. By the time it is known, the terms may have been substituted resulting in an arbitrary number of suspended computations to perform. In this case performing the reductions would not risk non-termination, but this is not the case in general.

Barendregt, Kennaway, Klop and Sleep studied the reduction of redexes beneath lambdas [20]. They devised *spine strategies* to determine statically before the application of a lambda abstraction is reduced, *some* of the redexes under the lambda which must be reduced in the process of reducing the application to normal form. Determining statically *all* of the redexes that must be reduced is undecidable in general. The spine strategies are a computable approximation.

Reduction to HNF does not require such foresight to know which redexes will be needed, the next redex to reduce is simply determined dynamically.

## 3.8 Complete laziness

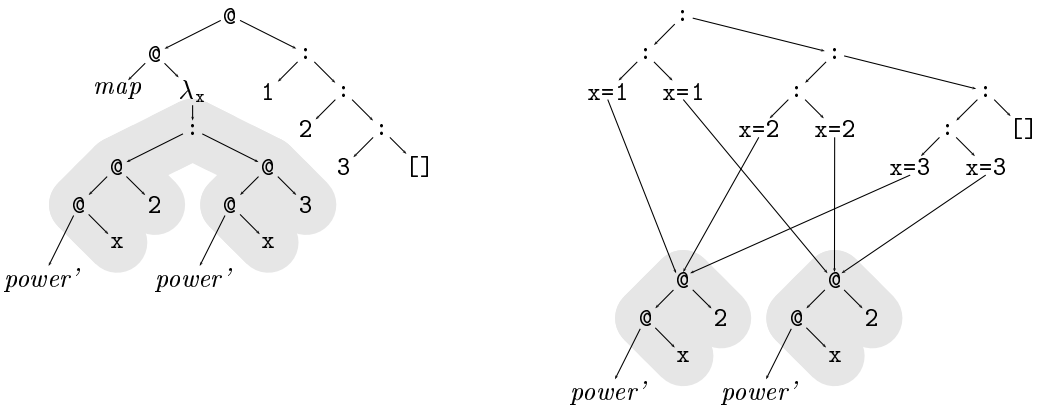
In order to prevent the loss of sharing resulting from substituting into unevaluated graph, a better scheme is needed in order to decide which reductions to perform. Without impossible foresight it is not possible to know in general which reductions within the body of a function should be reduced before that functions body is

copied. This is almost the mirror situation of not knowing whether to reduce the argument to a function before the argument is copied. Wadsworth [82] solved that by postponing the decision (see §2.3). Much the same solution can be used here, specifically the decision regarding which redexes within the body of a function to reduce is postponed until after the application.

This is achieved by *delayed substitution*. By delaying substituting into a graph until it is known to be needed, the graph can safely be evaluated before it is substituted, without risking non-termination.

Special *substitution nodes* are used in the graph, to delay the substitution. Figure 3.12 shows some substitution nodes in action. In the graph representation, variable names and not depths are still being shown. The substitutions also know by how much they should shift the depth of the nodes which they copy, although the shift associated with each substitution is not shown in the diagrams in this section.

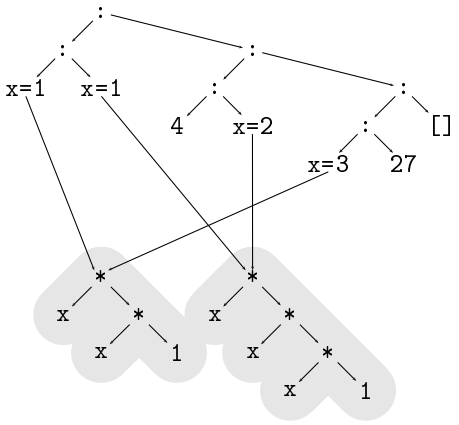
A beta-reduction is now performed by simply replacing the application node (`@`) by a substitution node (`=`). This substitution node is then pushed down when evaluated. This results in a multiplication of substitution nodes all originating from the same beta-reduction as the substitution is forced through nodes with an arity of two or greater. Substitution nodes are called *related* if they originated from the same beta-reduction. When the substitutions meet a node at a depth shallower than the depth of the variable they wish to bind, they know that they have left the scope of the function whose application created them. At this point they *die-out*, that is they replace themselves with an indirection. These substitution nodes remember the amount by which they should shift the graph they copy. This shift is just the same as for HNF reduction, just that now the copying is being done node by node. The depth of a substitution may change as the depth of the nodes it is substituting change. The nodes it substitutes may become deeper following the substitution of a nested lambda, and shallower upon leaving the scope of that nested lambda. The depth of the substitution is always the depth at which it expects to be creating new nodes. That is the depth it expects the node it is pointing to be at, plus the shift of the substitution. Since the substitution will not know the actual depth of the node it is pointing to until it actually comes to substitute it, the depth of the substitution may be deeper than necessary, this just means the substitution is on the edge of the scope of a function, that is, it has finished copying all the nodes for that function, but doesn't quite realise it yet.



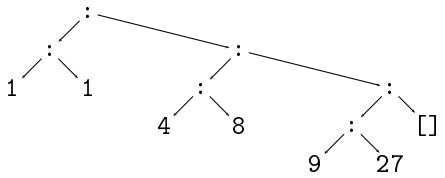

(a)




(b)




(c)




(d)

Figure 3.12: `square_cube` mapped over `[1,2,3]`. Nodes in italics such as *map* and *power'*, are just abbreviations for their full graph representations.

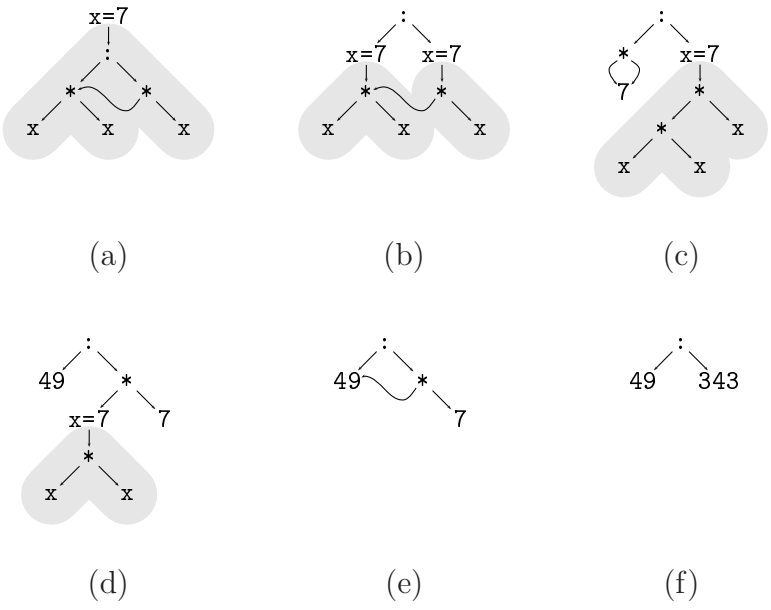
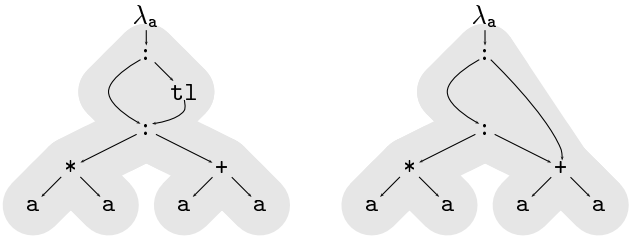



Figure 3.13: Substituting into a shared graph.

Figure 3.12 shows the result of forcing the evaluation of the `square_cube` example to increasing extents. So far so good. Problems are encountered however when delayed substitution into a shared graph is attempted. There is the possibility that two related delayed substitutions could each try to substitute into the same piece of graph. It must be ensured that such a graph is not substituted into more than once by substitutions originating from the same beta-reduction.

One solution is to create a new *memo-table* each time a beta-reduction is performed. Each related delayed substitution can then check and update this memo-table whenever it performs a substitution. Figure 3.13 shows the effect of related substitutions using a memo-table. In Figure 3.13(d) when a substitution node is about to substitute into the `x*x` graph, it checks in the memo-table and discovers that a related substitution node has already substituted into this graph. Instead of substituting into the graph again, the substitution node just replaces itself with an indirection pointing to the result of the previous substitution, which, in this case, just happens to have already been reduced.

It is important to understand that this is *not* the conventional use of memo functions [63]. If a function is applied to the same value twice the computations will *not* be shared. Substitutions are only shared when they originated from the same beta-reduction. Delayed substitutions remember which nodes they have been substituted into; but nodes do not remember which values have been substituted into them.




(a) (b)

Figure 3.14: Shared graph changing.

Using memo-tables may appear to be an unnecessarily powerful technique as memo-tables are being needlessly and wastefully checked and updated for nodes that are not shared. However knowing which nodes are shared is not as straightforward as it may first appear. As graph is evaluated before substituting into it, which nodes are shared changes. Consider the graph in Figure 3.14. When substitution starts, the `:` node is shared, but after evaluating the `tl` node, it is the `+` node that is shared. If details of which nodes are shared were to be maintained by using reference counting, and memo-tables only checked and updated for shared nodes, sharing would still be lost. A node could be unshared when first substituted and only become shared subsequently. This happens in Figure 3.14, if the head of the upper `:` node is substituted first. The `+` node will be substituted before it is known that it will become shared.

One of the stated attractions in using a non-strict language for specializing was the reduced need for the memoization in conventional partial evaluation. And yet here a new form of memoization is being used instead. There is hope however that with further study, the proposed use of memo-tables can be refined to be less of a sledgehammer-like approach.

This form of memoing results in a degree of sharing known as *complete laziness* [36]. Whereas full laziness ensures that *any* expressions within a lambda expression that are *syntactically* independent of the lambda-variable are shared across all bindings of that variable, complete laziness ensures that *reducible* expressions within a lambda expression that are *computationally* independent of the lambda-variable are shared across all bindings of that variable. For functions taking multiple arguments this means that all work computationally dependent only on the early arguments is shared for all applications to later arguments.

Although the sharing achieved by complete laziness is more impressive than that

achieved by full laziness, complete laziness does not achieve a strictly greater degree of sharing. An example where full laziness maintains sharing and complete laziness loses it is:

```
f 1 where
    f x = g 2 + g 3 where
                    g y = x+x
```

Complete laziness will ensure `x+x` is reduced as far as possible before substituting each of `2` and `3` through it. But in the process of substituting both `2` and `3` through the term `x+x`, the term is duplicated. However full laziness changes the scope of `g` so that any substitutions arising as a result of applying `g` will immediately die-out before they get a chance to copy `x+x` A degree of sharing that maintained all the sharing that both full and complete laziness maintains could easily be obtained, simply by performing the cheap static full laziness transformation before using a completely lazy reduction technique. However unless expensive *dynamic* fully lazy reduction is used this would result in the base language in a tower of interpreters having features that a simply implemented interpreter on top of it would not have. And as stated previously additional features are intended to be implemented on top of a primitive base language. Static full laziness only ensures that syntactically free expressions are not copied if only top-level reductions are performed.

The delayed substitution technique can be seen as orthogonal to Wadsworth's delayed argument evaluation technique. Just as reduction mechanisms can be classified with respect to their argument evaluation technique into the categories: call-by-name, call-by-value and call-by-need. A reduction mechanism can be classified with respect to its function body evaluation technique into one of the categories: substitute-by-name, substitute-by-value, and substitute-by-need. These terms are new, but the techniques they describe are not, the use of these names helps demonstrate the similarity and orthogonality that exists between the issues of argument evaluation and function body evaluation.

Conventional functional programming languages would be classified as substitute-by-name, as they freely copy function bodies even though redexes may exist within them. Substitute-by-value ensures that all redexes within the function body are reduced before the function is applied. This is the same as reducing functions to normal-form. Substitute-by-value risks non-termination by reducing redexes that may never be needed, in an analogous way to call-by-value. Substitute-by-value achieves a greater degree of sharing than substitute-by-name, analogous to call-by-

value achieving a greater degree of sharing than call-by-name. Substitute-by-need has the sharing properties of substitute-by-value, and the termination properties of substitute-by-name, again analogously.

In this classification, a completely lazy evaluator is call-by-need and substitute-by-need.

A call-by-value/substitute-by-value reduction mechanism raises an interesting issue. Typically a term is said to be a value if it is in WHNF. If no redexes beneath lambdas are reduced, then the copying of terms in WHNF will not be responsible for the duplication of work. However if a term in WHNF is subsequently to be reduced further, prior copying of that term will duplicate work. If a call-by-value/substitute-by-value evaluator is to be implementable without the use of graphs, then the *value* must refer to a term in normal-form not just WHNF.

The way in which substitute-by-value risks non-termination is far more serious than the way in which call-by-value does. As will be explained later (see §6.2), any typical recursive function contains an infinite number of redexes, so cannot be reduced to normal-form. Call-by-value/substitute-by-value achieves the same degree of sharing as complete laziness, but would have very poor termination properties.

Field [28] studied reduction mechanisms which employ a form of delayed substitution so as to make possible the sharing of the reduction of redexes in function bodies. However Field didn't use memo-tables or anything equivalent, so although his framework could support call-by-value/substitute-by-need and call-by-name/substitute-by-need, it could not support call-by-need/substitute-by-need.

In a reduction system, such as Field's, if sharing is only introduced by function application, i.e. there is no distinct concept of a let binding, then memo-tables are only required if both call-by-need and substitute-by-need are required. This introduction of a more advanced sharing mechanism to maintain the sharing and termination properties of call-by-need and the sharing and termination properties of substitute-by-need can be seen as analogous to the Wadsworth's introduction of graph sharing in order to maintain the sharing properties of call-by-value and the termination properties of call-by-name.

Complete laziness passes the tower of interpreters test (§6.1). Complete laziness could be said to achieve *one dimensional sharing*. The following example illustrates where what could be called *multi-dimensional sharing* is required:

```
map (\x-> map (\y-> f (x,y)) [0..]) [0..]
```

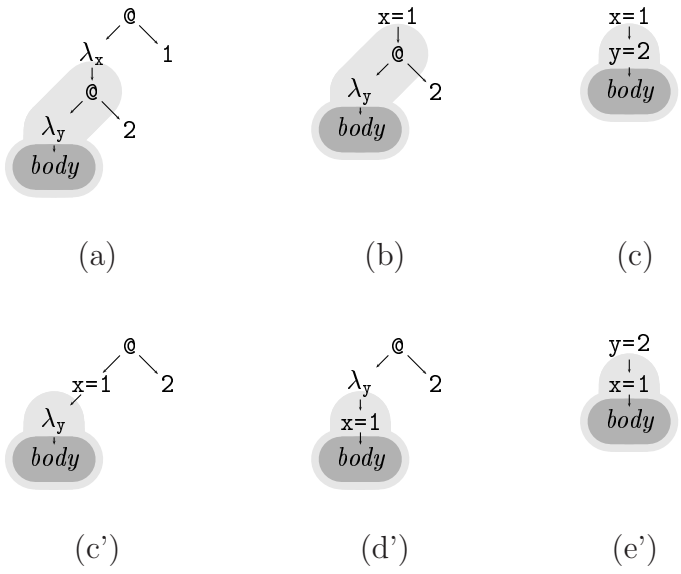



Figure 3.15: A choice of reduction order.

which reduces to

```
[ [f(0,0), f(0,1), f(0,2), ...],
  [f(1,0), f(1,1), f(1,2), ...],
  [f(2,0), f(2,1), f(2,2), ...],
  :...
  :...
]
```

This computes `f` applied to every pair of natural numbers. Depending on the reduction order used, the function `f` can be specialized with respect either to `x` or to `y`, and then repeatedly applied to the other argument. Which argument is substituted first will determine for which argument `f` is specialized. The best choice will depend on both the nature of `f` and the way in which the result is used.

A substitution in a completely lazy evaluator always ensures any graph is reduced before substituting into it. If the graph consists of a substitution then it pushes this down so that it can find some graph for it to substitute into. The order of substitutions never changes once formed, they just ripple through the graph.

Figure 3.15, shows how substitutions can be formed in different orders. Complete laziness will always use the reduction order shown in sub-figures (a),(b) and (c), but the reduction order (a),(b),(c'),(d') and (e') could be used instead. In Figure 3.15(a) a choice of reductions exists. If the beta-reduction which will bind `x` to `1` is reduced first, as shown in Figure 3.15(b), then there is a choice of either pushing the `x=1` substitution after (Figure 3.15(c)) or before (Figure 3.15(c')) reducing the beta-reduction which will bind `y` to `2`.

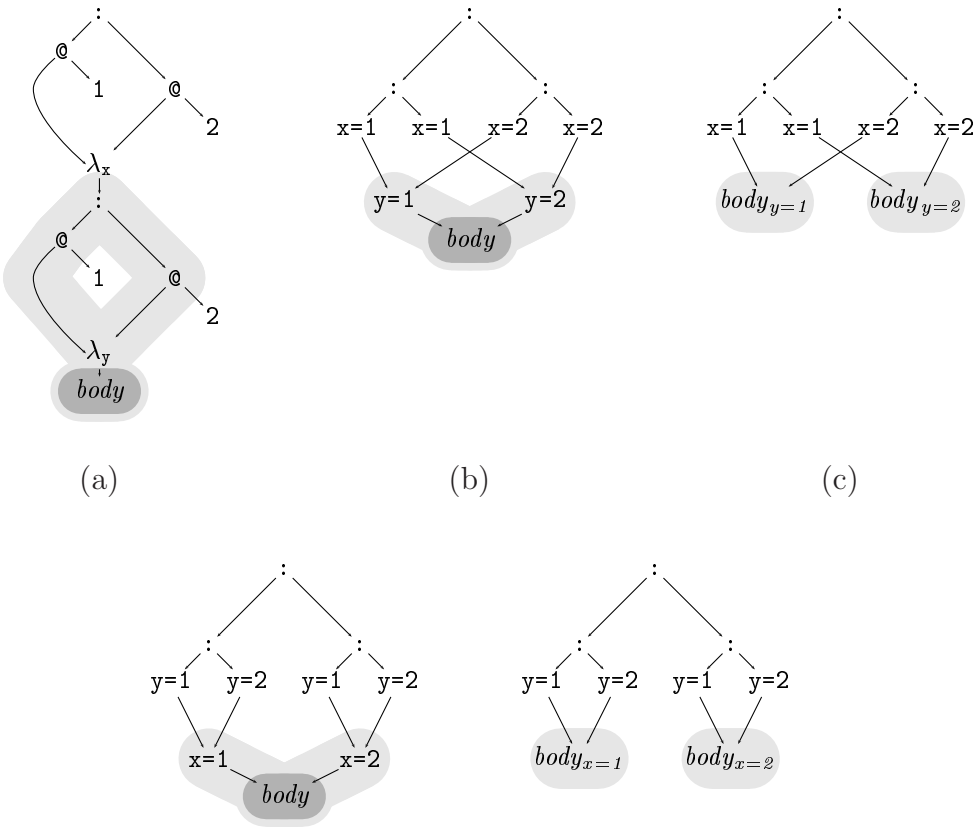



Figure 3.16: Alternative choice of reduction orders, unavoidably losing sharing.

For a program with multiple partial applications to a nested function, choosing one reduction order instead of another could lead to arbitrary loss of sharing as Figure 3.16 shows. In sub-figure (c) any computations dependent only on `x` fail to be shared. In sub-figure (c') the mirror situation exists with computations dependent only on `y` failing to be shared. If the graph that is computationally dependent only on `x` does not syntactically contain `y` then full laziness may come to the rescue. But in the general case, using the reductions schemes so far described, sharing is inevitably lost one way or the other.

An example of a piece of graph that contains reductions that are syntactically dependent on both of `x` and `y`, but not computationally dependent on both is:

```
power' x y : power' y x
```

Here `power' x y` is worth specializing to a value of `y` and `power' y x` is worth specializing to a value of `x`, but with the reduction techniques described so far, it is not possible to benefit from both these specializations.

The two reduction sequences in Figure 3.15 (and likewise for Figure 3.16) are mutually exclusive. Lazy evaluation and delayed substitutions effectively enable a reduction mechanism to change its mind about *whether* a redex is reduced before a beta-reduction. The issue here is not *whether* to reduce a redex, the issue is *which* of two mutually exclusive redexes should be reduced.

## 3.9 Optimal evaluation

The key to achieving complete laziness was to ensure that substitutions are only performed into graph that has been reduced as far as possible. But even this can still result in sharing being lost. Substitutions may be performed into graph that is computationally dependent on some other substitution, thus not aiding further reduction and copying graph needlessly. This is not a problem if a degree of sharing equivalent to partial evaluation is required.

To achieve optimal evaluation, substitution nodes which would substitute through graph that is computationally dependent on some other substitution must not be reduced. A mechanism capable of swapping substitutions is needed, so as to rearrange the order of substitutions such that only substitutions which will enable further reduction are performed.

Whether the techniques developed so far can be generalized to optimal evaluation is not clear. A previous version of this chapter ended with a not very convincing reason of why they could not. In a series of failed attempts to find a convincing reason, a seemingly correct optimal evaluator has been implemented.

The results chapter provides lots of evidence of the correctness of this implementation in terms of producing the correct results in the optimal number of beta-reductions, and no programs have been found that suggest the implementation is not correct.

Positive as the evidence is, it doesn't help greatly in finding either the originally sought after explanation of why it should not be possible to generalize complete-laziness to optimal evaluation; or an explanation of the correctness of this implementation. Nevertheless the techniques developed will be explained.

Firstly it must be determined which variables a node is computationally dependent on, that is, which variables are *blocking* further reduction. Every node is either reducible, fully reduced, or is prevented from further reduction by the presence of variables. For each *blocked* node a single variable can be identified as blocking further reduction. The nodes that can be blocked are application nodes, substitution nodes, primitive nodes such as `+` and variables. A variable node is blocked on itself. An application node is blocked on the same variable that the node reached by the function part of the application is blocked on. A primitive node is blocked on the same variable as the node corresponding to the first strict argument which is blocked. So taking for example the term $a$+$b$, if reduction of $a$ is blocked, then $a$+$b$ is blocked on whatever $a$ is blocked on, otherwise if $b$ is blocked then $a$+$b$ is blocked on whatever $b$ is blocked on, otherwise $a$+$b$ is not blocked and can be reduced. A substitution node is blocked on the *renamed* version of the variable the node the substitution is to substitute is blocked on.

When a substitution meets a node, it first tries to reduce that node, and then checks if that node is fully reduced. Nodes such as `(:)`, $\lambda$ and constants, are fully reduced, and can be substituted without loss of sharing. Other nodes such as `(+)` and `@` and variables, if they cannot be reduced must be blocked on something. If the variable the substitution is trying to bind, corresponds to the variable the node it is trying to substitute is blocked on, then the substitution goes ahead and substitutes the node. Otherwise the substitution refuses to budge and tags itself as being blocked.

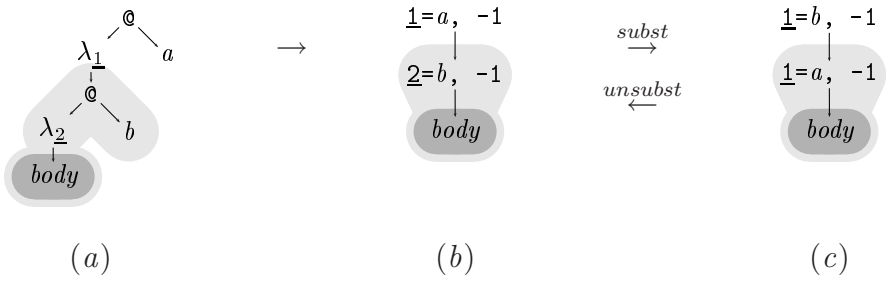



Figure 3.17: Substituting and unsubstituting substitutions.

Once a the variable on which a node is blocked has been determined, the node is tagged with this variable. The effect of this is that a variable which is blocking further reduction, propagates this fact up towards the root of the graph. When the substitution binding this variable (or a renamed version of the variable if the blocked tags have propagated through a substitution node), is found, a substitution is performed.

Substitution is performed just the same way as was the case for the completely lazy evaluator, except now it is possible a substitution node may be substituted and even unsubstituted as the required substitution swaps with any substitutions between its original position as the variable blocking further reduction.

Figure 3.17 shows a concrete example of substitutions being swapped. The use of variable names will now be stopped and numbers used instead. The binding-numbers on the lambdas and variables (no variables are actually shown, but if they were) are strictly redundant, and can be inferred from the depth of the graph in which they occur. The binding-numbers in the substitutions are not redundant and cannot be inferred from the depth of the graph. The shift of the substitutions is now also shown, this occurs after a comma.

Substitutions are swapped by either substituting them or unsubstituting them. In the following text, *upper* substitution refers to the substitution which points to the *lower* substitution. To substitute substitutions, the shift of the upper substitution is used to rename the binding-variable of the lower substitution. To unsubstitute substitutions, the shift of the lower substitution is used to *un*rename the binding-variable of the upper substitution. The general case is shown in Figure 3.18 without depth shading.

**Aside** The substitution of substitution nodes depicted in Figures 3.17 and 3.18 do not depict the substitution of the argument of a substitution node. Specif-

ically with reference to Figure 3.17, the upper substitution is rebuilt below the lower substitution and is shown ready to substitute the body of the lower substitution, but the upper substitution is not shown ready to substitute the argument $b$. This is not an over-simplification in the figure.

If the argument were substituted, trying to unsubstitute it later would be a daunting and very complex task, as the substitution may have started propagating through the argument $b$. At least this was the reason originally given as to why complete laziness could not be generalized to optimal evaluation.

There is in fact no possibility of the upper substitution propagating through the argument $b$, while the substitution nodes are in their swapped arrangement, as the argument $b$ will only be evaluated once it has been bound to some variable. But the variable to which it will become bound is beneath what was the upper substitution. It is inevitable that the substitutions will have to be swapped back again before the argument $b$ will ever be evaluated. There is no other way for what was the upper substitution ($\underline{1} = \mathtt{a}$) to escape from its position between what was the lower substitution and the variable $\underline{2}$ that $b$ will be bound to. It cannot just substitute through the variable $\underline{2}$, renaming it to $\underline{1}$, as substitutions will not substitute nodes which are blocked on a different variable from the one the substitution binds.

This insight means that there is no need to rebuild the upper substitution ($\underline{1} = \mathtt{a}$) ready to substitute the argument $b$ in the first place.

□

The depths of the substitutions are handled the same way regardless of whether substitutions are swapped by substituting or unsubstituting. As before, the depth of a substitution is the expected depth of the node it points to, plus the shift of the substitution. If *depth* is the depth of the upper substitution, and *depth'* is the depth of the lower substitution, Then the depth the lower substitution expects the node it points to be at is $depth' - shift'$. When the substitutions are rebuilt in the swapped order, the new lower substitution will have depth $(depth' - shift') + shift$. The depth of the new upper substitution, is the depth of the new lower substitution plus *shift'*, which is $depth' + shift$.

No explanation has yet been given of how it is known whether two substitutions should be swapped by substitution or unsubstitution. Actually not all substitutions

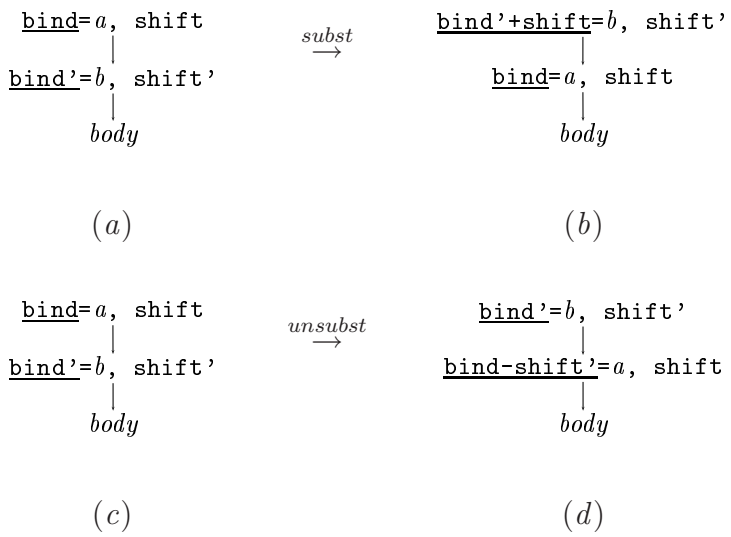
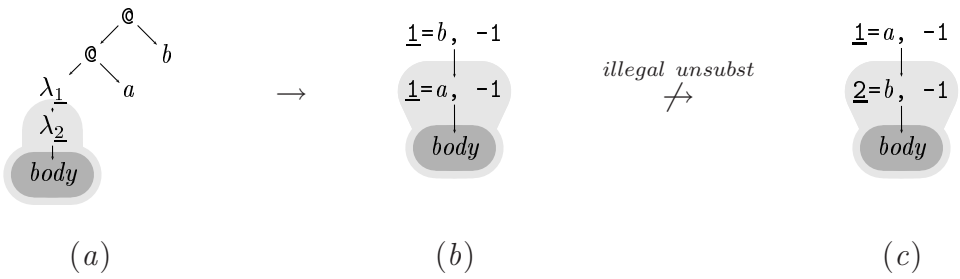



Figure 3.18: Substituting and unsubstituting substitutions in general.

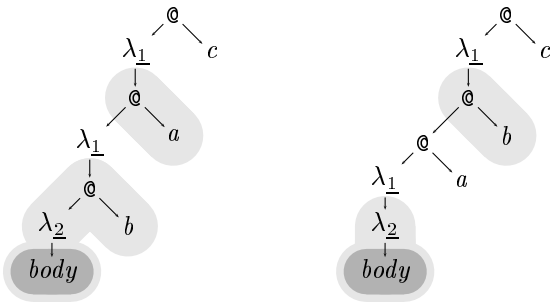
can be swapped. First it should be explained that it is only meaningful to swap substitutions when they could have been formed in the swapped order in the first place. This occurs when one substitution is formed within the scope of another. This ensures that the nested substitution will be binding a deeper variable, than the substitution whose scope it is formed in. This difference in binding depths is used to determine if an upper substitution can substitute the lower. If the variable the upper substitution binds is shallower than the variable the lower substitution binds, then the upper substitution may substitute the lower. If the converse is true then it may be possible for the lower substitution to be unsubstituted from the upper substitution, but not necessarily. Substitution nodes should only be swapped by unsubstitution, if they have previously been swapped by substitution. However comparing the depths of the substitutions binding variables is insufficient to determine if this is the case.

An example showing an unsubstitution which must be prevented is shown in Figure 3.19. There is no choice in the order in which the substitutions are formed, substitution $\underline{1} = \mathtt{a}$ must be formed first and substituted through the next lambda before the substitution $\underline{1} = \mathtt{b}$ can be formed. The result looks identical in terms of nodes depths, bound variables depths and substitutions shifts to Figure 3.17(c), however this case was not the result of substitution nodes being swapped by substitution previously. In this case if an unsubstitution was performed, all that would happen is that the substitution $\underline{1} = a$ would try to substitute through $b$, but since $b$ is at depth 0, the substitution would immediately die-out. However had $b$ not




(a) (b) (c)

Figure 3.19: Curried applications and substitution swapping.




(a) (b)

Figure 3.20: Potentially mixed variables.

been at depth 0, had it been within the scope of another lambda expression, then the variables would get mixed up. Figure 3.20 shows such an example. Each of the two sub-figures can reduce to the same sequence of substitutions, in the first case the substitutions binding $a$ and $b$ may be swapped by substitution and unsubstitution, in the second swapping the substitutions binding $a$ and $b$ would result in the substitution of $\underline{1} = a$ through $b$ instead of $\underline{1} = c$ through $b$.

The two cases can be distinguished by having substitution nodes keep a record of which substitutions have substituted them. To do this, substitution nodes are distinguished by an identifier which all related substitution nodes have in common. This identifier could, for example, be an integer numbering the beta-reduction from which all related substitutions originated.

When it is not possible to swap two substitutions, and yet the graph underneath is blocked and requires the upper substitution, the upper substitutions forces the lower substitution through, performing both substitutions in a single step, updating only the node containing the upper substitution, the lower node remaining unchanged. This doesn't lose sharing as this only occurs as a result of a curried function application where for any given second argument, the first argument is

determined.

Once the substitution that binds the variable that a graph is blocked on has been determined, the substitution must make its way to that variable, swapping with substitutions where it can, and forcing the ones it cannot to go along with it. Thus a substitution may build up a wave of substitutions before it. When the sequence of substitutions which comprise this wave meets a substitution, it must determine whether the entire sequence can make it through that substitution. If it cannot then this substitution must be added to the wave of substitutions also.

The next chapter gives a formal description of these reduction rules.

## 3.10 Summary

In this chapter it was shown how different degrees of sharing maintain and lose sharing. The terms substitute-by-name/value/need were introduced and demonstrated to be orthogonal to and analogous to call-by-name/value/need in a number of interesting ways. A new sharing mechanism based on memo-tables, was introduced so as to achieve the benefits of both call-by-need and substitute-by-need. Complete laziness was explained in terms of the two sharing mechanisms: graph sharing, and memo-tables. A third sharing mechanism: substitution swapping, was introduced so as to generalize complete-laziness to optimal evaluation.

# Chapter 4

# Reduction Rules

This chapter presents a more rigorous treatment of the techniques introduced in the previous chapter so as to provide a link between the general ideas and the specific implementations presented in the next chapter.

To define the reduction rules for completely lazy evaluation and optimal evaluation a new notation is devised (§4.1). This new notation, called cyclic scoped reverse de Bruijn notation, is defined and used both to assist in defining reduction rules (§4.5,§4.7) and to present examples of these reduction rules in action (§4.6,§4.8). Issues relating to indirections and blackholes (§4.2), memo-tables (§4.3) and blocked reductions (§4.4) are discussed.

## 4.1 Cyclic scoped reverse de Bruijn notation

During completely lazy and optimal evaluation, the state of the evaluator can be described by a triple $(S, H, M)$: the stack $S$ is list of addresses, the heap $H$ maps addresses onto terms or addresses and describes a graph with indirection nodes; the memo-table $M$ associates each $\beta$-reduction with a memo-table mapping addresses to addresses. Terms are tagged with *depth* and *blocked* tags.

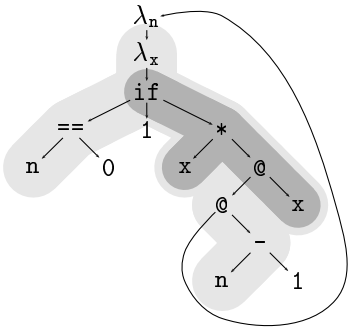
In order to describe the reduction rules concisely, the notation defined in Figure 4.1 is used to specify terms in the heap. The aim is to describe graph reduction rules in the same way as term reduction rules.

For a graph in the heap to correspond to a $\lambda$-calculus expression, the following conditions must be met:

1. if the same address is used in more than one place, then any term or indirection associated with the address should be the same in each case,

| | | | |
|---|---|---|---|
| $\mathbb{N}_1$ | ::= | 1 \| 2 \| 3 ... | |
| $\mathbb{N}$ | ::= | 0 \| 1 \| 2 \| 3 ... | |
| $\mathbb{Z}$ | ::= | ... `-3` \| `-2` \| `-1` \| `0` \| `1` \| `2` \| `3` ... | |
| *Node* | ::= | $\overset{addr}{\underset{depth}{}}$`(` *Term* `)`$^{blocked}$ | (term) |
| | \| | $^{addr}$`(` *Node* `)` | (indirection) |
| | \| | $^{addr}$`( )` | (reference) |
| *Addr* | ::= | $\mathbb{N}$ | |
| *Depth* | ::= | $\mathbb{N}$ | |
| *Blocked* | ::= | ✓ \| ✗ \| $\mathbb{N}_1$ | |
| *Term* | ::= | `@` *Node Node* | (application) |
| | \| | $\lambda$ *Node* | (abstraction) |
| | \| | _ | (variable) |
| | \| | : *Node Node* | (pair) |
| | \| | = *Body Bind Arg Shift Fam SubstBy* | (substitution) |
| | \| | # *Atomic* | |
| | \| | $\Delta_n$ $Primitive_n$ $Node_1$ ... $Node_n$ | |
| *Body* | ::= | *Node* | |
| *Bind* | ::= | $\mathbb{N}_1$ | |
| *Arg* | ::= | *Node* | |
| *Shift* | ::= | $-1$ \| $\mathbb{N}$ | |
| *Fam* | ::= | $\mathbb{N}$ | |
| *SubstBy* | ::= | `[`$\mathbb{N}, \mathbb{N}, ..., \mathbb{N}$`]` | |
| $Primitive_1$ | ::= | `head` \| `tail` ... | |
| $Primitive_2$ | ::= | `==` \| `+` \| `-` \| `*` ... | |
| $Primitive_3$ | ::= | `if` | |
| *Atomic* | ::= | `[]` \| `True` \| `False` \| $\mathbb{Z}$ \| *String* | |

Figure 4.1: Grammar for cyclic scoped reverse de Bruijn notation




```
power where
power n = \x -> if n_eq_0 then 1 else x * pwr_n_1 x
          where n_eq_0 = n==0
                pwr_n_1 = power (n-1)
```

$$
\begin{array}{l}
{}^{1}_{0}(\lambda\ {}^{2}_{1}(\lambda\ {}^{3}_{2}(\Delta\ \texttt{if}\ {}^{4}_{1}(\Delta\ \texttt{==}\ {}^{5}_{1}(\_)\ {}^{6}_{0}(\#\ \texttt{0}))\\
\qquad\qquad {}^{7}_{0}(\#\ \texttt{1})\\
\qquad\qquad {}^{8}_{2}(\Delta\ \texttt{*}\ \ {}^{9}_{2}(\_)\\
\qquad\qquad\qquad {}^{10}_{2}(\texttt{@}\ \ {}^{11}_{1}(\texttt{@}\ \ {}^{1}(\ )\\
\qquad\qquad\qquad\qquad\qquad {}^{12}_{1}(\Delta\ \texttt{-}\ {}^{13}_{1}(\_)\ {}^{14}_{2}(\#\ \texttt{1}))\\
\qquad\qquad\qquad\qquad {}^{15}_{2}(\_)))))
\end{array}
$$

Figure 4.2: The fully lazy `power` function written in nested cyclic scoped reverse de Bruijn notation.

2. nodes in the graph described by the heap may only point to nodes at an equal or shallower depth, with the exception of $\lambda$ nodes which may point to nodes at a depth at most one greater than their own,

3. no substitution nodes should exist and

4. for all sets of nodes where all nodes are at a depth greater than some $n$, and where the set cannot be partitioned into unconnected subsets, there is a unique $\lambda$ node at depth $n$ which points to a node in the set.

The last requirement is equivalent to Wadsworth's admissible graph criteria [82, 3] that specifies that each variable should be bound by a unique $\lambda$ node.

When using the notation to describe a graph with sharing, a reference can be used to avoid duplication. Indeed this must be done if graphs with cycles in are to be described. References do not actually exist in the heap, they are a notational convenience used to express sharing and cycles.

To give an example of the notation in use, the fully lazy `power` function from

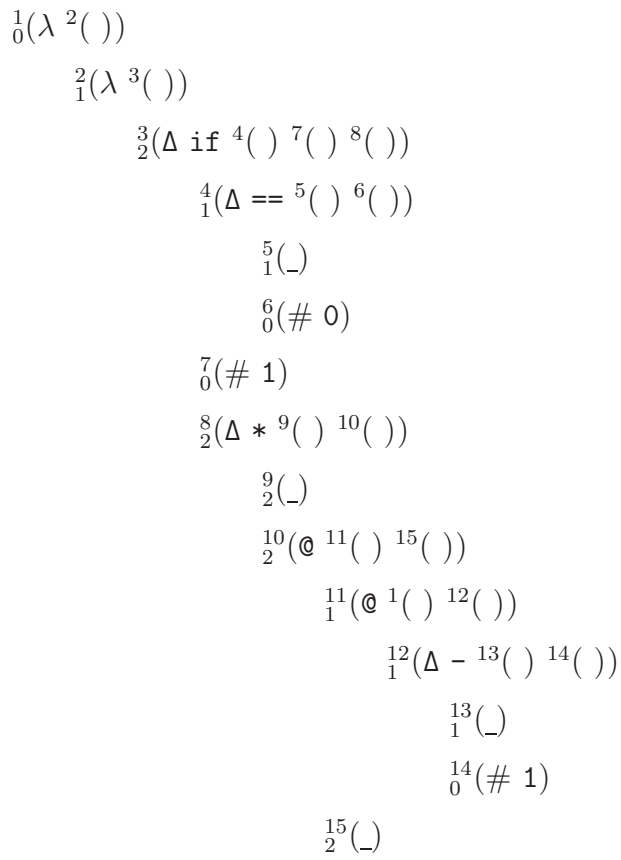



Figure 4.3: Unnested notation.

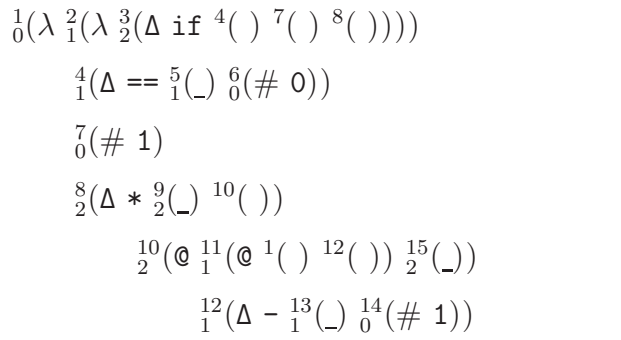



Figure 4.4: Semi-nested notation.

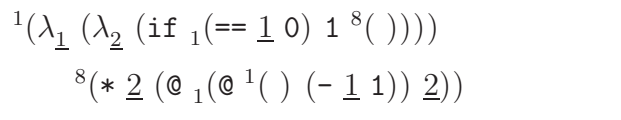



Figure 4.5: Condensed semi-nested notation.

Figure 3.2 is shown in Figure 4.2 using nested cyclic scoped reverse de Bruijn notation. An alternative to nesting terms within terms, is to write each term on a new line as shown in Figure 4.3. These two examples of the notation both represent exactly the same graph. The nested notation has the advantage of showing a term in context and saves the reader from matching up references with terms. The unnested notation has the advantage that each term can easily be identified without the reader having to pair up brackets. In practice a combination of the two notations is preferable, as shown in Figure 4.4. This semi-nested notation will be used in examples. The indentations have no formal significance, they serve only as a visual aid.

*Blocked* tags will be useful for defining the reduction rules later, but for the time-being they are not shown. Even so the formulation of `power` appears cluttered. Other tags may also be omitted when they are insignificant. For instance showing the address of a node which isn't shared may be superfluous. The depth and address of atomic values are of little interest, and for literal values the # is superfluous so the atomic value ${}^{7}_{0}(\#\ \texttt{1})$ may simply be denoted by `1`. In practice all atomic values are at depth zero: atomic values resulting from instantiation are created at depth zero; atomic results of primitive operations are created at depth zero. Even if atomic values did exist within the scope of a function, the resulting loss of sharing caused by copying this atomic value would be of little consequence. The address of a variable is also of little interest and the variable ${}^{5}_{1}(\_)$ can be written $\underline{1}$ instead. Whether two occurrences of a variable bound by a common $\lambda$ are shared or not, each occurrence of the variable will be replaced by an indirection to the same argument when the scope of the function is copied. It may also be preferable to write ${}_{1}(\lambda\ {}^{body}(\ ))$ as $(\lambda_{\underline{2}}\ {}^{body}(\ ))$. To make it clearer which variable it binds. For literal primitives the $\Delta$ is superfluous, so ${}^{4}_{1}(\Delta\ \texttt{==}\ {}^{5}(\ )\ {}^{6}(\ ))$ can be abbreviated ${}_{1}(\texttt{==}\ {}^{5}(\ )\ {}^{6}(\ ))$.

Where leaving out depth tags introduces no ambiguity, they may be omitted. This occurs when there is only one depth a node could be given without breaking the rules restricting the depths of two nodes where one points to the other. Specifically the body of an abstraction node ${}_{d}(\lambda\ {}^{body}(\ ))$ can be a node with depth between 0 and $d+1$ inclusive. The body of a substitution node ${}_{d}(=\ {}^{body}(\ )\ \mathit{bind}\ {}^{arg}(\ )\ \mathit{shift}\ \mathit{fam}\ \mathit{substby})$ can be a node with depth between 0 and $d - \mathit{shift}$ inclusive. Any other node ${}_{d}(\ )$ may point to a node with depth between 0 and $d$ inclusive.

Using these conventions the `power` function can be rewritten more simply as

shown in Figure 4.5. Note which nodes the depth tags are placed on. With reference to the graphical representation of the power function shown in Figure 4.2, the nodes with addresses can be seen to be those on the shallower side of the boundary delimiting the scope of the inner abstraction. Any nodes above this boundary must have a depth of two as these nodes are able to reach the variables at depth two. Any (non atomic) nodes below this boundary must have a depth of at least one as these nodes are either the variable at depth one or are able to reach this variable and they cannot have a depth any greater than one, as they would not then be reachable by the nodes shown with a depth of one. The depth of the atomic nodes is of little consequence, although as explained earlier there is no point giving them a depth other than zero. If the scope of the inner abstraction were not as tight as it could be, i.e. if the function where not in its fully lazy form, then it would be necessary to specify the depth of nodes on both sides of the scope delimiting boundary in order to precisely pin down where the boundary lies.

## 4.2 Indirections and black holes

The indirection nodes are useful both when instantiating an expression in a heap, and during $\beta$-reductions or reduction of primitives such as `if` and `head`. From a mathematical point of view, such indirections have the unfortunate effect of making otherwise identical graphs different. Although normal indirection nodes can be eliminated from a graph, self-referential nodes cannot. Self-referential indirection nodes are known as black holes. Black holes can be created either by instantiation of expressions such as `let x=x in x` or by the reduction of expressions such as `let f x=a; a=f 1 in a` . In either case, evaluation of a black hole is undefined, typically resulting in non-termination. No special reduction rules are needed to handle black holes. For further discussion on the mathematical implications of black holes, see [2, 5].

To prevent chains of indirection nodes building up, the following reduction rule can be applied:

**Definition 4.1 (indirection elimination ($\rightarrow_i$))**
$(S, H[^a(^b(^c(\ )))], M) \rightarrow_i (S, H[^a(^c(\ ))], M)$
□

This reduction rule says that if address $a$ contains an indirection node pointing

to address $b$, and address $b$ contains an indirection node pointing to address $c$, then address $a$ can be updated to contain an indirection pointing directly to address $c$, and no change is made to the stack S or the memo-tables M.

The notation $H[...]$ denotes a heap $H$, where the nodes ... exist. Where a node is present on the left-hand side of a reduction rule, but not the right, the node is assumed to remain unchanged. For example in the rule above address $b$ still contains an indirection node pointing to address $c$.

This reduction rule will shorten chains of indirection nodes to a single indirection node. It is also possible to provide rules to remove indirection nodes that originate from terms as well as other indirection nodes. When to apply these rules is an implementation issue.

The reduction rule $\rightarrow_i$ does not imply that addresses $a$, $b$ and $c$ need be different. In the presence of black holes the reduction $\rightarrow_i$ could be applied perpetually.

A chain of indirections can be written more concisely as ${}^{a,b,c}( )$. If the length of the chain is not important the chain can be written ${}^{a,\cdots,n}( )$, or even ${}^{a,\cdots}( )$ or ${}^{\cdots,n}( )$ or ${}^{\cdot\cdot}( )$.

## 4.3 Memo-tables and the heap

A memo-table is a function from addresses to addresses. The function denoted by $\varnothing$ is undefined for all arguments. The notation $m[a \mapsto b]$ denotes function overriding, the function $m[a \mapsto b]$ is the same as the function $m$, except $m[a \mapsto b](a) = b$. The domain of a function $m$ is given by $dom(m)$. Thus if $a \notin dom(m)$ the memo-table $m$ does not have an entry for address $a$.

The memo-table association $M$ associates each beta reduction's unique identifier with a memo-table. If $f \notin dom(M)$ then $f$ is available as a new beta reduction identifier.

The heap $H$ is a function from addresses to terms and tags. If $b \notin dom(H)$ then $b$ is a fresh address. Fresh addresses are required during substitution, Substitutions are performed on copies of the nodes being substituted, the orignal nodes are not substituted.

## 4.4 Blocked reduction and tags

The reduction rules are simplified by tagging the terms with a blocked tag to indicate if reduction on a term has been attempted. Terms that have not been evaluated are tagged with a ✗. Terms for which evaluation has been attempted will be tagged with something other than a ✗. This tagging prevents work being wasted in repeated attempts to evaluate a blocked term. Without this tagging the spines of curried function calls may be repeatedly traversed only to discover reduction is blocked by the presence of a variable where a function is needed.

For completely lazy evaluation a ✓ indicates that evaluation on a node has been attempted, for example $(\#\ \mathtt{7})^{\checkmark}$ and $(\mathtt{+}\ {}_1(\_)^{\checkmark}\ (\#\ \mathtt{7})^{\checkmark})^{\checkmark}$.

For optimal evaluation a ✓ is only used if evaluation has been attempted *and* no further reduction will be possible after the term is substituted, for example $(\#\ \mathtt{7})^{\checkmark}$. Otherwise the term is tagged with a positive number indicating the variable a substitution must bind in order to enable further reduction, for example $(\mathtt{+}\ {}_1(\_)^{1}\ (\#\ \mathtt{7})^{\checkmark})^{1}$.

When a graph is first instantiated, all terms are initially tagged with a ✗. Fresh terms created by substitution are also initially tagged with a ✗.

## 4.5 Completely lazy evaluation

This section contains the reduction rules for completely lazy evaluation.

The completely lazy evaluation reduction rules only use the blocked tags ✓and ✗. The distinction between a term whose further reduction is blocked by the presence of a variable (e.g. ${}^{1}_{1}(\mathtt{@}\ {}^{2}_{1}(\_)^{\checkmark}\ {}^{3}_{0}(\#\ \mathtt{1})^{\times})^{\checkmark}$), and a term which cannot be reduced any further because it is already fully reduced (e.g. ${}^{1}_{0}(\#\ \mathtt{1})^{\checkmark}$) is not important. If a reduction rule does not specify a *blocked* tag, this means it doesn't matter if the tag is a ✓ or a ✗. Tags are assumed to remain unchanged unless specified otherwise.

The *SubstBy* field in a substitution node (=) is used to remember which substitutions a substitution node has been substituted by. The field has no use in the completely lazy reduction rules, as substitution nodes are never swapped, so it will not be shown.

**Definition 4.2 (Completely lazy reduction ($\rightarrow_c$))**

These reduction rules define a deterministic reduction strategy. The earliest appli-

cable reduction rule takes precedence. Reduction begins with a heap containing a graph, a singleton stack containing the root of the graph, and an empty set of memo-tables. Reduction terminates when the stack is empty.

$$(a : S, H[^{a,b}(\ )], M) \ \rightarrow_c \ (b : S, H, M) \qquad \text{(eval ind)}$$

$$(a : S, H[^{a}(\ )^{\checkmark}], M) \ \rightarrow_c \ (S, H, M) \qquad \text{(eval blocked)}$$

$$(a : S, H[^{a}(\#\ x)^{\text{✗}}], M) \ \rightarrow_c \ (S, H[^{a}(\#\ x)^{\checkmark}], M) \qquad \text{(eval atom)}$$

$$(a : S, H[^{a}(\_)^{\text{✗}}], M) \ \rightarrow_c \ (S, H[^{a}(\_)^{\checkmark}], M) \qquad \text{(eval var)}$$

$$(a : S, H[^{a}(\lambda\ ^{body}(\ ))^{\text{✗}}], M) \ \rightarrow_c \ (S, H[^{a}(\lambda\ ^{body}(\ ))^{\checkmark}], M) \qquad \text{(eval abs)}$$

$$(a : S, H[^{a}(:\ ^{hd}(\ )\ ^{tl}(\ ))^{\text{✗}}], M) \ \rightarrow_c \ (S, H[^{a}(:\ ^{hd}(\ )\ ^{tl}(\ ))^{\checkmark}], M) \qquad \text{(eval pair)}$$

$$(a : S, H[^{a}(@\ ^{\cdots,func}(\ )^{\text{✗}}\ ^{arg}(\ ))], M) \ \rightarrow_c \ (func : a : S, H, M) \qquad \text{(eval app)}$$

$$(a : S, H[^{a}_{i}(@\ ^{\cdots,func}(\lambda_{\underline{j}}\ ^{body}(\ ))^{\checkmark}\ ^{arg}(\ ))], M) \qquad \text{(eval beta)}$$
$$\rightarrow_c \ (a : S, H[^{a}(=\ ^{body}(\ )\ j\ ^{arg}(\ )\ (i-j)\ f)^{\text{✗}}], M[f \mapsto \varnothing])$$
where $f \notin dom(M)$

$$(a : S, H[^{a}(@\ ^{\cdots,func}(\ )^{\checkmark}\ ^{arg}(\ ))], M) \qquad \text{(eval app blocked)}$$
$$\rightarrow_c \ (S, H[^{a}(@\ ^{func}(\ )^{\checkmark}\ ^{arg}(\ ))^{\checkmark}], M)$$

$$(a : S, H[^{a}(=^{\cdots,b}_{j}(\ )^{\text{✗}}\ bind\ ^{arg}(\ )\ shift\ f)], M) \qquad \text{(subst specialize)}$$
$$\rightarrow_c \ (b : a : S, H, M), \qquad \text{if } j \geq bind$$

$$(a : S, H[^{a}(=^{\cdots,b}_{j}(\ )\ bind\ ^{arg}(\ )\ shift\ f)], M) \qquad \text{(subst scope-boundary)}$$
$$\rightarrow_c \ (S, H[^{a,b}(\ )], M), \qquad \text{if } j < bind$$

$$(a : S, H[^{a}(=^{\cdots,b}(\ )\ bind\ ^{arg}(\ )\ shift\ f)], M) \qquad \text{(subst memoized)}$$
$$\rightarrow_c \ (S, H[^{a,z}(\ )], M), \qquad \text{if } M(f)(b) = z$$

$$(a : S, H[^{a}(=^{\cdots,b}_{j}(\_)\ bind\ ^{arg}(\ )\ shift\ f)], M) \qquad \text{(subst bind)}$$
$$\rightarrow_c \ (a : S, H[^{a,arg}(\ )], M), \qquad \text{if } j = bind$$

$$(a : S, H[^{a}(=^{\cdots,b}_{j}(\_)\ bind\ ^{arg}(\ )\ shift\ f)], M[f{\mapsto}m]) \qquad \text{(subst var)}$$
$$\rightarrow_c \ (S, H[^{a}_{j+shift}(\_)^{\checkmark}], M[f \mapsto m[b \mapsto a]])$$

$$(a : S, H[^{a}(=^{\cdots,b}_{j}(@\ ^{c}(\ )\ ^{d}(\ ))\ bind\ ^{arg}(\ )\ shift\ f)], M[f{\mapsto}m]) \qquad \text{(subst app)}$$
$$\rightarrow_c \quad (a : S,$$
$$H[^{a}_{j+shift}(@\ \ ^{e}_{j+shift}(=\ ^{c}(\ )\ bind\ ^{arg}(\ )\ shift\ f)^{\text{✗}}$$
$$\qquad\qquad ^{f}_{j+shift}(=\ ^{d}(\ )\ bind\ ^{arg}(\ )\ shift\ f)^{\text{✗}})^{\text{✗}}],$$
$$M[f \mapsto m[b \mapsto a]])$$
where $e, f \notin dom(H)$

$(a : S, H[^{a}(=\ {}^{\cdots,b}_{j}(\lambda\ {}^{body}(\ ))\ bind\ {}^{arg}(\ )\ shift\ f)], M[f{\mapsto}m])$ (subst abs)

$\rightarrow_c\ \ (S,$

$H[^{a}_{j+shift}(\lambda\ {}^{c}_{j+shift+1}(=\ {}^{body}(\ )\ bind\ {}^{arg}(\ )\ shift\ f)^{\text{✗}})^{\text{✓}}],$

$M[f \mapsto m[b \mapsto a]])$

where $c \notin dom(H)$

$(a : S, H[^{a}(=\ {}^{\cdots,b}_{j}(:\ {}^{c}(\ )\ {}^{d}(\ ))\ bind\ {}^{arg}(\ )\ shift\ f)], M[f{\mapsto}m])$ (subst pair)

$\rightarrow_c\ \ (S,$

$H[^{a}_{j+shift}(:\ \ {}^{e}_{j+shift}(=\ {}^{c}(\ )\ bind\ {}^{arg}(\ )\ shift\ f)^{\text{✗}}$

$\qquad\qquad {}^{f}_{j+shift}(=\ {}^{d}(\ )\ bind\ {}^{arg}(\ )\ shift\ f)^{\text{✗}})^{\text{✓}}],$

$M[f \mapsto m[b \mapsto a]])$

where $e, f \notin dom(H)$

$(a : S, H[^{a}(=\ {}^{\cdots,b}_{j}(\#\ x)\ bind\ {}^{arg}(\ )\ shift\ f)], M)$ (subst atom)

$\rightarrow_c\ \ (a : S, H[^{a}_{j+shift}(\#\ x)^{\text{✗}}], M)$

$(a : S,\ H[^{a}(=\ {}^{\cdots,b}_{j}(\Delta_n\ p\ {}^{c_1}(\ )\ \ldots{}^{c_n}(\ ))\ bind\ {}^{arg}(\ )\ shift\ f)],\ M[f \mapsto m])$ (subst prim)

$\rightarrow_c\ \ (a : S,$

$H[^{a}_{j+shift}(\Delta_n\ \ p\ {}^{d_1}_{j+shift}(=\ {}^{c_1}(\ )\ bind\ {}^{arg}(\ )\ shift\ f)^{\text{✗}}$

$\qquad\qquad \ldots\ {}^{d_n}_{j+shift}(=\ {}^{c_n}(\ )\ bind\ {}^{arg}(\ )\ shift\ f)^{\text{✗}})^{\text{✗}}],$

$M[f \mapsto m[b \mapsto a]])$

where $d_1, \ldots, d_n \notin dom(H)$

$(a : S, H[^{a}(+\ {}^{\cdots,b}(\#\ x)^{\text{✓}}\ {}^{\cdots,c}(\#\ y)^{\text{✓}})^{\text{✗}}], M)$ (prim + reduced)

$\rightarrow_c\ (S, H[^{a}_{0}(\#\ (x + y))^{\text{✓}}], M), \qquad \text{if } x, y \in \mathbb{Z}$

$(a : S, H[^{a}(+\ {}^{\cdots,b}(\ )^{\text{✓}}\ {}^{\cdots,c}(\ )^{\text{✓}})^{\text{✗}}], M)\ \rightarrow_c\ (S, H[^{a}(\ )^{\text{✓}}], M)$ (prim + blocked)

$(a : S, H[^{a}(+\ {}^{\cdots,b}(\ )\ {}^{\cdots,c}(\ ))^{\text{✗}}], M)\ \rightarrow_c\ (b : c : a : S, H, M)$ (prim + in-progress)

$(a : S, H[^{a}(\texttt{if True}\ {}^{c}(\ )\ {}^{d}(\ ))], M)$ (prim `if True` reduced)

$\rightarrow_c\ (a : S, H[^{a,c}(\ )], M)$

$(a : S, H[^{a}(\texttt{if False}\ {}^{c}(\ )\ {}^{d}(\ ))], M)$ (prim `if False` reduced)

$\rightarrow_c\ (a : S, H[^{a,d}(\ )], M)$

$(a : S, H[^{a}(\texttt{if}\ {}^{\cdots,b}(\ )^{\text{✓}}\ {}^{c}(\ )\ {}^{d}(\ ))], M)$ (prim `if` blocked)

$\rightarrow_c\ (S, H[^{a}(\ )^{\text{✓}}], M)$

$(a : S, H[^{a}(\texttt{if}\ ^{\cdots,b}(\ )^{\text{✗}}\ ^{c}(\ )\ ^{d}(\ ))], M)$ (prim `if` in-progress)
$\quad \rightarrow_c\ (b : a : S, H, M)$

$(a : S, H[^{a}(\texttt{head}\ ^{\cdots,b}(:\ ^{hd}(\ )\ ^{tl}(\ ))^{\text{✓}})], M)$ (prim `head` reduced)
$\quad \rightarrow_c\ (a : S, H[^{a,hd}(\ )], M)$
$(a : S, H[^{a}(\texttt{head}\ ^{\cdots,b}(\ )^{\text{✓}})], M) \rightarrow_c (S, H[^{a}(\ )^{\text{✓}}], M)$ (prim `head` blocked)
$(a : S, H[^{a}(\texttt{head}\ ^{\cdots,b}(\ )^{\text{✗}})], M) \rightarrow_c (b : a : S, H, M)$ (prim `head` in-progress)
□

The reduction rule (eval ind) applies when the node whose address is at the top of the stack, is an indirection node. In this case the address at the top of the stack is replaced with the address which the indirection points to.

The rule (eval blocked) applies when the node whose address is at the top of the stack is a term tagged with a ✓. The ✓ tag indicates the term cannot be reduced further. In this case this address is popped from the stack and discarded. For top-level reduction of a well-typed expression, it is always clear whether a term is in its fully reduced form: Application and primitive nodes can always be reduced. Abstraction, atomic and pair nodes are already fully reduced. And variable and substitution nodes will never be encountered. When reduction is performed within the scope of an abstraction, identifying whether a node has been reduced or not is not so simple. The ✓ tag prevents repeated attempts to reduce the unreducible application and primitive nodes typically found within function bodies. Such unreducible application and primitive nodes can also occur at the top-level if the expression being reduced is not well-typed. For example reducing $(\texttt{+ 1 []})^{\text{✗}}$ results in $(\texttt{+ 1 []})^{\text{✓}}$, just as the reduction of $(\texttt{+ 1 }\underline{1}\ )^{\text{✗}}$ results in $(\texttt{+ 1 }\underline{1}\ )^{\text{✓}}$.

The reduction rules (eval atom), (eval var), (eval abs) and (eval pair), state that atoms, variables, abstractions and pairs are already in their fully reduced state and should be tagged with a ✓.

The rule (eval app) ensures the function part of an `@` node is evaluated by pushing the address of the function part onto the stack. When evaluation of the function part is complete, its address will be popped off the stack and a second attempt to reduce the `@` node will be made.

If the function part of an application node evaluates to an abstraction node, or an indirection or chain of indirections terminating in an abstraction node, then the reduction rule (eval beta) performs a beta reduction. A fresh beta reduction

identifier $f$ is associated with this beta reduction and a fresh memo-table, initially empty is associated with $f$. The (eval beta) rule does not evaluate the body of the abstraction. If the function part of the application node does not reduce to an abstraction node, then the application node cannot be reduced further and is tagged with a ✓ as shown in (eval app blocked).

The reduction rule (subst specialize) ensures that the body which a substitution node points to is evaluated before it is substituted. This reduction rule is responsible for the specializing effect. If the body has already been reduced and hence tagged with a ✓, then (subst specialized) does not apply and repeated attempts to evaluate the body will not be made.

The rule (subst scope) checks to see if a substitution node has reached the end of its scope. If it has then the substitution *dies out* and the substitution node is replaced with an indirection pointing to the body of the substitution.

The rule (subst memo) checks to see if the node to be substituted has already been substituted by a substitution node originating from the same $\beta$-reduction as the present substitution node. If so then the substitution node is replaced with an indirection pointing to the result of the previous substitution. There is no possibility of memo-tables failing due to indirection nodes causing confusion over the identity of nodes, as the address of indirection nodes are never used to index memo-tables. Only nodes that have already been tagged with a ✓ will be used to index memo-tables, and once a node has been tagged with a ✓ it will not subsequently be overwritten with an indirection or any other node.

The rules (subst bind) and (subst var) handle the substitution of variables. If the depth of the variable being substituted matches *bind* in the substitution, then (subst bind) replaces the substitution node with an indirection pointing to the address *arg* in the substitution. If the depth doesn't match, then (subst var) replaces the substitution node with a variable with depth equal to the depth of the variable being substituted plus *shift*, effectively renaming the variable.

The rules (subst app), (subst abs), (subst pair) and (subst prim), replace the substitution node with a @, $\lambda$, : or $\Delta$ node shifted in depth from the node being substituted and build new child substitution nodes with fresh addresses ready to substitute the child nodes of the node being substituted. In each case one new memo-table entry is made, associating the substituted node with the (now over written) node that substituted it, in the memo-table associated with beta-reduction

```
let f x = let y = head x
              z = y+y
          in if z < 0 then negate z else z
in f [1]
```

Figure 4.6: Completely lazy reduction example

that created the substitution. The node being substituted and its children remain unchanged.

Reduction rules for three primitive functions are shown. It should be clear from these how the others would be written. The primitive reduction rules contain within them the strictness of the primitive operations. The reduction rule (prim `if`) indicates that `if` is only strict in its first argument, where as (prim `+`) indicates that `+` is strict in both its arguments.

The (prim `+` reduced) rule applies when the reduction can be performed. If both arguments have been reduced as far as possible, but the previous rule did not apply then reduction of the `+` node is blocked, and the rule (prim `+` blocked) applies. This typically happens when a variable is present in one or both arguments, but will also occur if there is a type-error. If one or both the arguments have not been reduced, then both their addresses are placed on the stack, and reduction of the `+` node will be attempted again when both arguments have been reduced.

The (prim `if` reduced) rule applies when the predicate had been reduced to `True`. In this case the `if` node is replaced with an indirection pointing to the then clause of the `if` node.

## 4.6 Examples

To demonstrate the completely lazy reduction rules in action the reduction steps required to reduce the expression in Figure 4.6 will be explained.

The stack initially contains the root of the expression to be reduced. The initial state of the stack, heap and memo-tables are:

stack: [1]

heap: ${}^{1}_{0}$(`@` ${}^{2}$( ) ${}^{10}_{0}$(`:` ${}^{11}_{0}$(`1`)$^{\times}$ ${}^{12}_{0}$(`[]`)$^{\times}$)$^{\times}$)$^{\times}$

${}^{2}_{0}$($\lambda$ ${}^{3}_{1}$(`if` ${}^{4}$( ) ${}^{9}$( ) ${}^{5}$( ))$^{\times}$)$^{\times}$

${}^{4}_{1}$(`<` ${}^{5}$( ) ${}^{8}_{0}$(`0`)$^{\times}$)$^{\times}$

${}^{5}_{1}$(`+` ${}^{6}$( ) ${}^{6}$( ))$^{\times}$

$$\begin{array}{l} \quad\quad {}^{6}_{1}(\texttt{head}\ {}^{7}_{1}(\_)^{\times})^{\times} \\ {}^{9}_{1}(\texttt{negate}\ {}^{5}(\ )^{\times} \end{array}$$

memo-tables: {}

The first applicable reduction rule is (eval app). This rule pushes address 2 on to the stack and leaves the heap unchanged. Next (eval abs) is applied, this changes the ✗ tag on the $\lambda$ node to a ✓ tag, and pops and discards address 2 from the stack. Next (eval beta) is applied, this performs a beta reduction of the application at address 1.

The stack, heap and memo-tables now contain:

stack: [1]

$$\begin{array}{l} \text{heap: } {}^{1}_{0}(= {}^{3}(\ )\ 1\ {}^{10}(\ )\ \text{-1}\ 1)^{\times} \\ \quad\quad {}^{3}_{1}(\texttt{if}\ {}^{4}(\ )\ {}^{9}(\ )\ {}^{5}(\ ))^{\times} \\ \quad\quad\quad {}^{4}_{1}(\texttt{<}\ {}^{5}(\ )\ {}^{8}_{0}(\texttt{0})^{\times})^{\times} \\ \quad\quad\quad\quad {}^{5}_{1}(\texttt{+}\ {}^{6}(\ )\ {}^{6}(\ ))^{\times} \\ \quad\quad\quad\quad\quad {}^{6}_{1}(\texttt{head}\ {}^{7}_{1}(\_)^{\times})^{\times} \\ \quad\quad\quad {}^{9}_{1}(\texttt{negate}\ {}^{5}(\ )^{\times} \\ \quad\quad {}^{10}_{0}(\texttt{:}\ {}^{11}_{0}(\texttt{1})^{\times}\ {}^{12}_{0}(\texttt{[]})^{\times})^{\times} \end{array}$$

memo-tables: $\{\ 1 \mapsto \varnothing\ \}$

The next applicable rule is (subst specialize). This rule pushes address 3 onto the stack, so as to ensure the nodes to be substituted are evaluated before substituting (and copying) them. Then rule (prim `if` in-progress) applies, this pushes address 4 onto the stack in an attempt to evaluate the condition of the `if` primitive. The rule (prim `<` in-progress) applies followed by rules (prim `+` in-progress) and (prim `head` in-progress), and then (eval var). The rule (eval var) changes the ✗ tag in the variable at address 7 to a ✓ tag.

stack: [5,4,3,1]

$$\begin{array}{l} \text{heap: } {}^{1}_{0}(= {}^{3}(\ )\ 1\ {}^{10}(\ )\ \text{-1}\ 1)^{\times} \\ \quad\quad {}^{3}_{1}(\texttt{if}\ {}^{4}(\ )\ {}^{9}(\ )\ {}^{5}(\ ))^{\times} \\ \quad\quad\quad {}^{4}_{1}(\texttt{<}\ {}^{5}(\ )\ {}^{8}_{0}(\texttt{0})^{\times})^{\times} \\ \quad\quad\quad\quad {}^{5}_{1}(\texttt{+}\ {}^{6}(\ )\ {}^{6}(\ ))^{\times} \\ \quad\quad\quad\quad\quad {}^{6}_{1}(\texttt{head}\ {}^{7}_{1}(\_)^{\checkmark})^{\times} \\ \quad\quad\quad {}^{9}_{1}(\texttt{negate}\ {}^{5}(\ )^{\times} \\ \quad\quad {}^{10}_{0}(\texttt{:}\ {}^{11}_{0}(\texttt{1})^{\times}\ {}^{12}_{0}(\texttt{[]})^{\times})^{\times} \end{array}$$

memo-tables: { 1 ↦ ∅ }

The rule (eval `head` blocked) then changes the tag on the head primitive at address 6 to a ✓ tag. Similarly the tags at addresses 3, 4, 5 and 8 are changed to ✓ tags.

The result of the evaluation of the body of the substitution is:

stack: [1]

heap: ${}^{1}_{0}(= {}^{3}(\ )\ 1\ {}^{10}(\ )\ \text{-1}\ 1)^{\times}$
${}^{3}_{1}(\texttt{if}\ {}^{4}(\ )\ {}^{9}(\ )\ {}^{5}(\ ))^{\checkmark}$
${}^{4}_{1}(\texttt{<}\ {}^{5}(\ )\ {}^{8}_{0}(\texttt{0})^{\checkmark})^{\checkmark}$
${}^{5}_{1}(\texttt{+}\ {}^{6}(\ )\ {}^{6}(\ ))^{\checkmark}$
${}^{6}_{1}(\texttt{head}\ {}^{7}_{1}(\_)^{\checkmark})^{\checkmark}$
${}^{9}_{1}(\texttt{negate}\ {}^{5}(\ ))^{\times}$
${}^{10}_{0}(\texttt{:}\ {}^{11}_{0}(\texttt{1})^{\times}\ {}^{12}_{0}(\texttt{[]})^{\times})^{\times}$

memo-tables: { 1 ↦ ∅ }

Next (subst prim) is performed. This substitutes the `if` primitive at address 3. The node at address 3 remains unchanged, it will no longer play a part in the reduction. A copy of the node at address 3 is made at address 78. This node points via three new substitution nodes to the same nodes that the node at address 3 pointed to. A record of the substitution is made in a memo-table.

stack: [1]

heap: ${}^{1,78}_{0}(\texttt{if}\ {}^{75}(\ )\ {}^{76}(\ )\ {}^{77}(\ ))^{\times}$
${}^{75}_{0}(= {}^{4}(\ )\ 1\ {}^{10}(\ )\ \text{-1}\ 1)^{\times}$
${}^{76}_{0}(= {}^{9}(\ )\ 1\ {}^{10}(\ )\ \text{-1}\ 1)^{\times}$
${}^{77}_{0}(= {}^{5}(\ )\ 1\ {}^{10}(\ )\ \text{-1}\ 1)^{\times}$
${}^{4}_{1}(\texttt{<}\ {}^{5}(\ )\ {}^{8}_{0}(\texttt{\#}\ \texttt{0})^{\checkmark})^{\checkmark}$
${}^{5}_{1}(\texttt{+}\ {}^{6}(\ )\ {}^{6}(\ ))^{\checkmark}$
${}^{6}_{1}(\texttt{head}\ {}^{7}_{1}(\_)^{\checkmark})^{\checkmark}$
${}^{10}_{0}(\texttt{:}\ {}^{11}_{0}(\texttt{1})^{\times}\ {}^{12}_{0}(\texttt{[]})^{\times})^{\times}$
${}^{9}_{1}(\texttt{negate}\ {}^{5}(\ ))^{\times}$

memo-tables: { 1 ↦ {3 ↦ 78} }

The substitution node at address 1 has been overwritten with an indirection node pointing to the new node at address 78. Next (eval ind) will follow this indirection and push address 78 onto the stack. The (eval `if` in-progress) rule will attempt to

evaluate the condition of the `if` primitive. This in turn results in the substitution node at address 75, substituting the primitive at address 4.

stack: [75,78,1]

heap: ${}^{1,78}_{0}$(`if` ${}^{75}$( ) ${}^{76}$( ) ${}^{77}$( ))$^{\times}$
${}^{75,81}_{0}$(`<` ${}^{79}$( ) ${}^{80}$( ))$^{\times}$
${}^{79}_{0}$(`=` ${}^{5}$( ) 1 ${}^{10}$( ) -1 1)$^{\times}$
${}^{5}_{1}$(`+` ${}^{6}$( ) ${}^{6}$( ))$^{\checkmark}$
${}^{6}_{1}$(`head` ${}^{7}_{1}$(_)$^{\checkmark}$)$^{\checkmark}$
${}^{10}_{0}$(`:` ${}^{11}_{0}$(`1`)$^{\times}$ ${}^{12}_{0}$(`[]`)$^{\times}$)$^{\times}$
${}^{80}_{0}$(`=` ${}^{8}$( ) 1 ${}^{10}$( ) -1 1)$^{\times}$
${}^{8}_{0}$(`0`)$^{\checkmark}$
${}^{76}_{0}$(`=` ${}^{9}$( ) 1 ${}^{10}$( ) -1 1)$^{\times}$
${}^{9}_{1}$(`negate` ${}^{5}$( ))$^{\times}$
${}^{77}_{0}$(`=` ${}^{5}$( ) 1 ${}^{10}$( ) -1 1)$^{\times}$

memo-tables: { 1 ↦ {3 ↦ 78, 4 ↦ 81} }

Substitution of the nodes at addresses 5 and 6 proceed in a similar fashion.

stack: [7,85,82,83,79,80,75,78,1]

heap: ${}^{1,78}_{0}$(`if` ${}^{75}$( ) ${}^{76}$( ) ${}^{77}$( ))$^{\times}$
${}^{75,81}_{0}$(`<` ${}^{79}$( ) ${}^{80}$( ))$^{\times}$
${}^{79,84}_{0}$(`+` ${}^{82}$( ) ${}^{83}$( ))$^{\times}$
${}^{82,86}_{0}$(`head` ${}^{85}$( ))$^{\times}$
${}^{85}_{0}$(`=` ${}^{7}$( ) 1 ${}^{10}$( ) -1 1)$^{\times}$
${}^{7}_{1}$(_)$^{\checkmark}$
${}^{10}_{0}$(`:` ${}^{11}_{0}$(`1`)$^{\times}$ ${}^{12}_{0}$(`[]`)$^{\times}$)$^{\times}$
${}^{83}_{0}$(`=` ${}^{6}$( ) 1 ${}^{10}$( ) -1 1)$^{\times}$
${}^{6}_{1}$(`head` ${}^{7}$( ))$^{\checkmark}$
${}^{80}_{0}$(`=` ${}^{8}$( ) 1 ${}^{10}$( ) -1 1)$^{\times}$
${}^{8}_{0}$(`0`)$^{\checkmark}$
${}^{76}_{0}$(`=` ${}^{9}$( ) 1 ${}^{10}$( ) -1 1)$^{\times}$
${}^{9}_{1}$(`negate` ${}^{5}_{1}$(`+` ${}^{6}$( ) ${}^{6}$( ))$^{\checkmark}$)$^{\times}$
${}^{77}_{0}$(`=` ${}^{5}$( ) 1 ${}^{10}$( ) -1 1)$^{\times}$

memo-tables: { 1 ↦ {3 ↦ 78, 4 ↦ 81, 5 ↦ 84, 6 ↦ 86} }

At this point (subst bind) is performed, and the substitution at node 85 is overwritten with an indirection node pointing to address 10.

stack: [85,82,83,79,80,75,78,1]

heap: $^{1,78}_{0}$(`if` $^{75}$( ) $^{76}$( ) $^{77}$( ))$^{\times}$
$^{75,81}_{0}$(`<` $^{79}$( ) $^{80}$( ))$^{\times}$
$^{79,84}_{0}$(`+` $^{82}$( ) $^{83}$( ))$^{\times}$
$^{82,86}_{0}$(`head` $^{85,10}$( ))$^{\times}$
$^{10}_{0}$(`:` $^{11}_{0}$(`1`)$^{\times}$ $^{12}_{0}$(`[]`)$^{\times}$)$^{\times}$
$^{83}_{0}$(`=` $^{6}$( ) 1 $^{10}$( ) -1 1)$^{\times}$
$^{6}_{1}$(`head` $^{7}_{1}$(`_`)$^{\checkmark}$)$^{\checkmark}$
$^{80}_{0}$(`=` $^{8}$( ) 1 $^{10}$( ) -1 1)$^{\times}$
$^{8}_{0}$(`0`)$^{\checkmark}$
$^{76}_{0}$(`=` $^{9}$( ) 1 $^{10}$( ) -1 1)$^{\times}$
$^{9}_{1}$(`negate` $^{5}_{1}$(`+` $^{6}$( ) $^{6}$( ))$^{\checkmark}$)$^{\times}$
$^{77}_{0}$(`=` $^{5}$( ) 1 $^{10}$( ) -1 1)$^{\times}$

memo-tables: { 1 ↦ {3 ↦ 78, 4 ↦ 81, 5 ↦ 84, 6 ↦ 86} }

The reduction rule (eval pair) tags the pair at address 10 with a ✓. The reduction rule (prim `head` reduced) reduces the primitive `head` node at address 86, replacing it with an indirection pointing to address 11.

stack [11,82,83,79,80,75,78,1]

heap: $^{1,78}_{0}$(`if` $^{75}$( ) $^{76}$( ) $^{77}$( ))$^{\times}$
$^{75,81}_{0}$(`<` $^{79,84}_{0}$(`+` $^{82,86,11}_{0}$(`1`)$^{\times}$ $^{83}$( ))$^{\times}$ $^{80}$( ))$^{\times}$
$^{83}_{0}$(`=` $^{6}$( ) 1 $^{10}$( ) -1 1)$^{\times}$
$^{6}_{1}$(`head` $^{7}_{1}$(`_`)$^{\checkmark}$)$^{\checkmark}$
$^{10}_{0}$(`:` $^{11}$( ) $^{12}_{0}$(`[]`)$^{\times}$)$^{\checkmark}$
$^{80}_{0}$(`=` $^{8}$( ) 1 $^{10}$( ) -1 1)$^{\times}$
$^{8}_{0}$(`0`)$^{\checkmark}$
$^{76}_{0}$(`=` $^{9}$( ) 1 $^{10}$( ) -1 1)$^{\times}$
$^{9}_{1}$(`negate` $^{5}_{1}$(`+` $^{6}$( ) $^{6}$( ))$^{\checkmark}$)$^{\times}$
$^{77}_{0}$(`=` $^{5}$( ) 1 $^{10}$( ) -1 1)$^{\times}$

memo-tables: { 1 ↦ {3 ↦ 78, 4 ↦ 81, 5 ↦ 84, 6 ↦ 86} }

The reduction rule (eval atom) tags the atomic node at address 11 with a ✓. The first argument of the + primitive at address 84 is now reduced, reduction continues

with the reduction of the second argument, at address 83. The substitution node at address 83 wants to substitute the node at address 6. This node has already been substituted by a related substitution. Instead of repeating the substitution, the substitution node is simply replaced with a indirection to the result of the previous substitution.

stack: [82,83,79,80,75,78,1]

heap: $^{1,78}_{0}$(if $^{75}$( ) $^{76}$( ) $^{77}$( ))$^{\times}$
$^{75,81}_{0}$(< $^{79,84}_{0}$(+ $^{82,86,11}_{0}$(1)$^{\checkmark}$ $^{83,86}$( ))$^{\times}$ $^{80}$( ))$^{\times}$
$^{80}_{0}$(= $^{8}$( ) 1 $^{10}$( ) -1 1)$^{\times}$
$^{8}_{0}$(0)$^{\checkmark}$
$^{10}_{0}$(: $^{11}$( ) $^{12}_{0}$([])$^{\times}$)$^{\checkmark}$
$^{76}_{0}$(= $^{9}$( ) 1 $^{10}$( ) -1 1)$^{\times}$
$^{9}_{1}$(negate $^{5}_{1}$(+ $^{6}$( ) $^{6}$( ))$^{\checkmark}$)$^{\times}$
$^{6}_{1}$(head $^{7}_{1}$(_)$^{\checkmark}$)$^{\checkmark}$
$^{77}_{0}$(= $^{5}$( ) 1 $^{10}$( ) -1 1)$^{\times}$

memo-tables: { 1 ↦ {3 ↦ 78, 4 ↦ 81, 5 ↦ 84, 6 ↦ 86} }

The + primitive can now be reduced.

stack: [79,80,75,78,1]

heap: $^{1,78}_{0}$(if $^{75}$( ) $^{76}$( ) $^{77}$( ))$^{\times}$
$^{75,81}_{0}$(< $^{79,84}_{0}$(2)$^{\checkmark}$ $^{80}$( ))$^{\times}$
$^{80}_{0}$(= $^{8}$( ) 1 $^{10}$( ) -1 1)$^{\times}$
$^{8}_{0}$(0)$^{\checkmark}$
$^{10}_{0}$(: $^{11}_{0}$(1)$^{\checkmark}$ $^{12}_{0}$([])$^{\times}$)$^{\checkmark}$
$^{76}_{0}$(= $^{9}$( ) 1 $^{10}$( ) -1 1)$^{\times}$
$^{9}_{1}$(negate $^{5}_{1}$(+ $^{6}$( ) $^{6}$( ))$^{\checkmark}$)$^{\times}$
$^{6}_{1}$(head $^{7}_{1}$(_)$^{\checkmark}$)$^{\checkmark}$
$^{77}_{0}$(= $^{5}$( ) 1 $^{10}$( ) -1 1)$^{\times}$

memo-tables: { 1 ↦ {3 ↦ 78, 4 ↦ 81, 5 ↦ 84, 6 ↦ 86} }

Reduction now continues with the second argument of the < primitive, at address 80. The substitution node at address 80 is attempting to substitute the atomic node at address 8. The depth of the atomic node is shallower than the binding depth of the substitution. This causes (subst scope-boundary) to replace the substitution

with an indirection pointing to address 8.

stack: [79,80,75,78,1]

heap: $^{1,78}_{0}$(`if` $^{75}$( ) $^{76}$( ) $^{77}$( ))$^{\times}$
$^{75,81}_{0}$(`<` $^{79,84}_{0}$(`2`)$^{\checkmark}$ $^{80,8}_{0}$(`0`)$^{\checkmark}$)$^{\times}$
$^{76}_{0}$(`=` $^{9}$( ) `1` $^{10}$( ) `-1` `1`)$^{\times}$
$^{9}_{1}$(`negate` $^{5}_{1}$(`+` $^{6}$( ) $^{6}$( ))$^{\checkmark}$)$^{\times}$
$^{6}_{1}$(`head` $^{7}_{1}$(`_`)$^{\checkmark}$)$^{\checkmark}$
$^{10}_{0}$(`:` $^{11}_{0}$(`1`)$^{\checkmark}$ $^{12}_{0}$(`[]`)$^{\times}$)$^{\checkmark}$
$^{77}_{0}$(`=` $^{5}$( ) `1` $^{10}$( ) `-1` `1`)$^{\times}$

memo-tables: { 1 ↦ {3 ↦ 78, 4 ↦ 81, 5 ↦ 84, 6 ↦ 86} }

The `<` primitive can now be reduced.

stack: [75,78,1]

heap: $^{1,78}_{0}$(`if` $^{75,81}_{0}$(`False`)$^{\checkmark}$ $^{76}$( ) $^{77}$( ))$^{\times}$
$^{76}_{0}$(`=` $^{9}$( ) `1` $^{10}$( ) `-1` `1`)$^{\times}$
$^{9}_{1}$(`negate` $^{5}_{1}$(`+` $^{6}$( ) $^{6}$( ))$^{\checkmark}$)$^{\times}$
$^{6}_{1}$(`head` $^{7}_{1}$(`_`)$^{\checkmark}$)$^{\checkmark}$
$^{10}_{0}$(`:` $^{11}_{0}$(`1`)$^{\checkmark}$ $^{12}_{0}$(`[]`)$^{\times}$)$^{\checkmark}$
$^{77}_{0}$(`=` $^{5}$( ) `1` $^{10}$( ) `-1` `1`)$^{\times}$

memo-tables: { 1 ↦ {3 ↦ 78, 4 ↦ 81, 5 ↦ 84, 6 ↦ 86} }

And now the if primitive can be reduced, the substitution at address 78 is replaced with an indirection pointing to address 77, the else clause of the if primitive.

stack: [77,78,1]

heap: $^{1,78,77}_{0}$(`=` $^{5}$( ) `1` $^{10}$( ) `-1` `1`)$^{\times}$
$^{5}_{1}$(`+` $^{6}$( ) $^{6}$( ))$^{\checkmark}$
$^{6}_{1}$(`head` $^{7}_{1}$(`_`)$^{\checkmark}$)$^{\checkmark}$
$^{10}_{0}$(`:` $^{11}_{0}$(`1`)$^{\checkmark}$ $^{12}_{0}$(`[]`)$^{\times}$)$^{\checkmark}$

memo-tables: { 1 ↦ {3 ↦ 78, 4 ↦ 81, 5 ↦ 84, 6 ↦ 86} }

Finally the substitution node at address 77 is trying to substitute the primitive at address 5. Since this node has already been substituted by a related substitution, the substitution is replaced with an indirection pointing to the result of the previous substitution.

stack: [1]

heap: ${}^{1,78,77,84}_{0}(2)^{\checkmark}$

memo-tables: { $1 \mapsto \{3 \mapsto 78,\ 4 \mapsto 81,\ 5 \mapsto 84,\ 6 \mapsto 86\}$ }

The last element on the stack is popped and discarded. Since the stack is now empty no further reductions are performed.

## 4.7 Optimal evaluation

In the reduction rules for complete laziness, the reason why further reduction is blocked is not important. In the reduction rules for optimal evaluation, the reason why further reduction is blocked is crucial. As well as using ✓ and ✗ to indicate whether further reduction is blocked, a positive integer, $n$ indicates that further reduction is blocked by the need for a substitution binding a variable at depth $n$. Zero is not needed as a binding depth as variables cannot exist at the top-level, all variables exist within the scope of their abstraction. A node such as an atomic node or pair which cannot be further reduced by substitution, is tagged with a ✓, just as was the case for complete laziness, (e.g. ${}^{1}_{0}(\#\ \mathtt{1})^{\checkmark}$). A node for which further reduction would be possible after substitution is tagged to indicate which variable a suitable substitution must bind in order to further reduction, (e.g. ${}^{1}_{1}(@\ {}^{2}_{1}(\_)^{1}\ {}^{3}_{0}(\#\ \mathtt{1})^{\times})^{1}$ ).

**Definition 4.3 (Optimal evaluation reduction ($\rightarrow_o$))**
These reduction rules define a deterministic reduction strategy. The earliest applicable reduction rule takes precedence. Reduction begins with a heap containing a graph, a singleton stack containing the root of the graph, and an empty set of memo-tables. Reduction terminates when the stack is empty.

$(a : S, H[{}^{a,b}(\ )], M) \rightarrow_o (b : S, H, M)$ (eval ind)

$(a : S, H[{}^{a}(\ )^{\checkmark}], M) \rightarrow_o (S, H, M)$ (eval finished)

$(a : S, H[{}^{a}(\ )^{v}], M) \rightarrow_o (S, H, M), \quad \text{if } v \in \mathbb{N}_1$ (eval blocked)

$(a : S, H[{}^{a}(\#\ x)^{\times}], M) \rightarrow_o (S, H[{}^{a}(\#\ x)^{\checkmark}], M)$ (eval atomic)

$(a : S, H[{}^{a}_{n}(\_)^{\times}], M) \rightarrow_o (S, H[{}^{a}_{n}(\_)^{n}], M)$ (eval variable)

$(a : S, H[{}^{a}(\lambda_{\underline{n}}\ {}^{body}(\ ))^{\times}], M) \rightarrow_o (S, H[{}^{a}(\lambda_{\underline{n}}\ {}^{body}(\ ))^{\checkmark}], M)$ (eval abstraction)

$(a : S, H[{}^{a}(:\ {}^{hd}(\ )\ {}^{tl}(\ ))^{\times}], M) \rightarrow_o (S, H[{}^{a}(:\ {}^{hd}(\ )\ {}^{tl}(\ ))^{\checkmark}], M)$ (eval pair)

$(a : S, H[{}^{a}(@\ {}^{\cdot,func}(\ )^{\times}\ {}^{arg}(\ ))^{\times}], M) \rightarrow_o (func : a : S, H, M)$ (eval app)

$(a : S, H[{}^{a}_{i}(@\ {}^{\cdot,b}(\lambda_{\underline{j}}\ {}^{body}(\ ))^{\checkmark}\ {}^{arg}(\ ))], M) \rightarrow_o$ (eval beta)

$(a : S, H[^{a}_{i}(= {}^{body}(\ )\ j\ {}^{arg}(\ )\ (i-j)\ f\ [])], M[f \mapsto \varnothing]),$ where $f \notin dom(M)$

$(a : S, H[^{a}_{i}(@\ {}^{\cdots,func}(\ )^{v}\ {}^{arg}(\ ))^{\boldsymbol{\times}}], M) \rightarrow_o$ (eval app blocked)

$(S, H[^{a}_{i}(@\ {}^{func}(\ )^{v}\ {}^{arg}(\ ))^{v}], M)$

$(a : S, H[^{a}(= ^{\cdots,b}_{j}(\ )^{\boldsymbol{\times}}\ bind\ {}^{arg}(\ )\ shift\ f\ sb)], M) \rightarrow_o$ (subst specialize)

$(b : a : S, H, M),$ if $j \geq bind$

$(a : S, H[^{a}(= ^{\cdots,b}_{j}(\ )\ bind\ {}^{arg}(\ )\ shift\ f\ sb)], M) \rightarrow_o$ (subst scope-boundary)

$(S, H[^{a,b}_{j}(\ )], M),$ if $j < bind$

$(a : S, H[^{a}(= ^{\cdots,b}_{j}(\ )\ bind\ {}^{arg}(\ )\ shift\ f\ sb)], M) \rightarrow_o$ (subst memoized)

$(S, H[^{a,z}(\ )], M),$ if $M(f)(body) = z$

$(a : S, H[^{a}(= ^{\cdots,b}(\ )^{\checkmark} bind\ {}^{arg}(\ )\ shift\ f\ sb)], M) \rightarrow_o$ (subst node)

$(\mathcal{S}\ (a, b)\ (bind, arg, shift, f, sb) : a : S, H, M)$

$(a : S, H[^{a}(= ^{\cdots,b}(\ )^{v}\ bind\ {}^{arg}(\ )\ shift\ f\ sb)], M) \rightarrow_o$ (subst found)

$(\mathcal{C}\ (a, [(bind, arg, shift, f, sb)], b) : S, H, M),$ if $v = bind$

$(a : S, H[^{a}(= ^{\cdots,b}(\ )^{v}\ bind\ {}^{arg}(\ )\ shift\ f\ sb)^{\boldsymbol{\times}}], M) \rightarrow_o$ (subst blocked)

$(S, H[^{a}(= {}^{b}(\ )^{v}\ bind\ {}^{arg}(\ )\ shift\ f\ sb)^{v}], M)$

if $v < bind$

$(a : S, H[^{a}(= ^{\cdots,b}(\ )^{v}\ bind\ {}^{arg}(\ )\ shift\ f\ sb)^{\boldsymbol{\times}}], M) \rightarrow_o$ (subst blocked shift)

$(S, H[^{a}(= {}^{b}(\ )^{v}\ bind\ {}^{arg}(\ )\ shift\ f\ sb)^{v+shift}], M)$

if $v > bind$

$(\mathcal{S}\ (copy, orig)\ (bind, arg, shift, f, sb) : S, H[^{orig}_{j}(\_)], M) \rightarrow_o$ (subst bind)

$(S, H[^{copy,arg}(\ )], M),$ if $j = bind$

$(\mathcal{S}\ (copy, orig)\ (bind, arg, shift, f, sb) : S, H[^{orig}_{j}(\_)^{v}], M) \rightarrow_o$ (subst var)

$(S, H[^{copy}_{j+shift}(\_)^{v+shift}], M)$

$(\mathcal{S}\ (copy, orig)\ (bind, arg, shift, f, sb) : S,$ (subst app)

$H[^{orig}_{j}(@\ {}^{a}(\ )\ {}^{b}(\ ))], M[f \mapsto m]) \rightarrow_o$

$(S, H[^{copy}_{j+shift}(@\ \ {}^{c}_{j}(= {}^{a}(\ )\ bind\ {}^{arg}(\ )\ shift\ f\ sb)^{\boldsymbol{\times}}$

$\qquad\qquad\qquad\qquad {}^{d}_{j}(= {}^{b}(\ )\ bind\ {}^{arg}(\ )\ shift\ f\ sb)^{\boldsymbol{\times}})^{\boldsymbol{\times}}], M[f \mapsto m[orig \mapsto copy]])$

where $c, d \notin dom(H)$

$(\mathcal{S}\ (copy, orig)\ (bind, arg, shift, f, sb) : S,$ (subst abs)

$H[^{orig}_{j}(\lambda\ {}^{a}(\ ))], M[f \mapsto m]) \rightarrow_o$

$(S, H[^{copy}_{j+shift}(\lambda\ {}^{b}_{j+1}(= {}^{a}(\ )\ bind\ {}^{arg}(\ )\ shift\ f\ sb)^{\boldsymbol{\times}})^{\boldsymbol{\times}}], M[f \mapsto m[orig \mapsto copy]])$

where $b \notin dom(H)$

$(\mathcal{S}\ (copy, orig)\ (bind, arg, shift, f, sb) : S,$ (subst pair)

$H[^{orig}_{j}(:\ ^{a}(\ )\ ^{b}(\ ))], M[f \mapsto m]) \rightarrow_o$

$(S, H[^{copy}_{j+shift}(:\quad ^{c}_{j}(=\ ^{a}(\ )\ bind\ ^{arg}(\ )\ shift\ f\ sb)^{\boldsymbol{\times}}$

$\qquad\qquad\qquad\qquad ^{d}_{j}(=\ ^{b}(\ )\ bind\ ^{arg}(\ )\ shift\ f\ sb)^{\boldsymbol{\times}})^{\boldsymbol{\times}}], M[f \mapsto m[orig \mapsto copy]])$

where $c, d \notin dom(H)$

$(\mathcal{S}\ (copy, orig)\ (bind, arg, shift, f, sb) : S,$ (subst atom)

$H[^{orig}_{j}(\#\ x)], M[f \mapsto m]) \rightarrow_o$

$(S, H[^{copy}_{j+shift}(\#\ x)^{\boldsymbol{\times}}], M[f \mapsto m[orig \mapsto copy]])$

$(\mathcal{S}\ (copy, orig)\ (bind, arg, shift, f, sb) : S,$ (subst prim)

$H[^{orig}_{j}(\Delta_n\ p\ ^{a_1}(\ )...^{a_n}(\ ))], M[f \mapsto m]) \rightarrow_o$

$(S, H[^{copy}_{j+shift}(\Delta_n\ p\ ^{b_1}_{j}(=\ ^{a_1}(\ )\ bind\ ^{arg}(\ )\ shift\ f\ sb)^{\boldsymbol{\times}}$

$\qquad\qquad\qquad ...\ ^{b_n}_{j}(=\ ^{a_n}(\ )\ bind\ ^{arg}(\ )\ shift\ f\ sb)^{\boldsymbol{\times}})^{\boldsymbol{\times}}], M[f \mapsto m[orig \mapsto copy]])$

where $b_1, ...\ b_n \notin dom(H)$

$(\mathcal{S}\ (copy, orig)\ (bind, arg, shift, f, sb) : S,$ (subst subst)

$H[^{orig}_{i}(=\ ^{a}(\ )\ bind'\ ^{arg'}(\ )\ shift'\ f'\ sb')], M[f \mapsto m]) \rightarrow_o$

$(S, H[\ ^{copy}_{i+shift}(=\ ^{b}(\ )\ (bind' + shift)\ ^{arg'}(\ )\ shift'\ f'\ (f : sb'))^{\boldsymbol{\times}}$

$\qquad ^{b}_{i+shift-shift'}(=\ ^{a}(\ )\ bind\ ^{arg}(\ )\ shift\ f\ sb)^{\boldsymbol{\times}}],$

$M[f \mapsto m[orig \mapsto copy]])$

where $b \notin dom(H), \qquad$ if $bind < bind'$

$(\mathcal{S}\ (copy, orig)\ (bind, arg, shift, f, sb) : S,$ (unsubst subst)

$H[^{orig}_{i}(=\ ^{a}(\ )\ bind'\ ^{arg'}(\ )\ shift'\ f'\ sb')], M[f \mapsto m]) \rightarrow_o$

$(S, H[\ ^{copy}_{i+shift}(=\ ^{b}(\ )\ bind'\ ^{arg'}(\ )\ shift'\ f'\ sb')^{\boldsymbol{\times}}$

$\qquad ^{b}_{i+shift-shift'}(=\ ^{a}(\ )\ (bind - shift')\ ^{arg}(\ )\ shift\ f\ (delete\ f'\ sb))^{\boldsymbol{\times}}],$

$M[f \mapsto m[orig \mapsto copy]])$

where $b \notin dom(H), \qquad$ if $bind \geq bind'$

$(\mathcal{C}\ (copy, substs, orig) : S,$ (collected enough)

$H[^{orig}(=\ ^{body}(\ )\ bind\ ^{arg}(\ )\ shift\ f\ sb)], M) \rightarrow_o$

$(\mathcal{R}\ (copy, substs, orig) : S, H, M), \qquad$ if $\mathcal{F}\ (substs, (bind, shift, f, sb))$

$(\mathcal{C}\ (copy, substs, orig) : S,$ (collect more)

$H[^{orig}(=\ ^{body}(\ )\ bind\ ^{arg}(\ )\ shift\ f\ sb)], M) \rightarrow_o$

$(\mathcal{C}\ (copy, (bind, shift, arg, f, sb) : substs, body) : S, H, M), \qquad$ otherwise

$(\mathcal{C}\ (copy, substs, orig) : S, H[^{orig}(\ )], M) \rightarrow_o$ (collected all)

$(\mathcal{R}\ (copy, substs, orig) : S, H, M)$

$(\mathcal{R}\ (copy, [(bind, arg, shift, f, sb)], orig) : S, H, M) \ \rightarrow_o$ (rebuild finished)
$\quad (\mathcal{S}\ (copy, orig)\ (bind, arg, shift, f, sb) : S, H, M)$

$(\mathcal{R}\ (copy, (bind, arg, shift, f, sb) : substs, orig) : S, H, M) \ \rightarrow_o$ (rebuild more)
$\quad (\mathcal{S}\ (new, orig)\ (bind, arg, shift, f, sb) : \mathcal{R}\ (copy, substs, new) : S, H, M)$
$\quad$ where $new \notin dom(H)$

$\mathcal{F}\ ([], \_) = true$ (test passed)

$\mathcal{F}\ ((bind, \_, shift, f, sb) : substs,\ (bind', shift', f', sb')) =$ (test subst)
$\quad \mathcal{F}\ (substs, (bind' + shift, shift', f', f : sb')), \qquad$ if $bind < bind'$

$\mathcal{F}\ ((bind, \_, shift, f, sb) : substs,\ (bind', shift', f', sb')) =$ (test unsubst)
$\quad \mathcal{F}\ (substs, (bind', shift', f', sb')), \qquad$ if $f \in sb'$

$\mathcal{F}\ ((bind, \_, shift, f, sb) : substs,\ (bind', shift', f', sb')) =$ (test failed)
$\quad false, \qquad$ otherwise

$(a : S, H[^a(\texttt{+}\ ^{\cdots,b}(\#\ x)^{\checkmark}\ ^{\cdots,c}(\#\ y)^{\checkmark})^{\times}], M)$ (prim + reduced)
$\quad \rightarrow_o\ (S, H[^a_0(\#\ (x + y))^{\checkmark}], M), \qquad$ if $x, y \in \mathbb{Z}$

$(a : S, H[^a(\texttt{+}\ ^{\cdots,b}(\ )^{\times}\ ^{\cdots,c}(\ ))^{\times}], M) \ \rightarrow_o\ (b : a : S, H, M)$ (prim + in-progress 1)

$(a : S, H[^a(\texttt{+}\ ^{\cdots,b}(\ )\ ^{\cdots,c}(\ )^{\times})^{\times}], M) \ \rightarrow_o\ (c : a : S, H, M)$ (prim + in-progress 2)

$(a : S, H[^a(\texttt{+}\ ^{\cdots,b}(\ )^{\checkmark}\ ^{\cdots,c}(\ )^{v})^{\times}], M) \ \rightarrow_o\ (S, H[^a(\ )^v], M)$ (prim + blocked 1)

$(a : S, H[^a(\texttt{+}\ ^{\cdots,b}(\ )^{v}\ ^{\cdots,c}(\ ))^{\times}], M) \ \rightarrow_o\ (S, H[^a(\ )^{\checkmark}], M)$ (prim + blocked 2)

$(a : S, H[^a(\texttt{if True}\ ^c(\ )\ ^d(\ ))], M)$ (prim `if True` reduced)
$\quad \rightarrow_o\ (a : S, H[^{a,c}(\ )], M)$

$(a : S, H[^a(\texttt{if False}\ ^c(\ )\ ^d(\ ))], M)$ (prim `if False` reduced)
$\quad \rightarrow_o\ (a : S, H[^{a,d}(\ )], M)$

$(a : S, H[^a(\texttt{if}\ ^{\cdots,b}(\ )^v\ ^c(\ )\ ^d(\ ))], M)$ (prim `if` blocked)
$\quad \rightarrow_o\ (S, H[^a(\ )^v], M)$

$(a : S, H[^a(\texttt{if}\ ^{\cdots,b}(\ )^{\times}\ ^c(\ )\ ^d(\ ))], M)$ (prim `if` in-progress)
$\quad \rightarrow_o\ (b : a : S, H, M)$

$(a : S, H[^a(\texttt{head}\ ^{\cdots,b}(:\ ^{hd}(\ )\ ^{tl}(\ ))^{\checkmark})], M)$ (prim `head` reduced)
$\quad \rightarrow_o\ (a : S, H[^{a,hd}(\ )], M)$

$(a : S, H[^a(\texttt{head}\ ^{\cdots,b}(\ )^{\times})], M) \ \rightarrow_o\ (b : a : S, H, M)$ (prim `head` in-progress)

$(a : S, H[^a(\texttt{head}\ ^{\cdots,b}(\ )^{v})], M) \ \rightarrow_o\ (S, H[^a(\ )^v], M)$ (prim `head` blocked)

□

The optimal (eval ...) reduction rules are almost identical to the completely lazy (eval ...) rules. However the rule (eval finished) is introduced, so as to draw a distinction between nodes where variables are blocking further reductions, and nodes where reduction is finished. The rule (eval var) differs in that variables are now given a blocked tag equal to their depth, rather than a ✓. Similarly blocked applications are tagged with the same blocked tag as the function part of an application, and not necessarily a ✓.

The *SubstBy* field of the substitution nodes is now used. Substitution nodes are created by the (eval beta) rule, where the *SubstBy* field is initially set to the empty list []. The *SubstBy* field is used as a bag, elements are added to it using the cons operator, as shown in (subst subst), a single occurrence of an element is removed using the *delete* function, as shown in (unsubst subst), and bag membership is tested with the set membership operator, $\in$, as shown in (test unsubst).

The (eval app blocked) rule applies when evaluation of the function part of an application has been attempted, but the function part did not reduce to an abstraction. Reduction of the function part may either be blocked on a variable, in which case the function part will be tagged with the depth of the variable it is blocked on. Alternatively the function part of the application may be fully reduced, in which case it will be tagged with a ✓. For the function part to be tagged with a ✓ but not be an abstraction indicates a type-error. In either case the blocked tag on the function part is propagated to the application.

The optimal (subst specialize), (subst scope-boundary) and (subst memoized) rules are just the same as their completely lazy counterparts.

The optimal reduction rules handle substitutions somewhat differently from the completely lazy reduction rules. As explained in §3.8 performing substitutions through graph which is blocked by a different variable from the one the substitution binds risks losing sharing. To avoid this loss of sharing, substitution nodes are no longer capable of substituting all nodes. Substitution nodes are now only capable of substituting nodes which are blocked on the variable the substitution binds, or nodes which are fully evaluated and so tagged with a ✓. When the body of a substitution is blocked on a variable other than the variable the substitution binds, the reduction rules (subst blocked) and (subst blocked shift) apply. These rules propagate the blocked tag from the body of the substitution to the substitu-

tion itself. The rule (subst blocked shift) renames the blocked tag in the process of propagating it. This is because the current substitution would rename the blocked variable if the current substitution substitutes the variable before the required substitution reaches the variable. The rule (subst blocked) does not rename the blocked tag, as in this case the current substitution would *die out* before ever reaching the blocked variable.

These blocked tags propagate up from the variable blocking reduction towards the root. When the blocked tags propagate up to the substitution which binds the variable blocking reduction, substitution can proceed. This binding substitution will then try to substitute through all the nodes between itself and the blocking variable until it binds a value for that variable, enabling further reduction. However, as explained in §3.9, substitutions cannot always substitute other substitution nodes. Substitutions only need be and should only be swapped (by substitution or *un*substitution) if they could have been formed (via an alternative reduction order) in the swapped order in the first place. If a substitution exists in-between a variable and the substitution that binds that variable, and the binding substitution is unable to swap with the in-between substitution, then the in-between substitution must be performed along with the binding substitution. To handle these in-between substitutions the stack commands $\mathcal{C}$, $\mathcal{R}$, $\mathcal{S}$ and the function $\mathcal{F}$ are introduced.

The stack command $\mathcal{C}$ is used to *collect* a sequence of substitutions comprising the binding substitution and all the in-between substitutions which the sequence is unable to swap with.

Whether or not a sequence of substitutions is able to swap with a substitution is determined by the function $\mathcal{F}$. Two substitutions can be swapped by substitution if the upper substitution binds a shallower variable than the lower substitution. Two substitutions can be swapped by unsubstitution if there is a record in the *SubstBy* field of the upper substitution of the lower substitution having substituted the upper substitution. A sequence of substitutions can be swapped with a substitution if the inner most substitution of the sequence can be swapped with the substitution and then if in-turn each next inner-most substitution can be swapped with the result of the previous swap. If the sequence of substitutions is able to swap with a substitution then the sequence has been built up *far enough*, and the function $\mathcal{F}$ will return *true*. The reduction rule (collected enough) then applies and the stack command $\mathcal{R}$ is invoked (explained later).

If the function $\mathcal{F}$ returns *false* then the reduction rule (collect more) applies and the latest substitution is added to the sequence of substitutions and the stack command $\mathcal{C}$ is invoked again.

If the body of the inner most substitution in the sequence does not point to a substitution then the reduction rule (collected all) applies and the stack command $\mathcal{R}$ is invoked.

The stack command $\mathcal{R}$ is used to *rebuild* a sequence of substitutions once the sequence has been extended far enough that the sequence can swap with body of the inner most substitution of the sequence. This body will either be a substitution node, if $\mathcal{R}$ was invoked by (collected enough), or a non-substitution node, if $\mathcal{R}$ was invoked by (collected all).

The stack command $\mathcal{R}$ repeatedly invokes the stack command $\mathcal{S}$ to perform each individual substitution. It is only the outer-most substitution (the binding substitution), which is overwritten by the result of these substitutions. The results of the intermediate substitutions are created at fresh addresses.

The substitution of fully evaluated nodes (tagged with a ✓) is initiated by the (subst node) rule. This rule pushes the substitution command $\mathcal{S}$ onto the stack. Fully evaluated nodes, such as abstraction and pair, can be substituted by any substitution without loss of sharing.

The reduction rule (subst found) applies when the blocked tag from a variable blocking further reduction has propagated outward and reached the substitution that can bind a value for this variable. This rule places the *collect* substitutions command $\mathcal{C}$ on the stack, and initiates the collection of substitutions described above.

The optimal rules (subst bind), (subst var), (subst app), (subst abs), (subst pair), (subst atom) and (subst prim) are much the same as their completely lazy counter parts. The difference being the use of the substitution command, $\mathcal{S}$, on the stack.

The rules (subst subst) and (unsubst subst) are used to swap substitutions. The rule (subst subst) applies when the outer substitution binds a shallower variable than the inner substitution. The rule (unsubst subst) applies otherwise. The rule (subst subst) uses the outer substitution to rename (what was) the inner substitution, and places a record of the substitution in the *SubstBy* field of the new outer substitution. The rule (unsubst subst) undoes this by using the inner substitution to *un*rename what was the outer substitution, and removes the record of the prior substitution

from the *SubstBy* field of what was the outer substitution.

In the case of primitives with more than one strict argument, such as (prim +), where the blocked tags of the strict arguments may differ, it is the blocked tag of the earliest strict argument that is not a ✓ that is propagated. If all the strict arguments of a primitive node are tagged with ✓ and yet the primitive cannot be reduced, then a type-error has occurred and the primitive node is tagged with a ✓.

## 4.8 Examples

This section demonstrates the optimal evaluation reduction rules used by *Ef*. Reduction of the term `(two two I I)` is used as a running example. The memo-tables only change by growing, once an entry has been made it will not be changed or removed. Instead of repeatedly showing this accumulating data in its entirety, only new entries will be shown.

Reduction begins in the following state:

stack: [1]

heap: ${}^{1}_{0}(@\ {}^{2}_{0}(@\ {}^{3}_{0}(@\ {}^{4}(\ )\ {}^{4}(\ ))^{\times}\ {}^{10}(\ ))^{\times}\ {}^{10}(\ ))^{\times}$
${}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{\times}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{\times})^{\times})^{\times}$
${}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

memo-tables: {}

The first four reductions to be performed are three (eval app) reductions followed by a (eval abs) reductions. The (eval app) reductions traverse the spine of the expression instantiated in the heap pushing 2, 3 and 4 onto the stack. The (eval abs) reduction tags the abstraction node at address 4 with a ✓ and pops this address from the stack.

stack: [3,2,1]

heap: ${}^{1}_{0}(@\ {}^{2}_{0}(@\ {}^{3}_{0}(@\ {}^{4}(\ )\ {}^{4}(\ ))^{\times}\ {}^{10}(\ ))^{\times}\ {}^{10}(\ ))^{\times}$
${}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{\times}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{\times})^{\times})^{\checkmark}$
${}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

memo-tables: {}

The reduction (eval beta) now updates the node at address 3 with a substitution node, and creates an empty memo-table.

stack: [3,2,1]

heap: ${}^{1}_{0}(@\ {}^{2}_{0}(@\ {}^{3}(\ )\ {}^{10}(\ ))^{\times}\ {}^{10}(\ ))^{\times}$

$\quad {}^{3}_{0}(=\ {}^{5}(\ )\ 1\ {}^{4}(\ )\ \text{-}1\ 1\ [])^{\times}$

$\quad\quad {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{\times}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{\times})^{\times}$

$\quad\quad {}^{4}_{0}(\lambda\ {}^{5}(\ ))^{\checkmark}$

$\quad {}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

memo-tables: $\{\ 1 \mapsto \varnothing\ \}$

The reduction (subst specialize) will push the address of the body of the substitution on to the stack. The reduction (eval abs) tags the abstraction node at address 5 with a ✓. Now that the body of the substitution is tagged with a ✓ the reduction (subst node) can be performed. This pushes the command ($\mathcal{S}$ (3,5) (1,4,-1,1,[])) onto the stack.

stack: [$\mathcal{S}$ (3,5) (1,4,-1,1,[]),3,2,1]

heap: ${}^{1}_{0}(@\ {}^{2}_{0}(@\ {}^{3}(\ )\ {}^{10}(\ ))^{\times}\ {}^{10}(\ ))^{\times}$

$\quad {}^{3}_{0}(=\ {}^{5}(\ )\ 1\ {}^{4}(\ )\ \text{-}1\ 1\ [])^{\times}$

$\quad\quad {}^{5}_{1}(\lambda\ {}^{6}(\ ))^{\checkmark}$

$\quad\quad\quad {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{\times}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{\times}$

$\quad\quad {}^{4}_{0}(\lambda\ {}^{5}(\ ))^{\checkmark}$

$\quad {}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

memo-tables: {}

The command ($\mathcal{S}$ (3,5) (1,4,-1,1,[])) triggers the rule (subst abs) which performs the substitution.

stack: [3,2,1]

heap: ${}^{1}_{0}(@\ {}^{2}_{0}(@\ {}^{3}(\ )\ {}^{10}(\ ))^{\times}\ {}^{10}(\ ))^{\times}$

$\quad {}^{3,26}_{0}(\lambda\ {}^{25}(\ ))^{\times}$

$\quad\quad {}^{25}_{1}(=\ {}^{6}(\ )\ 1\ {}^{4}(\ )\ \text{-}1\ 1\ [])^{\times}$

$\quad\quad\quad {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{\times}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{\times}$

$\quad\quad\quad {}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}(\ ))^{\checkmark})^{\checkmark}$

$\quad {}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

memo-tables: $\{\ 1 \mapsto \{5 \mapsto 3\}\ \}$

The reduction (eval beta) now tags the abstraction node at address 2 with a ✓, and the reduction (eval beta) creates another substitution node.

stack: [2,1]

heap: ${}^{1}_{0}(@\ {}^{2}(\ )\ {}^{10}(\ ))^{\times}$

${}^{2}_{0}(=\ {}^{25}(\ )\ 1\ {}^{10}(\ )\ \text{-}1\ 2\ [])^{\times}$

${}^{25}_{1}(=\ {}^{6}(\ )\ 1\ {}^{4}(\ )\ \text{-}1\ 1\ [])^{\times}$

${}^{6}_{2}(@\ {}^{7}_{1}(\_)^{\times}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{\times}$

${}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}(\ ))^{\checkmark})^{\checkmark}$

${}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

memo-tables: $\{\ 2 \mapsto \varnothing\ \}$

The reduction rule (subst specialize) will traverse the graph through the substitutions at addresses 2 and 25. The reduction rule (eval app) will traverse through the application node at address 6. The reduction rule (eval var) will update the tag on the variable at address 7 from a ✗ to a 1, to indicate that this node requires a substitution binding variables at depth 1.

stack: [6,25,2,1]

heap: ${}^{1}_{0}(@\ {}^{2}(\ )\ {}^{10}(\ ))^{\times}$

${}^{2}_{0}(=\ {}^{25}(\ )\ 1\ {}^{10}(\ )\ \text{-}1\ 2\ [])^{\times}$

${}^{25}_{1}(=\ {}^{6}(\ )\ 1\ {}^{4}(\ )\ \text{-}1\ 1\ [])^{\times}$

${}^{6}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{\times}$

${}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}(\ ))^{\checkmark})^{\checkmark}$

${}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

memo-tables: {}

The reduction rule (eval app blocked) now copies the tag from the node blocking further reduction, to indicate that it also requires a substitution binding variables at depth 1. The reduction rule (subst found) then applies as the substitution at address 25 binds the variable that the body of the substitution requires. This substitution node pushes the command ($\mathcal{C}$ (25,[(1,4,-1,1,[])],6)) onto the stack.

stack: [$\mathcal{C}$ (25,[(1,4,-1,1,[])],6),25,2,1]

heap: ${}^{1}_{0}(@\ {}^{2}(\ )\ {}^{10}(\ ))^{\times}$

${}^{2}_{0}(=\ {}^{25}(\ )\ 1\ {}^{10}(\ )\ \text{-}1\ 2\ [])^{\times}$

${}^{25}_{1}(=\ {}^{6}(\ )\ 1\ {}^{4}(\ )\ \text{-}1\ 1\ [])^{\times}$

${}^{6}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{1}$

${}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}(\ ))^{\checkmark})^{\checkmark}$

${}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

memo-tables: {}

The $\mathcal{C}$ command is used to collect up all the substitutions between the substitution that has been found to be needed, and the first node that it can substitute (or unsubstitute). In this case the substitution can be performed straight away.

stack: [25,2,1]

heap: ${}^{1}_{0}(@\ {}^{2}(\ )\ {}^{10}(\ ))^{\times}$
${}^{2}_{0}(=\ {}^{25}(\ )\ 1\ {}^{10}(\ )\ {-1}\ 2\ [])^{\times}$
${}^{25,29}_{1}(@\ {}^{27}(\ )\ {}^{28}(\ ))^{\times}$
${}^{27}_{1}(=\ {}^{7}(\ )\ 1\ {}^{4}(\ )\ {-1}\ 1\ [])^{\times}$
${}^{7}_{1}(\_)^{1}$
${}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}(\ )\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{1})^{\checkmark})^{\checkmark}$
${}^{28}_{1}(=\ {}^{8}(\ )\ 1\ {}^{4}(\ )\ {-1}\ 1\ [])^{\times}$
${}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

memo-tables: $\{\ 1 \mapsto \{6 \mapsto 25\}\ \}$

The reduction (subst var) can now be applied as the substitution at address 27 has met the variable it wishes to bind. The substitution is overwritten with an indirection pointing to the argument of the substitution.

stack: [27,25,2,1]

heap: ${}^{1}_{0}(@\ {}^{2}(\ )\ {}^{10}(\ ))^{\times}$
${}^{2}_{0}(=\ {}^{25}(\ )\ 1\ {}^{10}(\ )\ {-1}\ 2\ [])^{\times}$
${}^{25,29}_{1}(@\ {}^{27}(\ )\ {}^{28}(\ ))^{\times}$
${}^{27,4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{1})^{\checkmark})^{\checkmark}$
${}^{28}_{1}(=\ {}^{8}(\ )\ 1\ {}^{4}(\ )\ {-1}\ 1\ [])^{\times}$
${}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

memo-tables: {}

The application at address 25 can now be reduced by reduction rule (eval beta).

stack: [25,2,1]

heap: ${}^{1}_{0}(@\ {}^{2}(\ )\ {}^{10}(\ ))^{\times}$
${}^{2}_{0}(=\ {}^{25}(\ )\ 1\ {}^{10}(\ )\ {-1}\ 2\ [])^{\times}$
${}^{25,29}_{1}(=\ {}^{5}(\ )\ 1\ {}^{28}(\ )\ 0\ 3\ [])^{\times}$
${}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{1})^{\checkmark}$
${}^{28}_{1}(=\ {}^{8}(\ )\ 1\ {}^{4}(\ )\ {-1}\ 1\ [])^{\times}$

$$\begin{array}{l} \quad {}^{4}_{0}(\lambda\ {}^{5}(\ ))^{\checkmark} \\ {}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times} \end{array}$$

memo-tables: { 3 ↦ ∅ }

The substitution at address 25 is substituted through the abstraction node at address 5. The resulting abstraction node is then substituted by the substitution at address 2.

stack: [2,1]

$$\begin{array}{ll} \text{heap:} & {}^{1}_{0}(@\ {}^{2}(\ )\ {}^{10}(\ ))^{\times} \\ & \quad {}^{2,33}_{0}(\lambda\ {}^{32}(\ ))^{\times} \\ & \quad\quad {}^{32}_{1}(=\ {}^{30}(\ )\ 1\ {}^{10}(\ )\ {-1}\ 2\ [])^{\times} \\ & \quad\quad\quad {}^{30}_{2}(=\ {}^{6}(\ )\ 1\ {}^{28}(\ )\ 0\ 3\ [])^{\times} \\ & \quad\quad\quad\quad {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}_{2}(\_)^{\times})^{\times})^{1} \\ & \quad\quad\quad\quad {}^{28}_{1}(=\ {}^{8}(\ )\ 1\ {}^{4}(\ )\ {-1}\ 1\ [])^{\times} \\ & \quad\quad\quad\quad\quad {}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}(\ ))^{\checkmark})^{\checkmark} \\ & \quad\quad {}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times} \end{array}$$

memo-tables: { 3 ↦ {5 ↦ 31}, 2 ↦ {31 ↦ 33} }

Reduction continues in this fashion. Rather than show every reduction in repetitive tedious detail, a selection of the more interesting cases are shown next.

stack: [32,1]

$$\begin{array}{ll} \text{heap:} & {}^{1}_{0}(=\ {}^{32}(\ )\ 1\ {}^{10}(\ )\ {-1}\ 4\ [])^{\times} \\ & \quad {}^{32}_{1}(=\ {}^{30}(\ )\ 1\ {}^{10}(\ )\ {-1}\ 2\ [])^{\times} \\ & \quad\quad {}^{30,36}_{2}(=\ {}^{40}(\ )\ 2\ {}^{35}(\ )\ 0\ 6\ [])^{1} \\ & \quad\quad\quad {}^{40,44}_{2}(@\ {}^{42}(\ )\ {}^{43}(\ ))^{1} \\ & \quad\quad\quad\quad {}^{42,38}_{1}(=\ {}^{9}(\ )\ 1\ {}^{4}(\ )\ {-1}\ 1\ [])^{1} \\ & \quad\quad\quad\quad\quad {}^{9}_{2}(\_)^{2} \\ & \quad\quad\quad\quad\quad {}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}(\ ))^{1})^{1})^{\checkmark})^{\checkmark} \\ & \quad\quad\quad\quad {}^{43}_{2}(=\ {}^{8}(\ )\ 1\ {}^{38}(\ )\ 0\ 5\ [])^{\times} \\ & \quad\quad {}^{35}_{2}(=\ {}^{8}(\ )\ 1\ {}^{28}(\ )\ 0\ 3\ [])^{\times} \\ & \quad\quad\quad {}^{28,41}_{1}(\lambda\ {}^{40}(\ ))^{\checkmark} \\ & \quad {}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times} \end{array}$$

memo-tables: { }

At this point the variable at address 9 is preventing further reduction. A substitution binding a variable at depth 2 is required. The substitution at address 38

binds a variable at the wrong depth, so substituting this substitution through the variable will not aid further reduction. This substitution at address 38 has a shift of -1, so if it were to substitute the variable at depth 2, the result would be a variable at depth 1. For this reason this substitution at address 38 is tagged with a 1 to indicate that a substitution binding a variable at depth 1 is required to aid further reduction. This tagging propagates upwards to the application node at address 44, and the substitution node at address 36 (with a shift of 0). Finally, the substitution at address 32 is found. This substitution is required to enable further reduction.

The reduction rule (subst found) is applied. This places the command ($\mathcal{C}$ (32,[(1,10,-1,2,[])],30)) on the stack. The reduction rule (collected enough) is applied as the function call $\mathcal{F}$ ([(1,10,-1,2,[])],(2,0,6,[])) indicates substitutions can be swapped. A record of the substitution is kept in the *SubstBy* field of the substituted substitution.

stack: [($\mathcal{C}$ (32,[(1,10,-1,2,[])],30)), 32,1]

stack: [($\mathcal{R}$ (32,[(1,10,-1,2,[])],30)), 32,1]

stack: [($\mathcal{S}$ (46,30) (1,10,-1,2,[]), 32,1]

stack: [32,1]

heap: ${}^{1}_{0}(= {}^{32}( ) \; 1 \; {}^{10}( ) \; \text{-1} \; 4 \; [])^{\times}$

${}^{32,46}_{1}(= {}^{45}( ) \; 1 \; {}^{35}( ) \; 0 \; 6 \; [2])^{\times}$

${}^{45}_{1}(= {}^{40}( ) \; 1 \; {}^{10}( ) \; \text{-1} \; 2 \; [])^{\times}$

${}^{40,44}_{2}(@ \; {}^{42}( ) \; {}^{43}( ))^{1}$

${}^{42,38}_{1}(= {}^{9}_{2}(\_)^{2} \; 1 \; {}^{4}( ) \; \text{-1} \; 1 \; [])^{1}$

${}^{4}_{0}(\lambda \; {}^{5}_{1}(\lambda \; {}^{6}_{2}(@ \; {}^{7}_{1}(\_)^{1} \; {}^{8}_{2}(@ \; {}^{7}( ) \; {}^{9}( ))^{1})^{1})^{\checkmark})^{\checkmark}$

${}^{43}_{2}(= {}^{8}( ) \; 1 \; {}^{38}( ) \; 0 \; 5 \; [])^{\times}$

${}^{10}_{0}(\lambda \; {}^{11}_{1}(\_)^{\times})^{\times}$

${}^{35}_{2}(= {}^{8}( ) \; 1 \; {}^{28}( ) \; 0 \; 3 \; [])^{\times}$

${}^{28,41}_{1}(\lambda \; {}^{40}( ))^{\checkmark}$

memo-table: $\{ \; 2 \mapsto \{30 \mapsto 46\} \; \}$

The substitution which will bind a value to the variable at address 9 is now one step closer. It substitutes the application node at address 44. Now only one node separates the variable blocking further reduction and the substitution able bind a value to that variable. This last remaining node is another substitution node. This time the substitutions cannot be swapped. The reduction rule (collected all) applies.

stack: [47,45,32,1]

heap: ${}^{1}_{0}(= {}^{32}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 4\ [])^{\times}$

${}^{32,46}_{1}(= {}^{45}(\ )\ 1\ {}^{35}(\ )\ 0\ 6\ [2])^{\times}$

${}^{45,49}_{1}(@\ {}^{47}(\ )\ {}^{48}(\ ))^{\times}$

${}^{47}_{1}(= {}^{42}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 2\ [])^{\times}$

${}^{42,38}_{1}(= {}^{9}(\ )\ 1\ {}^{4}(\ )\ {-}1\ 1\ [])^{1}$

${}^{9}_{2}(\_)^{2}$

${}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}(\ ))^{1})^{1})^{\checkmark})^{\checkmark}$

${}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

${}^{48}_{1}(= {}^{43}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 2\ [])^{\times}$

${}^{43}_{2}(= {}^{8}(\ )\ 1\ {}^{38}(\ )\ 0\ 5\ [])^{\times}$

${}^{35}_{2}(= {}^{8}(\ )\ 1\ {}^{28}(\ )\ 0\ 3\ [])^{\times}$

${}^{28,41}_{1}(\lambda\ {}^{40,44}_{2}(@\ {}^{42}(\ )\ {}^{43}(\ ))^{1})^{\checkmark}$

The stack goes through the following transitions:

stack: [($\mathcal{C}$ (47, [(1,10,-1,2,[])], 42), 47,45,32,1]

stack: [($\mathcal{C}$ (47, [(1,10,-1,2,[]),(1,4,-1,1,[])], 9), 47,45,32,1]

stack: [($\mathcal{R}$ (47, [(1,4,-1,1,[]),(1,10,-1,2,[])], 9), 47,45,32,1]

stack: [($\mathcal{S}$ (50,9) (1,4,-1,1,[])), ($\mathcal{R}$ (47, [(1,10,-1,2,[])], 50), 47,45,32,1]

stack: [($\mathcal{R}$ (47, [(1,10,-1,2,[])], 50), 47,45,32,1]

stack: [($\mathcal{S}$ (47, [(1,10,-1,2,[])], 50), 47,45,32,1]

stack: [47,45,32,1]

heap: ${}^{1}_{0}(= {}^{32}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 4\ [])^{\times}$

${}^{32,46}_{1}(= {}^{45}(\ )\ 1\ {}^{35}(\ )\ 0\ 6\ [2])^{\times}$

${}^{45,49}_{1}(@\ {}^{47}(\ )\ {}^{48}(\ ))^{\times}$

${}^{47,10}_{0}(\lambda\ {}^{11}_{1}(\_)^{\times})^{\times}$

${}^{48}_{1}(= {}^{43}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 2\ [])^{\times}$

${}^{43}_{2}(= {}^{8}(\ )\ 1\ {}^{38}(\ )\ 0\ 5\ [])^{\times}$

${}^{8}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{9}_{2}(\_)^{2})^{1}$

${}^{38}_{1}(= {}^{9}(\ )\ 1\ {}^{4}(\ )\ {-}1\ 1\ [])^{1}$

${}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}(\ )\ {}^{8}(\ ))^{1})^{\checkmark})^{\checkmark}$

${}^{35}_{2}(= {}^{8}(\ )\ 1\ {}^{28}(\ )\ 0\ 3\ [])^{\times}$

${}^{28,41}_{1}(\lambda\ {}^{40,44}_{2}(@\ {}^{42,38}(\ )\ {}^{43}(\ ))^{1})^{\checkmark}$

Later two substitutions swapping by unsubstitution can be seen.

stack: [32,1]

heap: $^{1}_{0}(= {}^{32}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 4\ [])^{\times}$

$^{32,46}_{1}(= {}^{45}(\ )\ 1\ {}^{35}(\ )\ 0\ 6\ [2])^{\times}$

$^{45,49,55}_{1}(= {}^{52}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 2\ [])^{1}$

$^{52}_{2}(= {}^{9}(\ )\ 1\ {}^{38}(\ )\ 0\ 5\ [])^{2}$

$^{9}_{2}(\_)^{2}$

$^{38}_{1}(= {}^{9}(\ )\ 1\ {}^{4}(\ )\ {-}1\ 1\ [])^{1}$

$^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}(\ ))^{1})^{1})^{\checkmark})^{\checkmark}$

$^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{1})^{\checkmark}$

$^{35}_{2}(= {}^{8}(\ )\ 1\ {}^{28}(\ )\ 0\ 3\ [])^{\times}$

$^{28,41}_{1}(\lambda\ {}^{40,44}_{2}(@\ {}^{42,38}(\ )\ {}^{43,53}_{2}(@\ {}^{51,38}(\ )\ {}^{52}(\ ))^{1})^{1})^{\checkmark}$

Here the variable at address 9 is blocking further reduction and the substitution at address 46 is able to bind a value for that variable.

The stack goes through the following states:

stack: [($\mathcal{C}$ (32,[(1,35,0,6,[2])],45)), 32,1]

stack: [($\mathcal{R}$ (32,[(1,35,0,6,[2])],45)), 32,1]

stack: [($\mathcal{S}$ (32,45) (1,35,0,6,[2])), 32,1]

stack: [32,1]

heap: $^{1}_{0}(= {}^{32}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 4\ [])^{\times}$

$^{32,46,58}_{1}(= {}^{57}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 2\ [])^{\times}$

$^{57}_{2}(= {}^{52}(\ )\ 2\ {}^{35}(\ )\ 0\ 6\ [])^{\times}$

$^{52}_{2}(= {}^{9}(\ )\ 1\ {}^{38}(\ )\ 0\ 5\ [])^{2}$

$^{9}_{2}(\_)^{2}$

$^{38}_{1}(= {}^{9}(\ )\ 1\ {}^{4}(\ )\ {-}1\ 1\ [])^{1}$

$^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}(\ ))^{1})^{1})^{\checkmark})^{\checkmark}$

$^{35}_{2}(= {}^{8}(\ )\ 1\ {}^{28}(\ )\ 0\ 3\ [])^{\times}$

$^{28,41}_{1}(\lambda\ {}^{40,44}_{2}(@\ {}^{42,38}(\ )\ {}^{43,53}_{2}(@\ {}^{51,38}(\ )\ {}^{52}(\ ))^{1})^{1})^{\checkmark}$

$^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{1})^{\checkmark}$

Finally the last reduction steps demonstrate a sequence of three substitutions.

stack: [1]

heap: $^{1}_{0}(= {}^{32}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 4\ [])^{\times}$

$^{32,46,66}_{1}(= {}^{65}(\ )\ 1\ {}^{10}(\ )\ {-}1\ 2\ [])^{1}$

$$
\begin{array}{l}
{}^{65,61}_{2}(= {}^{9}(\ )\ 1\ {}^{28}(\ )\ 0\ 3\ [])^{2} \\
\quad {}^{9}_{2}(\_)^{2} \\
\quad {}^{28,41}_{1}(\lambda\ {}^{40,44}_{2}(@\ {}^{42}(\ )\ {}^{43}(\ ))^{1})^{\checkmark} \\
\qquad {}^{42,38}_{1}(= {}^{9}(\ )\ 1\ {}^{4}(\ )\ \text{-}1\ 1\ [])^{1} \\
\qquad\quad {}^{4}_{0}(\lambda\ {}^{5}_{1}(\lambda\ {}^{6}_{2}(@\ {}^{7}_{1}(\_)^{1}\ {}^{8}_{2}(@\ {}^{7}(\ )\ {}^{9}(\ ))^{1})^{1})^{\checkmark})^{\checkmark} \\
\qquad {}^{43,53}_{2}(@\ {}^{51,38}(\ )\ {}^{52}(\ ))^{1} \\
\qquad\quad {}^{52}_{2}(= {}^{9}(\ )\ 1\ {}^{38}(\ )\ 0\ 5\ [])^{2} \\
{}^{10}_{0}(\lambda\ {}^{11}_{1}(\_)^{1})^{\checkmark}
\end{array}
$$

Further reduction is blocked by the variable at address 9, the substitution at address 1 is able to bind a value to this variable. The binding substitution is unable to swap with the two in-between substitutions, so these substitutions are performed along with the binding substitution. Since the node being substituted is variable, the effect of performing the two in-between substitutions is to rename the variable.

The stack goes through the following states:

stack: [($\mathcal{C}$ (1,[(1,10,-1,4,[])],32)), 1]

stack: [($\mathcal{C}$ (1,[(1,10,-1,2,[]),(1,10,-1,4,[])],65)), 1]

stack: [($\mathcal{C}$ (1,[(1,28,0,3,[]),(1,10,-1,2,[]),(1,10,-1,4,[])],9)), 1]

stack: [($\mathcal{R}$ (1,[(1,28,0,3,[]),(1,10,-1,2,[]),(1,10,-1,4,[])],9)), 1]

stack: [($\mathcal{S}$ (67,(1,28,0,3,[]),9),($\mathcal{R}$ (1,(1,10,-1,2,[]),(1,10,-1,4,[])],67)), 1]

stack: [($\mathcal{R}$ (1,[(1,10,-1,2,[]),(1,10,-1,4,[])],67)), 1]

stack: [($\mathcal{S}$ (68,(1,10,-1,2,[]),67)), ($\mathcal{R}$ (1,[(1,10,-1,4,[])],68)), 1]

stack: [($\mathcal{R}$ (1,[(1,10,-1,4,[])],68)), 1]

stack: [($\mathcal{S}$ (1,(1,10,-1,4,[]),68)), 1]

stack: [1]

And the reduction is complete, the result of evaluating (`two two I I`) is the identity function, as expected:

stack: [1]

heap: ${}^{1,10}_{0}(\lambda\ {}^{11}_{1}(\_)^{1})^{\checkmark}$

## 4.9 Discussion

The reduction rules are not as concise as could be hoped for. One of the reasons for this is that the reduction rules mirror the implementation in that the evaluation rules and substitution rules are kept separate.

In much the same way that the introduction of a graph sharing, results in reduction rules more complicated than term rewriting rules, the memo-tables introduce additional complexity. Perhaps some improvement over the current scheme can be found.

However, a graphical notation such as is used to describe interaction nets [53, 61] would be inappropriate as the transformations performed are not strictly local graph transformations. A term graph rewriting approach [27, 19] would be inappropriate due to the presence of memo-tables. The syntactic brackets used to specify many program semantics and transformations [64, 73, 72], would be inappropriate as the reduction rules apply to graphs not trees.

## 4.10 Summary

This chapter has presented the reduction rules for completely lazy evaluation and optimal evaluation. The formalism used enables the reduction rules to be explicitly named.

The examples reveal just how small the reduction steps are and just how tediously reduction proceeds. One of the reasons for this is the fact that all new nodes are initially tagged with a ✗ even though there is no point ever tagging abstractions, atoms and pairs with anything other than a ✓. The reasons for this is the close correspondence between the reduction rules and the implementation, and the conciseness of the implementation. The implementation of the reduction rules presented in the next chapter is actually shorter that the specification of the reduction rules in this chapter. This is because the implementation uses higher-order functions. To help present each reduction rule separately the use of higher-order functions has been avoided in this chapter. The close correspondence between specification and implementation is maintained as the examples in this chapter are actually generated by the implementation in the next chapter.

# Chapter 5

# Implementation

This chapter presents the implementation of some of the degrees of sharing discussed in the previous chapter. First a conventional abstract syntax tree representation of programs is explained (§5.1). The heap with which programs will exist when executed is introduced (§5.2). A translation by instantiation of the conventional parse tree representation of the program to the unconventional heap with depth tags representation is explained (§5.3). A program to perform conventional lazy evaluation in this unconventional heap is given (§5.4). A novel implementation of full laziness which can be used on a heap both before and during execution is shown (§5.5). The memo-table operations required by the completely lazy and optimal evaluator are explained (§5.6). The implementations of complete-laziness (§5.7) and optimal evaluation (§5.8) are given.

## 5.1 Parser

*Ef*'s syntax is much like Haskell's. The parser returns a parse tree of type `Exp` as shown in Figure 5.1. The `Exp` data-type makes use of the `Atomic` data-type to represent atomic values. The `Atomic` data-type is also used by the data-types used to represent the heap.

## 5.2 Heap

Before the parse tree can be instantiated as a graph, the operation of the heap must be explained.

At times the result of evaluating a node in the graph will be another node. For

```
datatype Atomic =
  AInt of int
| AStr of string
| ANil | ATrue | AFalse

datatype Exp =
  EAtomic of Atomic
| EVar of string
| EPair of (Exp*Exp)
| ELet of ((string*Exp) list*Exp)
| ELambda of (string*Exp)
| EApply of (Exp*Exp)
```

Figure 5.1: Expression type used for parse trees.

```
datatype Val =
  VAtomic of Atomic
| VPair of Addr*Addr
| VVar
| VAp of Addr*Addr
| VLambda of Addr
| VDelta of string*Addr list*(Addr*Addr list->unit)
| VSubst of Addr * (int * Addr * int * MemoTable)                    (comp)
| VSubst of Addr * (int * Addr * int * MemoTable * int list)          (opt)

and Tag =
  T of int                                                        (lazy,full)
| T of int*bool                                                     (comp)
| T of int*int option                                               (opt)
| TReachedBy of Addr list*Tag                                       (full)
| TCopiedTo of Addr*Tag                                          (lazy,full)
```

Figure 5.2: Values and Tags used in the heap.

```
new : Value*Tag -> Addr
newHole : unit -> Addr
deref : Addr -> Value*Tag
update : Addr*(Value*Tag) -> unit
link : Addr*Addr -> unit
addrEq : Addr*Addr -> bool
addrOrd : Addr*Addr -> order
```

Figure 5.3: Heap manipulating functions.

example when reducing a delta node such as `if` or `head`. The result cannot simply be copied as this would lose sharing. One solution is to allow indirection nodes to exist along side other values in the heap. However this complicates any algorithms built on top of the heap as they have to know what to do with an indirection node.

The technique adopted here is to handle indirections at a lower level in a way inspired by the *unifiable references* distributed with the SML/NJ system. Conventional references in SML support the operations create (`ref`), update (`:=`), dereference (`!`) and equality (`=`).

The implementation used here adds the ability to link references and order references. Linking references makes one point to another. Ordering is useful for the memo-tables (see §5.6). Algorithms built on top of this heap abstraction typically need never worry about indirections. When an algorithm would otherwise explicitly place an indirection in the heap, it now just links one node to another. When dereferencing addresses to nodes, the heap abstraction will never return an indirection. Any chains of indirections that may build up, are compressed when they are dereferenced.

The heap is a graph of value-tag pairs, Figure 5.2 shows the definition of `Value` and `Tag`. There are four different versions of *Ef* described in this chapter: lazy, fully lazy, completely lazy and optimal. Unless indicated otherwise the code is common to all implementations. Where a line of code is appended with a comment such (comp,opt), this indicates that that line is only present in some of the implementations.

The functions that are used to perform operations on the heap are shown with their type signatures in Figure 5.3. The function `addrOrd` is used to compare addresses so as to make the implementation of the memo-tables more efficient. The function `newHole` returns an address for a new node before it is known what will be put in the node. This is useful for building cyclic structures. The use of *tags* makes it easy to more uniformly add information to nodes in the heap. Not all node strictly speaking need tags, for example there is little point tagging an atomic node with a depth. Similarly tagging a node to indicate if it is evaluated or blocked on something is only worthwhile for application and primitive nodes (and substitution nodes for optimal evaluation). However tagging all nodes in a consistent way makes the implementation more concise and easier to understand.

## 5.3 Instantiation

The parse tree of an expression is translated into a graphical representation, by instantiating the parse tree in the heap. Instantiation takes a parse tree of type `Exp` and returns the address of type `Addr` of the instantiation. Note how in the instantiated heap the `VVar` constructor takes no arguments; the only way to distinguish variables is by their depth indicated in their tag. It is not necessary to perform a renaming pass over the parse tree as the names of variables are not used after instantiation and the transformation performed on the graphical representation distinguish variables by their depth not their name.

The instantiate function does not create `VDelta` nodes in the heap. The initial environment passed to `inst` is of type `(string*Addr) list` and includes entries for the primitive functions, which already exist in the heap.

```
fun newTag depth = T depth                                    (lazy,full)
fun newTag depth = T (depth,false)                              (comp)
fun newTag depth = T (depth,NONE)                               (opt)

fun inst (ed as (env,depth)) exp =
  case exp of
    EAtomic a => new (VAtomic a,newTag 0)
  | EVar var => lookup var env
  | EPair (e1,e2) => new (VPair (inst ed e1, inst ed e2),newTag depth)
  | ELambda (var,body) =>
     let val lamR = newHole()
         val varR = new (VVar,newTag (depth+1))
         val bodyR = inst ((var,varR)::env,depth+1) body
         val _ = update (lamR,(VLambda bodyR,newTag depth))
     in lamR end
  | EApply (e1,e2) => new (VAp (inst ed e1,inst ed e2),newTag depth)
  | ELet (decls,exp) =>
     let val holesR = map (fn _ =>newHole()) decls
         val (binds,exps) = unzip decls
         val new_env = zip (binds,holesR) @ env
         val addrs = map (inst (new_env,depth)) exps
         val _ = map link (zip (holesR,addrs))
     in inst (new_env,depth) exp end
```

Figure 5.4: Instantiation.

## 5.4 Lazy evaluation

Figure 5.5 shows the implementation of the lazy evaluator, and Figure 5.6 show the implementation of the substitution. The implementation is straightforward. Note how `eval` can assume that reducing the function part of an application node will result in a lambda expression. Since only top-level reductions are performed, for any correct programs this is a safe assumption. The substitution function is fairly straightforward also, again simplified by the fact it is only used for top-level reductions. The `subst` function copies the entire scope of a function. As each node is copied it is tagged with the address of where it has been copied to. When nothing remains to be copied, the graph is *cleaned* of `TCopiedTo` tags, by the function `clean`.

The function `value_reaches` is noteworthy in how it helps abstract the process of substitution over different values. This function will be used again in the implementation of completely lazy and optimal evaluation.

For performing HNF reduction, a slightly modified versions of `eval` and `subst` is used. These versions cope with unreducible nodes, and arbitrary shifts when performing substitutions under lambdas.

```
fun eval addr =
  case deref addr of
    (VDelta (name, args, func),_) => func (addr,args)
  | (VAp (func,arg),_) =>
      let val _ = eval func
          val (VLambda body,_) = deref func
          val copy = subst arg body
          val _ clean body
          val _ = link (addr,copy)
      in eval addr end
  | _ => ()
```

Figure 5.5: `eval` for lazy evaluation.

```
and subst arg orig =
  case deref orig of
    (_,TCopiedTo (to,_))=> to
  | (_,T 0) => orig
  | (VVar,T 1) => arg
  | (value,T depth)=>
      let val (childList,rebuild) = value_reaches value
          val newAddr = newHole ()
          val _ = update (orig,(value,TCopiedTo (newAddr,tag)))
          val newChildList = map (subst' arg) childList
          val _ = update (newAddr,(rebuild newChildList,T (depth-1)))
      in newAddr end

and clean addr =
  case deref addr of
    (_,T _) => ()
  | (value, TCopiedTo (_,tag)) =>
      ( update (addr,(value,tag))
      ; app clean (#1 (value_reaches value)))

and value_reaches (VPair (a1,a2)) = ([a1,a2],fn [a1,a2]=>VPair (a1,a2))
|   value_reaches (VAp (a1,a2)) = ([a1,a2], fn [a1,a2]=>VAp (a1,a2))
|   value_reaches (VLambda a) = ([a], fn [a]=>VLambda a)
|   value_reaches (VDelta (n,args,f)) = (args,fn args=> VDelta (n,args,f))
|   value_reaches value = ([],fn []=>value)
```

Figure 5.6: Substitution for lazy evaluation.

## 5.5 Full laziness

As explained in Chapter 2, full laziness can be achieved using the same reduction mechanism as lazy evaluation. Only a static transformation is required to ensure graph that is syntactically free of a substitution is not duplicated. Previously this transformation has been performed on a parse tree. Figure 5.7 shows an algorithm that achieves the same results on a heap representation.

This scheme works in three stages:

**Stage 1 :** `reached_by` traverses the graph tagging each node with a `TReachedBy` tag containing a list of the nodes that point to it. The function `reached_by` also sets the depth of all nodes to zero, and returns a list associating the addresses of lambdas with the address of the variables they bind. The order of this list is important as the next stage relies upon any nested lambdas occurring later in the list than the lambda whose scope they are nested within.

**Stage 2 :** `sink_bound` repeatedly traverses the graph in reverse from each variable along the `TReachedBy` tags, setting the depth of the nodes met be to one greater than the depth of the variables associated lambda. This upward traversal from each variable does not proceed beyond the lambda that binds the variable. This stage ends once traversals has been performed from every variable.

**Stage 3 :** `clean` cleans the graph of the `TReachedBy` tags.

This algorithm can be generalised to work for just nodes no shallower than a given depth, rather than all reachable nodes. This is useful when the transformation is used dynamically.

Dynamic full laziness has been used to increase the degree of sharing used by head normal form reduction. After performing reductions under a lambda, but before applying that lambda, a dynamic full laziness pass can be performed, ensuring as little graph as possible is copied. Dynamic full laziness would be expensive to apply very often. Applying it intermittently might be a practical compromise. However no full laziness passes (dynamic or otherwise) are required in order to pass the tower of interpreters test, and no worthwhile benefit has been found is using dynamic full laziness.

```
fun floatMFE addrs =
  let val lambdavars = reached_by addrs
  in (app sink_bound lambdavars; app clean addrs) end

and reached_by addrs =
  let val hole = new (VHole,T 0)
     val (lamvars,_) = unzip (map (reached_by_rec hole) addrs)
  in foldr compose id lamvars [] end

and reached_by_rec from addr =
  case deref addr of
    (value,TReachedBy (reflist,tag)) =>
      let val _ = update (addr, (value,TReachedBy (from::reflist,tag)))
      in (id,[]) end
  | (value,T depth) =>
      let val _ = update (addr, (value,TReachedBy ([from], T 0)))
          val (lamvars,vars) =
           unzip (map (reached_by_rec addr) (#1(value_reaches value)))
          val lamvars' = foldr compose id lamvars
          val vars' = mergeListsBy (fn ((a,_),(b,_))=>a>b) vars
      in
        case value of
          VVar => (id,[(depth,addr)])
        | VLambda body =>
            let val (v,vs) = span (fn (d,v)=>d=depth+1) vars'
            in (fn x=>(addr,map #2 v)::lamvars' x,vs) end
        | _ => (lamvars',vars')
      end

and sink_bound (lam,vars) =
  let val (VLambda _,TReachedBy (_,T lambda_depth)) = deref lam
  in app (sink_bound_rec (lam,lambda_depth+1)) vars end

and sink_bound_rec (lam,new_depth) addr =
  let val (value,TReachedBy (reflist, T this_depth)) = deref addr
  in if this_depth < new_depth andalso not (addrEq (addr,lam)) then
       ( update (addr, (value,TReachedBy (reflist, T new_depth)))
       ; app (sink_bound_rec (lam,new_depth)) reflist)
     else ()
  end

and clean addr =
  case deref addr of
    (_,T _) => ()
  | (value, TReachedBy (_,tag)) =>
      ( update (addr,(value,tag))
      ; app clean (#1 (value_reaches value)))
```

Figure 5.7: Full laziness graph transformation.

## 5.6 Memo-tables

The memo-tables are implemented using *red black trees*, as explained by Okasaki [65]. The operations supported on them are shown in Figure 5.8. A node added to a memo-table must not later become an indirection node. Ordering between the addresses is used to speed up memo-table access. The address of the node at the end of any chain of indirection nodes is used to determine this order. If this node itself were to become an indirection, the ordering in the memo-tables would break down. Fortunately, the way in which the completely lazy and optimal evaluators use memo-tables, nodes are always evaluated as far as possible before they are used as an index into a memo-table. Each memo-table has a unique identifier, which can be read using the `memoId` function. This is useful in the implementation of optimal evaluation as this identifier is used to distinguish substitution nodes originating from the different beta-reductions.

```
newMemo : unit -> MemoTable
addMemo : MemoTable -> Addr*(Value*Tag) -> unit
lookupMemo : MemoTable -> Addr -> (Value*Tag) option
memoId : MemoTable -> int
```

Figure 5.8: Memo-table operations.

## 5.7 Complete laziness

For complete laziness the constructor `VSubst` takes five arguments as indicated in Figure 5.2. These arguments correspond to the address of the node to be substituted, the depth of the variable to be bound, the address of the argument the variable is to be bound to, the depth shift to be applied when substituting, and the memo-table to be used.

When performing top-level reductions, no variables will be encountered and reduction can proceed unhindered. When reductions are performed under lambdas this is no longer the case: an attempted reduction may result in an attempt to apply a variable, or add a variable. The path between a node that cannot be further reduced and the variable it is blocked on may be arbitrarily long. To prevent repeated traversing of this path only to discover that the reduction is blocked, the node can be tagged with a boolean indicating whether an attempt has already been

made to reduce it. Thus the constructor `T` contains a boolean as shown in Figure 5.2.

The code for the completely lazy evaluator is shown in Figure 5.9. The code is fairly straightforward. To evaluate a node a check is first made to determine if an attempt has already been made to evaluate the node. If it has then evaluation on this node is complete. Otherwise if the node to be evaluated is an application node the function part is evaluated. If this reduces to a lambda, then a new substitution is formed and evaluated. If the function part does not reduce to a lambda then reduction of the application node is blocked (either because of a type-error, or due to the presence of a variable, the implementation doesn't distinguish which).

The code to perform substitution is shown in Figure 5.10. The evaluation of a substitution proceeds firstly by evaluating the node that is to be substituted. Substitution then proceeds as follows. First the depth of the node to be substituted is compared to the depth of the variable to be bound. If the node is at a shallower depth, then the substitution node has reached the end of its scope and *shorts* itself out, i.e. the substitution node is replaced with a link to the node that was to be substituted. Next a check is made to see if the node to be substituted has been substituted by a related substitution before. (Two substitutions are related if they originated from the same $\beta$-reduction). If it has then the substitution node is replaced by a link to the result of the previous substitution. If the node to be substituted is the variable the substitution wishes to bind, as determined by its depth, then the substitution is replaced by a link to the argument. All other nodes are copied with their depth incremented by `shift`, if the node to be copied points to other nodes (such as a application, abstraction or pair node), then the copied node points to these same nodes via new substitution nodes.

No attempt to substitute a substitution node will ever occur. The reduction of the body of the substitution ensures that any substitution is *evaluated away* before any attempt to substitute the substitution occurs. It is this evaluation step which has the specializing effect. By replacing this specializing evaluation step with a call to a function which only *evaluates away* substitution nodes and doesn't reduce application or delta nodes, the completely lazy implementation can be turned into an (inefficient) implementation of a lazy evaluation.

After the substitution is finished, (back in Figure 5.9), the result is evaluated. Delta nodes contain the function used to perform the desired primitive operation.

```
fun eval addr =
 let val (value,T(depth,blocked)) = deref addr in
   if blocked then () else
     case value of
       VAp (func,arg) =>
        (case (eval func; deref func) of
         (VLambda body,T (lam_depth,_)) =>
          let val bind = lam_depth+1
              val shift = depth-bind
              val memo = newMemo ()
              val _ = update (addr,(VSubst (body,(bind,arg,shift,memo)),
                                    T(depth,false)))
          in eval addr end
        | _ => set_blocked addr)
     | VSubst (addr',common as (depth,arg,shift,memo)) =>
        (eval addr'; link (addr,subst (addr',common)); eval addr)
     | VDelta (name,args,func) => func (addr,args)
     | _ => set_blocked addr
 end

and set_blocked addr =
  let val (value,T(depth,_)) = deref addr
  in update (addr,(value,T(depth,true))) end
```

Figure 5.9: `eval` for Complete laziness.

```
and subst (orig,common as (bind,arg,shift,memo)) =
 let val (value,T(depth,_)) = deref orig
     val new_depth = depth+shift
 in if depth < bind then orig else
     case lookupMemo memo orig of
       SOME copy => copy
     | NONE =>
        let val copy =
         case value of
           VVar=>
            if depth=bind then arg
            else new (VVar,T(new_depth,true))
         | VLambda body =>
            let val subst_body = new (VSubst (body,common),
                                      T(new_depth+1,false))
            in new (VLambda subst_body,T(new_depth,true)) end
         | other=>
            let val (reaches,rebuild) = value_reaches other
                fun mkSubst addr = new (VSubst (addr,common),
                                        T(new_depth,false))
                val new_addrs = map mkSubst reaches
            in new (rebuild new_addrs,T(new_depth,false)) end
        in (addMemo memo (orig,copy); copy) end
 end
```

Figure 5.10: Delayed substitution.

These primitive functions make calls back to `eval` as necessary. The implementation of these primitive functions is not shown here. All other nodes (i.e. atomic, variable and pairs nodes) are already fully reduced and simply need to be tagged as such.

## 5.8 Optimal evaluation

To implement optimal evaluation, each node needs to be tagged to indicate which variable it is blocked on. Rather than just tagging nodes with a `bool` to indicate if an attempt has been made to reduce the node, nodes are tagged with an `int option`. A node is tagged with:

`NONE`, if no attempt has been made to evaluate it,

`SOME 0`, if the node has been evaluated and further reduction is not blocked by the presence of any variables, and

`SOME` $n$, if the node has been evaluated and further reduction is blocked by the need for a substitution binding a variable at depth $n$.

Nodes are still tagged with their depth, just as before, so the constructor `T` is now of type `int*int option -> Tag`, as defined in Figure 5.2.

The substitution nodes used for optimal evaluation contain one more fields compared to the substitution nodes used in completely lazy evaluation. This extra field, typically denoted `sb` (standing for *subst-by)* in the code, indicates which substitutions have substituted the substitution.

The key difference between the completely lazy and optimal implementations of *Ef*, is the handling of substitution nodes. The completely lazy evaluator only ever tags a substitution node as unevaluated (✗) as a substitution node is always reducible. In contrast the optimal *Ef* only permits a substitution node to be reduced in two situations:

1. when the graph that the substitution node will substitute requires that substitution in order to enable further reduction, and

2. when the graph to be substituted is already fully reduced, such as a pair or lambda node.

```
fun eval addr =
 let val (value,T(depth,blocked)) = deref addr
 in
   if blocked <> NONE then () else
     case value of
      VAp (func,arg) =>
       (case (eval func; deref func) of
        (VLambda body,T (lambda_depth,_))=>
         let
          val memo = newMemo ()
          val bind = lambda_depth + 1
          val shift = depth - bind
          val _ = update (addr,(VSubst (body,(bind,arg,shift,memo,[]))
                                ,T(depth,NONE)))
          val _ = eval addr
         in () end
        |(_,T(_,SOME lambda_blocked)) => set_blocked (addr,lambda_blocked)
       )
      |VDelta (name,args,func) => func (addr,args)
      |VVar => set_blocked (addr,depth)
      |VSubst (addr',common as (bind,arg,shift,memo,sb)) =>
       (case (eval addr'; deref addr') of
        (_,T(_,SOME subst_blocked)) =>
         if subst_blocked=0 orelse subst_blocked=bind then
          let  val new_addr = (collectSubsts) ([common],addr')
               val _ = link (addr,new_addr)
          in eval addr end
         else
          if subst_blocked < bind
          then set_blocked (addr, subst_blocked)
          else set_blocked (addr, subst_blocked+shift)
       )
      |other=> set_blocked (addr,0)
 end

and set_blocked (addr,blocked_on) =
  let val (value,T(depth,_)) = deref addr
  in update (addr,(value,T(depth,SOME blocked_on))) end
```

Figure 5.11: `eval` for optimal evaluation.

```
and subst (s as (addr,common as (bind,arg,shift,memo,sb))) =
 let val (value',T(depth,_)) = deref addr
     val new_depth = depth+shift
 in
  if depth < bind then addr else
   case lookupMemo memo addr of
     SOME to' => to'
   | NONE =>
     let
      val substed_addr =
       case value' of
         VSubst (s' as (addr',common as (bind',arg',shift',memo',sb'))) =>
           if bind < bind' then substSubst (depth,s,s')
                           else unsubstSubst (depth,s,s')
       | VVar =>
         if depth=bind then arg
         else new (VVar,T(depth+shift,NONE))
       | VLambda body =>
         let val new_subst = new (VSubst (body,common),T(new_depth+1,NONE))
         in new (VLambda new_subst,T(new_depth,NONE)) end
       | other =>
         let
          fun mkSubst addr = new (VSubst (addr,common),T(new_depth,NONE))
          val (reaches,rebuild) = subst_reaches other
          val new_substs = map mkSubst reaches
          val substed_addr = new (rebuild new_substs,T(new_depth,NONE))
         in substed_addr end
      val _ = addMemo memo (addr,substed_addr)
     in substed_addr end
 end
```

Figure 5.12: Substitution for optimal evaluation.

```
and collectSubsts (substs,addr) =
 let val (value,_) = deref addr
 in
   case value of
     VSubst (subst as (addr',common as (bind,arg,shift,memo,sb))) =>
       if far_enough (substs,(bind,shift,memo,sb)) then rebuildSubsts (substs,addr)
       else collectSubsts ((common::substs),addr')
   | other =>
       rebuildSubsts (substs,addr)
 end

and rebuildSubsts (substs,addr) =
 foldl (fn (common,addr) => subst (addr,common)) addr substs

and far_enough ([],subst) = true
|   far_enough ((bind,_,shift,memo,sb)::substs, subst' as (bind',shift',memo',sb')) =
 if bind < bind'  then far_enough (substs,(bind'+shift,shift',memo',memoId memo::sb'))
 else if (memoId memo') elem sb  then far_enough (substs,(bind',shift',memo',sb'))
 else false
```

Figure 5.13: Handling sequences of substitutions.

```
and substSubst
 (depth,
  (addr, (bind ,arg ,shift ,memo ,sb)),
  (addr',(bind',arg',shift',memo',sb'))
 ) =
 let
  val lower_subst =
    new (VSubst (addr', (bind, arg, shift, memo, sb)),
         T(depth+shift-shift',NONE))
  val upper_subst =
    new (VSubst (lower_subst,
           (bind'+shift,arg',shift',memo',memoId memo::sb')),
         T(depth+shift,NONE))
 in upper_subst end

and unsubstSubst
 (depth,
  (addr, (bind ,arg ,shift ,memo ,sb)),
  (addr',(bind',arg',shift',memo',sb'))
 ) =
 let
  val lower_subst =
    new (VSubst (addr',
           (bind-shift',arg, shift, memo,remove1 (memoId memo') sb)),
         T(depth+shift-shift',NONE))
  val upper_subst =
    new (VSubst (lower_subst, (bind',arg',shift',memo',sb')),
         T(depth+shift,NONE))
 in upper_subst end

and remove1 a (l::ls) = if a=l then ls else l::remove1 a ls
```

Figure 5.14: Substitution swapping.

If neither of these conditions are met, the substitution node is blocked and is tagged to indicate which variable a substitution must bind in order to enable further reduction.

Figure 5.13 shows how the build up of unswappable substitutions between a required substitution node and the node to be substituted are handled. `eval` starts the process by calling `collectSubsts` with the required substitution. If the node the first substitution points to is not a substitution node, then `collectSubsts` calls `rebuildSubsts` which will actually perform the substitution. Otherwise `collectSubsts` will call `far_enough`, which tests to see if an entire sequence of substitutions can substitute/unsubstitute their way through a substitution. If all the substitutions within the sequence can then `rebuildSubsts` is called and performs substitutions for each of the substitutions in the sequence. Otherwise the substitution is added to the existing sequence.

Figure 5.12 shows how substitutions are handled. The main difference from the completely lazy version is that this version is able swap substitutions. The function `subst` is only ever called when the substitution (or unsubstitution) is possible, it will never encounter two substitutions which cannot be swapped.

There are a number of improvements which could be made to the implementation of optimal evaluation. Such as using a tree instead of a list to remember which substitutions are allowed to swap. Also there is room for improvement in the way sequences of substitutions are handled. However, although it has been interesting to see how complete laziness can be generalised to optimal evaluation, it achieves a greater degree of sharing than is needed to achieve the specializing effect of partial evaluation. Furthermore as will be seen in the next chapter, optimal evaluation is not suited to removing layers of interpretation, and it is difficult to see how optimal evaluation could be used to generate specialized code dynamically.

## 5.9 Summary

This chapter presented implementations of different degrees of sharing, all the way from lazy to optimal. A full laziness transformation was presented which can be used on graphs in the heap at run-time.

# Chapter 6

# Results

This chapter presents the results of experiments conducted, using the lazy, fully lazy, completely lazy and optimal versions of *Ef*, and also Asperti et al's BOHM [8] optimal evaluator.

Section 6.1 describes the experiments conducted to demonstrate that the completely lazy *Ef* does indeed pass the tower of interpreters test. *Ef* must be able to pass the this test in order for *Ef* to be able to specialize programs which are evaluated indirectly through one or more interpreters. Section 6.2 discusses some of the limitations of *Ef*. Sections 6.3 and 6.4 generalise the tower of interpreters results to heterogeneous towers of interpreters. Section 6.5 describes how well the optimal *Ef* and BOHM perform on the tower of interpreters test. Section 6.6 demonstrates *Ef* specializing away an imperative interpreter and further specializing the program evaluated by the imperative interpreter. Finally Section 6.7 compares the lazy, fully lazy, completely lazy and optimal versions of *Ef* with each other and BOHM on the examples presented in the BOHM literature.

Each experiment conducted was run as a separate process to prevent the contents of the heap in one experiment influencing future experiments. This also makes it possible to compare experiments in an implementation-language independent manner.

For each experiment the CPU time is reported. This time is the sum of the user time and the system time reported by the operating system after the process has finished. The real time is also measured so that the proportion of time spent by the CPU on each experiment can be calculated. This proportion was over 90% for all long running experiments (experiments running longer than 10 seconds). This indicates the experiments were not significantly affected by other processes running

on the machine, or by virtual memory paging delays.

For each experiment, the operating system was instructed to terminate the experiment's process if the process required more than 3600 seconds of CPU time, or more than 480 MBytes of virtual address space.

The maximum size of each experiment's address space is also measured. This is done by intercepting the operating system calls made by a process. After each system call is complete but before returning control to the process being measured, the current size of the processes address space is compared with the running maximum. This technique makes it possible to compare the memory requirements of programs implemented in different languages.

All experiments were conducted on a Sun with a 270MHz UltraSPARC IIi processor and 512MB memory. This computer was running SunOS version 5.6. Version 110.0.3 of SML/NJ was used. Version 4.08.1 of GHC was used. The default compiler settings were used.

## 6.1 Towers of interpreters

To demonstrate that the completely lazy version of *Ef* is able to specialize away interpretive overhead, a tower of interpreters is constructed. The same interpreter is used at each stage in the tower. The reason for conducting experiments with many identical interpretive layers present is to provide convincing evidence that the execution speed of the program at the top of the tower is the same regardless of the number of layers of interpretation. If the languages at each layer in the tower were different then it would not be possible to have the same program at the top of the tower in all the experiments.

The interpreter used in the tower is shown in Figure 6.1. The interpreter evaluates parse trees. An example of such a parse tree is shown in Figure 6.4. The result of evaluating the parse tree for a function is that function, rather than a representation of the function. If the parse tree being evaluated is that of the interpreter, then the resulting function can be used in just the same way as the original interpreter. The function `primitive` takes a string and returns the corresponding primitive function.

Figure 6.3 shows constructors that can be used to build up suitable parse trees for the interpreter to process. To indicate to the reader the structure of the parse tree

```
eval where

map f x@(x1:xs) = if x==[] then [] else f x1:map f xs
lookup a l@((key,value):ls) = if a==key then value else lookup a ls
a@(a1:as) ++ b = if a==[] then b else a1:(as++b)

eval = evale []
evale env = ee where
  ee exp@(tag,a@(a1,a2)) =
    if tag=="EVar"    then lookup a env                   else
    if tag=="EApply"  then (ee a1) (ee a2)                else
    if tag=="EPrim"   then primitive a                    else
    if tag=="EPair"   then (ee a1,ee a2)                  else
    if tag=="ELit"    then a                              else
    if tag=="ELambda" then (\arg->eval ((a1,arg):env) a2) else
    if tag=="ELet"    then eval_decls env a1 a2           else
    "undefined"

eval_decls env decls = evalenv where
   env' = map decl_to_env decls
   evalenv = evale (env'++env)
   decl_to_env (id,exp) = (id,evalenv exp)
```

Figure 6.1: An interpreter for *Ef* parse trees written in *Ef*.

```
addup where
 addup n = if n==0 then 0 else n+addup(n-1)
```

Figure 6.2: A simple program at the top of the tower of interpreters.

```
                                          data Exp =
EVar var = ("EVar", var)                    EVar String
ELit lit = ("ELit", lit)                   |ELit Value
EPrim prim = ("EPrim", prim)               |EPrim String
EApply func arg = ("EApply", (func,arg))   |EApply Exp Exp
EPair hd tl = ("EPair", (hd,tl))           |EPair Exp Exp
ELambda var body = ("ELambda", (var,body)) |ELambda String Exp
ELet decls body = ("ELet", (decls,body))   |ELet [(String,Exp)] Exp
```

Figure 6.3: Parse tree constructors.

```
ELet                                                          let
 [("addup",                                                   addup=
   ELambda "n" (EApply (EApply (EAppply (EPrim "if")          \n->if
    (EApply (EApply (EPrim "==") (EVar "n")) (ELit 0))        n==0
    (ELit 0)                                                  then 0
    (EApply (EApply (EPrim "+") (EVar "n"))                   else n+
     (EApply (EVar "addup")                                   addup
      (EApply (EApply (EPrim "-") (EVar "n")) (ELit 1)))))))  (n-1)
 )]                                                           in
(EVar "addup")                                                addup
```

Figure 6.4: Parse tree for `addup`.

| | addup | | | | |
|---|---|---|---|---|---|
| layers | 1 | 10 | 100 | 1000 | 10000 |
| 0 | 0.02s | 0.02s | 0.01s | 0.02s | 0.07s |
| 1 | 0.01s | 0.03s | 0.01s | 0.05s | 0.61s |
| 2 | 0.17s | 0.20s | 0.48s | 3.10s | 41.68s |
| 3 | 0.71s | 2.98s | 25.75s | 258.88s | 3238.73s |
| 4 | 39.10s | 228.17s | 2185.00s | 22564.91s | |
| 5 | 3182.41s | | | | |

Table 6.1: GHC evaluating a tower of interpreters (time).

| | addup | | | | |
|---|---|---|---|---|---|
| layers | 1 | 10 | 100 | 1000 | 10000 |
| 0 | 2.4MB | 2.4MB | 2.4MB | 2.4MB | 2.4MB |
| 1 | 2.9MB | 2.9MB | 2.9MB | 2.9MB | 3.9MB |
| 2 | 2.9MB | 2.9MB | 2.9MB | 2.9MB | 5.9MB |
| 3 | 2.9MB | 2.9MB | 3.9MB | 5.9MB | 35.6MB |
| 4 | 3.9MB | 3.9MB | 3.9MB | 14.1MB | |
| 5 | 3.9MB | | | | |

Table 6.2: GHC evaluating a tower of interpreters (space).

| | addup | | | | |
|---|---|---|---|---|---|
| layers | 1 | 10 | 100 | 1000 | 10000 |
| 0 | 0.03s | 0.03s | 0.02s | 0.02s | 0.05s |
| 1 | 0.07s | 0.07s | 0.06s | 0.16s | 1.63s |
| 2 | 0.64s | 0.79s | 1.79s | 12.11s | 116.20s |
| 3 | 2.50s | 12.05s | 105.96s | 1056.30s | |
| 4 | 166.80s | 1019.95s | 9700.78s | | |
| 5 | 15045.69s | | | | |

Table 6.3: SML/NJ evaluating a tower of interpreters (time).

| | addup | | | | |
|---|---|---|---|---|---|
| layers | 1 | 10 | 100 | 1000 | 10000 |
| 0 | 6.1MB | 6.1MB | 6.1MB | 6.1MB | 6.1MB |
| 1 | 6.7MB | 6.7MB | 6.7MB | 9.4MB | 17.0MB |
| 2 | 10.2MB | 10.2MB | 10.2MB | 13.6MB | 30.9MB |
| 3 | 13.7MB | 15.4MB | 21.8MB | 40.3MB | |
| 4 | 24.2MB | 285.7MB | 328.6MB | | |
| 5 | 34.0MB | | | | |

Table 6.4: SML/NJ evaluating a tower of interpreters (space).

built by these constructors, a corresponding Haskell definition for the constructors is also shown. *Ef* doesn't support user defined types and doesn't distinguish identifiers by the capitalization of the first letter, so the constructors in Figure 6.1 are just functions. Although these constructors could be used to construct suitable parse trees, in practice the parse trees are created by an external process that reads in *Ef* source code and outputs a concrete representation of the *Ef* value representing the parse tree. This value is then re-parsed by the implementation of *Ef* being used.

The strings in the parse tree and interpreter such as `"EVar"` are used to tag data with types in much the same way constructors are used in Haskell or SML. In all the example *Ef* programs shown in this chapter no operations to construct or take apart strings are performed. The only strings in existence during execution are those present before execution began.

To demonstrate the advantage of the completely lazy *Ef* evaluator over more conventional language implementations, the tower of interpreters experiment is also conducted with GHC and SML/NJ: simple implementations of *Ef* written in Haskell and SML are used. In each experiment, the program at the top of the tower is the `addup` function shown in Figure 6.2. For a more complex program at the top of the tower, the last interpreter can be considered to be at the top. Zero interpretive layers corresponds to the addup function being written directly in Haskell and SML. One interpretive layer corresponds to the simple implementation of *Ef* written in Haskell or SML directly evaluating the `addup` function. More layers correspond to the simple implementation of *Ef* interpreting one or more layers of the interpreter in Figure 6.1 interpreting the parse tree of the `addup` function.

The results in Tables 6.1 and 6.3 indicate that each interpreter in the tower contributes approximately an 80-fold increase in evaluation time. The important point here is that the interpretive overhead is multiplicative, not additive.

From Table 6.5 it can be seen that the interpretive overhead when using the completely lazy *Ef* is additive and not multiplicative. Each additional interpreter in the tower contributes approximately 20 seconds to the evaluation time.

The number of additions performed by the `addup` function at the top of the tower is varied in order to be able to measure the speed of execution once any additive overhead due to the number of layers of interpretation has been accounted for. It can be seen from Table 6.5 that the speed at which the completely lazy *Ef* adds numbers using the `addup` function is the same regardless of the number of layers

of interpretation used. There is an apparent inconsistency in the speed at which `addup` operates. The time difference between evaluating `addup 10000` and `addup 1000` is more than the ten times the time difference between evaluating `addup 1000` and `addup 100`. However this unexpected slowdown in adding up numbers can also be seen to be independent of the number of layers of interpretation being used. An explanation of this slowdown is given in §6.2.

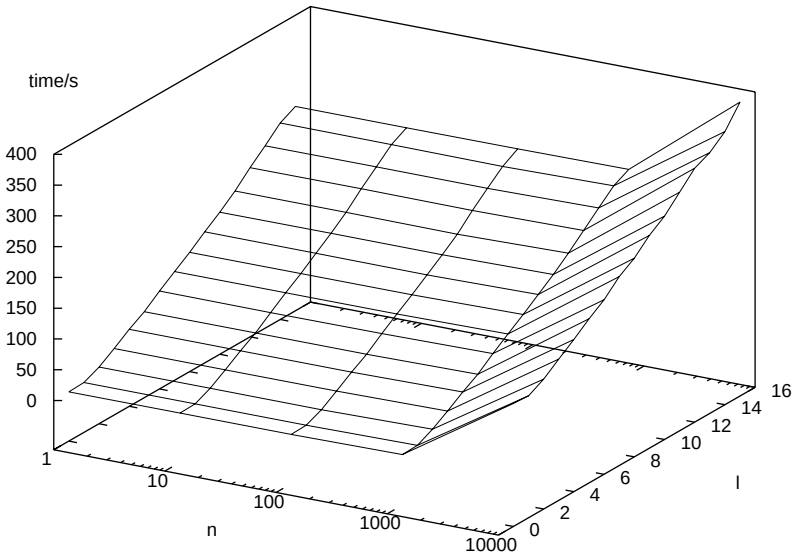
Any small non-linearity in the results in Table 6.5 can be explained by the additional time taken during garbage collection. It can be seen in Table 6.6 that the space usage grows linearly with the number of layers of interpretation. Ideally the space usage would be independent of the number of layers of interpretation. As each layer is eliminated it no longer has an impact on the speed of evaluation and yet appears to retain space in the heap. The reason for this is that the specialization of one interpreter is not finished before the specialization of the next starts. The specialization of the interpreters takes place concurrently in a lazy fashion. An explanation of why the space usage is so high is given in §6.2.

The apparent variability in the additional space required by additional interpretive layers is due to the nature of the underlying garbage collector. *Ef* is implemented in SML/NJ which uses a generational copying garbage collector. The additional space requested by SML/NJ from the operating system is influenced by the number of cells in the heap which remain after a garbage collection; this in turn is influenced by the precise point in time at which a garbage collection is triggered. So the space used by a program is only an approximate indication of the space required by the program.

## 6.2 Infinite unfolding

Using memo-tables to maintain sharing causes *Ef* to have limitations. As these memo-tables grow larger, the time taken to update and inspect them grows longer. Delayed substitutions always evaluate nodes as far as they can before substituting into them. This evaluation can result in the scope of a substitution becoming infinite. This occurs when delayed substitution is performed on typical recursive functions which cannot be reduced to normal form. A simple example is the `addup` function from Figure 6.5. Before a substitution node substitutes into the recursive function call, it reduces the recursive function call. This reduction results in a cyclic substitution which has an unbounded scope to substitute through. The body of

| | | $n$ | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 10 | 100 | 1000 | 10000 |
| | 0 | 0.04s | 0.05s | 0.13s | 2.00s | 119.71s |
| | 1 | 0.91s | 0.91s | 0.97s | 2.89s | 118.06s |
| | 2 | 11.99s | 12.18s | 12.07s | 15.01s | 131.20s |
| | 3 | 28.01s | 28.06s | 28.24s | 30.33s | 149.49s |
| | 4 | 45.02s | 44.85s | 44.91s | 47.22s | 167.11s |
| | 5 | 59.55s | 59.78s | 59.77s | 61.92s | 186.62s |
| | 6 | 79.07s | 78.71s | 79.28s | 81.37s | 203.45s |
| | 7 | 96.63s | 96.67s | 96.87s | 99.17s | 226.94s |
| $l$ | 8 | 113.68s | 113.89s | 113.98s | 116.64s | 242.77s |
| | 9 | 132.88s | 132.73s | 132.13s | 135.05s | 265.73s |
| | 10 | 150.76s | 151.14s | 150.90s | 153.45s | 282.92s |
| | 11 | 170.88s | 170.91s | 170.06s | 173.97s | 307.68s |
| | 12 | 194.74s | 195.30s | 194.52s | 196.70s | 332.39s |
| | 13 | 216.16s | 216.61s | 216.31s | 219.24s | 343.75s |
| | 14 | 239.21s | 238.57s | 237.74s | 241.00s | 363.90s |
| | 15 | 251.89s | 251.78s | 252.21s | 253.62s | 397.53s |

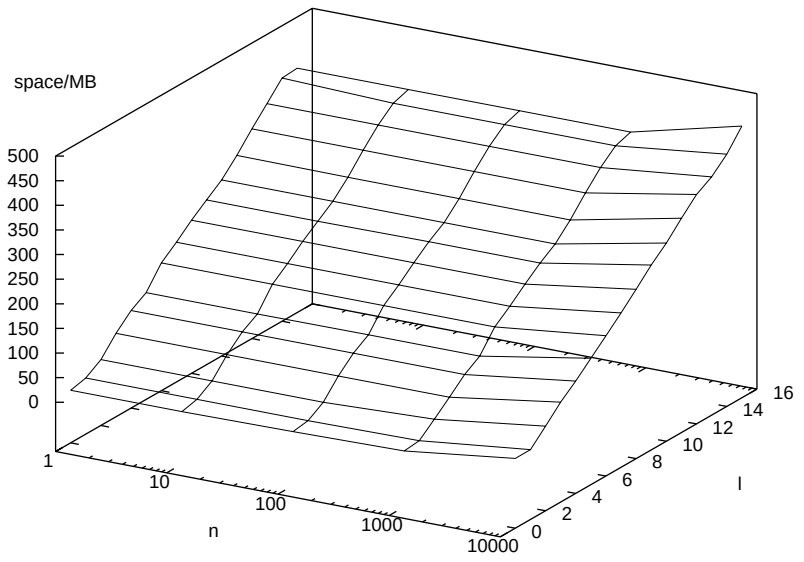



```
il l = if l==1 then eval else il (l-1) eval_prs
tower l n = if l==0 then addup n else il l addup_prs n
```

*Time[* `tower l n` *]*

Table 6.5: *Ef* completely lazy evaluation of a tower of interpreters (time).

| | | $n$ | | | | |
|---|---|---|---|---|---|---|
| | | `1` | `10` | `100` | `1000` | `10000` |
| | 0 | 6.6MB | 6.6MB | 9.4MB | 13.3MB | 41.6MB |
| | 1 | 13.9MB | 13.9MB | 13.9MB | 16.6MB | 41.8MB |
| | 2 | 33.7MB | 33.7MB | 33.7MB | 42.7MB | 71.9MB |
| | 3 | 69.5MB | 69.5MB | 69.5MB | 69.5MB | 102.9MB |
| | 4 | 98.1MB | 98.1MB | 98.1MB | 98.1MB | 128.0MB |
| | 5 | 116.8MB | 116.8MB | 117.8MB | 117.8MB | 156.7MB |
| | 6 | 159.5MB | 159.5MB | 159.5MB | 159.5MB | 184.7MB |
| | 7 | 183.8MB | 183.8MB | 183.8MB | 183.8MB | 213.6MB |
| $l$ | 8 | 211.0MB | 211.0MB | 211.0MB | 211.0MB | 243.9MB |
| | 9 | 234.5MB | 234.5MB | 236.1MB | 236.1MB | 275.3MB |
| | 10 | 257.4MB | 257.4MB | 257.4MB | 257.4MB | 302.5MB |
| | 11 | 289.5MB | 288.5MB | 288.5MB | 288.5MB | 335.1MB |
| | 12 | 325.7MB | 325.7MB | 325.7MB | 325.7MB | 365.9MB |
| | 13 | 359.4MB | 359.4MB | 359.4MB | 359.4MB | 384.0MB |
| | 14 | 393.8MB | 385.9MB | 385.9MB | 385.9MB | 412.2MB |
| | 15 | 396.2MB | 396.2MB | 396.2MB | 396.2MB | 451.7MB |




*Space[* `tower l n` *]*

Table 6.6: *Ef* completely lazy evaluation of a tower of interpreters (space).

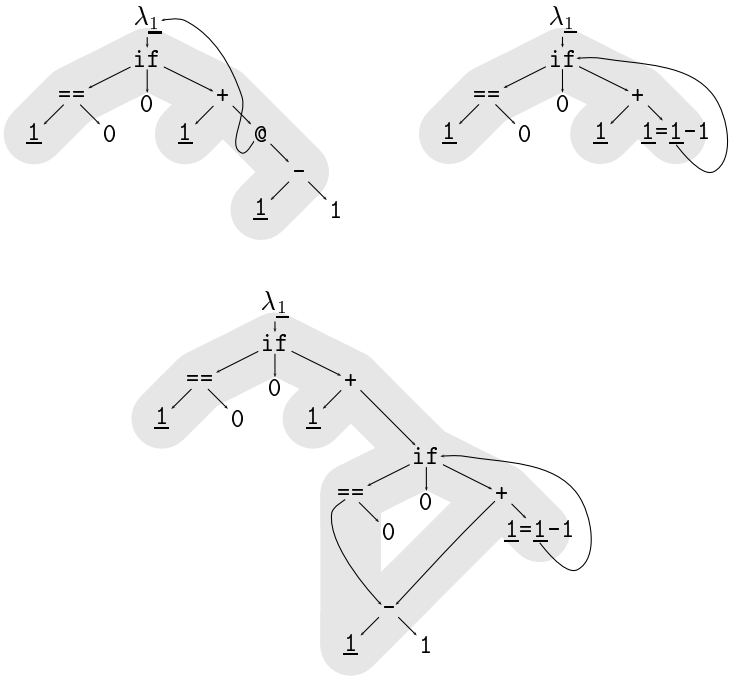



Figure 6.5: Infinite unfolding.

the addup function grows to the size of the largest number it is applied to. In this case the $\beta$-reductions are shared across multiple applications, and in principle share work. However, from a practical point of view, this kind of specializing is worse than useless: it causes the memo-tables to grow very large and the use of the function to slow down.

There are tricks that could be used to prevent cyclic substitutions forming. It would be possible to have every substitution node know which lambda *owned* the graph though which the substitution node is substituting. Then when performing reductions ahead of substitution nodes it would be possible to prevent any applications that would create an infinite scope. This would ensure that all substitutions had a finite amount of graph through which to substitute.

However this is not a natural solution. It is not clear that such applications should always be prevented, and preventing such applications does not prevent the symptoms from occurring. Preventing cyclic substitutions does not stop function bodies becoming arbitrarily large. Consider a function (such as `addup`) made non-cyclic by using a non-cyclic Y combinator. While no single substitution will have an infinite scope, the `addup` function's body will still grow arbitrarily large. Any substitution resulting from this very large function being applied will grow similarly and all the same infinite unfolding slowdown symptoms will occur.

## 6.3 Heterogeneous tower — *Ef* / $Ef_{case}$

*Ef* is a relatively simple language and a user could quickly tire of its lack of features. Rather than program directly in *Ef*, a user can write in a more sophisticated language interpreted by *Ef*.

In order to demonstrate that the experiments in §6.1 were not merely an experimental artifact relying on the interpreters at each layer in the tower being the same, a heterogeneous tower of interpreters has been built. The interpreter in Figure 6.6 is an interpreter for $Ef_{case}$ written in *Ef*. The interpreter in Figure 6.7 is an interpreter for *Ef* written in $Ef_{case}$.

The parse trees evaluated by the $Ef_{case}$ interpreter are much like those evaluated by the *Ef* interpreter. The parse trees could be constructed with the same constructor functions shown in Figure 6.3, with the addition of a constructor for case expression. This constructor takes a list of patterns and a list of expressions. The patterns could be made using the constructors `EVar`, `ELit`, and `EPair`. As before, the parse trees are actually created by an external process.

The interpreter in Figure 6.6 is shown complete with all utility functions. It is the parse tree for the entire program which is used in the tower, including the utility functions.

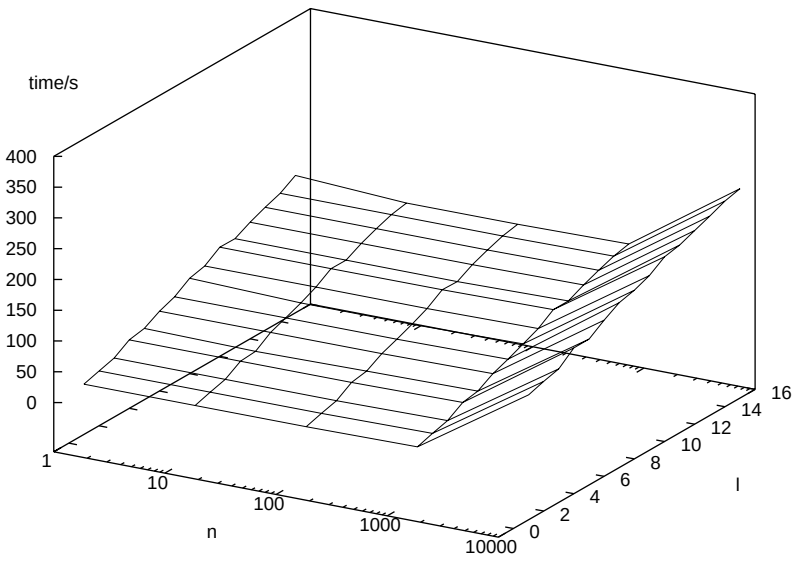
```
eval_efcase where

map f x@(x1:xs) = if x==[] then [] else f x1:map f xs
lookup a l@((key,value):ls) = if a==key then value else lookup a ls
a@(a1:as) ++ b = if a==[] then b else a1:(as++b)
unzip l = (map head l, map tail l)
filter f x@(x1:xs) = if x==[] then [] else
  if f x1 then x1:filter f xs else filter f xs
zip x@(x1:xs) y@(y1:ys) = if x==[] || y==[] then [] else (x1,y1):zip xs ys

eval_efcase = evale []
evale env = ee where
  lookup_env a = lookup a env
  ee exp@(tag,a@(a1,a2)) =
    if tag=="EVar"    then lookup_env a                    else
    if tag=="EApply"  then (ee a1) (ee a2)                 else
    if tag=="EPrim"   then primitive a                     else
    if tag=="EPair"   then (ee a1,ee a2)                   else
    if tag=="ELit"    then a                               else
    if tag=="ELambda" then (\arg->evale ((a1,arg):env) a2) else
    if tag=="ELet"    then eval_decls env a1 a2            else
    if tag=="ECase"   then eval_case env (ee a1) a2        else
    "undefined"

eval_decls env decls = evalenv where
   env' = map decl_to_env decls
   evalenv = evale (env'++env)
   decl_to_env (id,exp) = (id,evalenv exp)

eval_case env value cases =
  let
    (pats,exps) = unzip cases
    matches = zip (map (match value) pats) exps
    ((_,env'),exp'):_ = filter (\((match,env),exp)->match) matches
    result = evale (env'++env) exp'
  in result

match v@(v1,v2) (tag,p@(p1,p2)) =
  if tag=="ELit" then (v==p,[]) else
  if tag=="EVar" then (True,[(p,v)]) else
  if tag=="EPair" then
    let (match1,env1) = match v1 p1
        (match2,env2) = match v2 p2
    in (match1 && match2,env1++env2)
  else "undefined"
```

Figure 6.6: An interpreter for $Ef_{case}$ parse trees written in *Ef*.

```
eval_ef where

map f x = case x of [] -> []
                    x1:xs -> f x1 : map f xs
a ++ b = case a of [] -> b
                   a1:as -> a1:(as++b)
lookup a l@((key,value):ls) = if a==key then value else lookup a ls

eval_ef = evale []
evale env = ee where
 ee exp =
    case exp of
      ("EApply",func,arg)  -> (ee func) (ee arg)
      ("ELit",lit)         -> lit
      ("ELet",decls,body)  -> eval_decls env decls body
      ("ELambda",var,body) -> \arg->evale ((var,arg):env) body
      ("EPair",h,t)        -> (ee h,ee t)
      ("EVar",var)         -> lookup var env
      ("EPrim",prim)       -> primitive prim

eval_decls env decls = evalenv where
   env' = map decl_to_env decls
   evalenv = evale (env'++env)
   decl_to_env (id,exp) = (id,evalenv exp)
```

Figure 6.7: An interpreter for *Ef* parse trees written in $Ef_{case}$.

| | | $n$ | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 10 | 100 | 1000 | 10000 |
| | 1 | 0.65s | 0.68s | 0.69s | 1.73s | 61.84s |
| | 2 | 4.42s | 4.48s | 4.47s | 5.75s | 65.65s |
| | 3 | 8.22s | 8.28s | 8.27s | 9.26s | 69.09s |
| | 4 | 15.14s | 15.99s | 16.05s | 16.91s | 81.22s |
| | 5 | 18.91s | 18.47s | 18.90s | 19.99s | 80.98s |
| | 6 | 25.87s | 26.23s | 26.29s | 27.13s | 85.09s |
| | 7 | 30.39s | 29.09s | 29.28s | 30.13s | 95.11s |
| $l$ | 8 | 35.36s | 36.32s | 37.05s | 37.57s | 97.33s |
| | 9 | 40.37s | 40.15s | 40.17s | 42.81s | 101.86s |
| | 10 | 47.26s | 47.81s | 47.63s | 49.01s | 106.60s |
| | 11 | 51.87s | 50.81s | 51.27s | 53.16s | 110.66s |
| | 12 | 57.98s | 58.23s | 58.24s | 59.34s | 119.84s |
| | 13 | 60.89s | 60.84s | 61.25s | 62.34s | 126.55s |
| | 14 | 69.42s | 69.16s | 70.18s | 71.03s | 132.45s |
| | 15 | 74.62s | 74.37s | 73.88s | 74.75s | 134.17s |

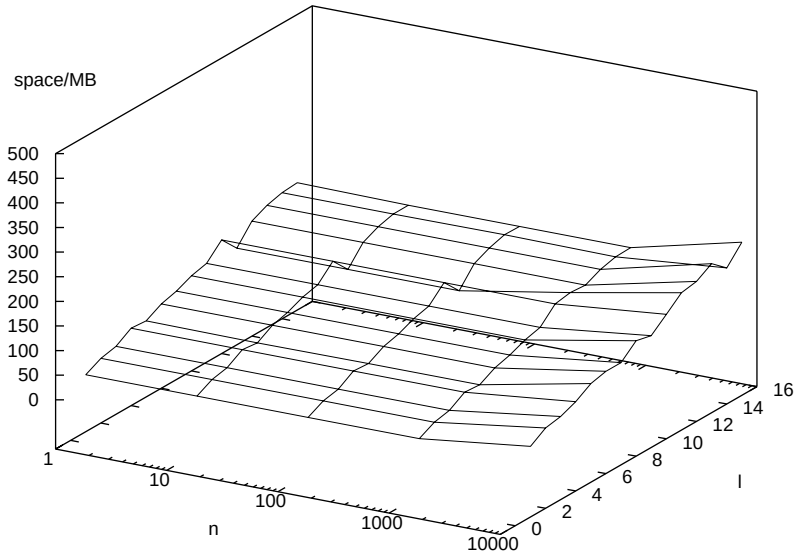



*Time[* `htower` *l n ]*

```
hil n =
  if n==1 then eval_efcase else
  if even n then hil (n-1) eval_ef_prs
            else hil (n-1) eval_efcase_prs

even n = n `mod` 2 == 0

htower l n = hil l addup_prs n
```

Table 6.7: *Ef* completely lazy evaluation of a hetergeneous tower of interpreters (time).

| | | $n$ | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 10 | 100 | 1000 | 10000 |
| | 1 | 11.6MB | 11.8MB | 11.8MB | 15.2MB | 37.0MB |
| | 2 | 23.4MB | 24.0MB | 24.0MB | 29.5MB | 58.3MB |
| | 3 | 34.4MB | 35.2MB | 35.0MB | 35.0MB | 59.0MB |
| | 4 | 43.4MB | 49.2MB | 48.7MB | 48.7MB | 82.0MB |
| | 5 | 58.5MB | 60.0MB | 72.5MB | 72.5MB | 89.7MB |
| | 6 | 70.8MB | 81.0MB | 70.6MB | 79.0MB | 106.7MB |
| | 7 | 85.2MB | 73.2MB | 84.8MB | 76.8MB | 134.9MB |
| $l$ | 8 | 79.6MB | 98.5MB | 98.5MB | 98.5MB | 142.1MB |
| | 9 | 94.5MB | 109.7MB | 94.4MB | 126.4MB | 147.2MB |
| | 10 | 126.8MB | 126.8MB | 126.9MB | 126.8MB | 156.8MB |
| | 11 | 143.8MB | 129.2MB | 129.2MB | 143.6MB | 160.5MB |
| | 12 | 118.1MB | 128.0MB | 128.0MB | 128.0MB | 191.6MB |
| | 13 | 127.1MB | 128.3MB | 128.3MB | 128.3MB | 194.2MB |
| | 14 | 150.7MB | 150.7MB | 150.7MB | 150.7MB | 192.4MB |
| | 15 | 173.4MB | 173.2MB | 173.2MB | 173.2MB | 209.5MB |




*Space[* `htower` *l n ]*

Table 6.8: *Ef* completely lazy evaluation of a hetergeneous tower of interpreters (space).

## 6.4 Heterogeneous tower — *Ef* / LispKit Lisp

It is instructive to explore which features of *Ef* are required in order to use the techniques presented in this thesis to completely eliminate a layer of interpretation. Building towers of interpreters provides very strong experimental evidence that a layer of interpretation has been eliminated.

Two features of *Ef* which appear essential are:

- non-strict semantics,
- typelessness.

Non-strict semantics are required to be able to use the knot-tying technique which lazy-specialization relies upon.

Typelessness is required to prevent the double-encoding issues which occur when specializing typed languages. Although techniques to mitigate the overhead incurred when self-applying a partial evaluator for a typed language have been developed [56], the fact still remains that it is representations of values which are manipulated, and repeated encoding still incurs an overhead.

LispKit Lisp [33] serves as an example of a language which is non-strict and typeless and yet differs from *Ef* in some respects:

- LispKit Lisp supports run-time type inspection (RTTI) operations such as `ispair` and `isfunc`,
- LispKit Lisp primitives are non-curried, whereas *Ef*'s are curried.

In order to implement an interpreter for a language such as LispKit Lisp, which contain primitives such as `ispair` and `isfunc` two approaches could be taken:

1. Extend the implementation of *Ef* to contain the required primitives,
2. Tag all the values operated on by the LispKit Lisp interpreter so that additional information is carried around.

Taking the first approach would be simpler, but taking the second approach is more instructive.

The values operated on by the LispKit Lisp interpreter can be constructed using the constructor functions shown in Figure 6.8. If the LispKit Lisp interpreter

```
VAtomic a = ("VAtomic", a)
VPair h t = ("VPair", (h,t))
VFunc f = ("VFunc", f)
```

Figure 6.8: Constructor functions for LispKit Lisp values.

```
eval_efrtti where

map f x@(x1:xs) = if x==[] then [] else f x1:map f xs
lookup a l@((key,value):ls) = if a==key then value else lookup a ls
a@(a1:as) ++ b = if a==[] then b else a1:(as++b)

eval_efrtti = evale []

evale env = ee where
  ee exp@(tag,a@(a1,a2)) =
    if tag=="EVar"    then lookup a env else
    if tag=="EApply"  then apply (ee a1) (ee a2) else
    if tag=="EPrim"   then lookup a prims else
    if tag=="EPair"   then ("VPair", (ee a1,ee a2)) else
    if tag=="ELit"    then ("VAtomic", a) else
    if tag=="ELambda" then ("VFunc", (\arg->evale ((a1,arg):env) a2)) else
    if tag=="ELet"    then eval_decls env a1 a2 else
    "undefined"

eval_decls env decls = evalenv where
   env' = map decl_to_env decls
   evalenv = evale (env'++env)
   decl_to_env (id,exp) = (id,evalenv exp)

apply ("VFunc", func) arg = func arg

liftop f = ("VFunc", \("VAtomic",a)->
           ("VFunc", \("VAtomic",b)->
             ("VAtomic",f a b)))

prims = [
  ("+",    liftop (+)),  ("-",    liftop (-)), ("*",    liftop (*)),
  ("||",   liftop (||)), ("&&",   liftop (&&)),
  ("if",   ("VFunc", \("VAtomic",a)->
           ("VFunc",\b->
           ("VFunc",\c->
             if a then b else c)))),
  ("==",   liftop (==)), ("/=",   liftop (/=)),
  (":",    ("VFunc",\a->
           ("VFunc",\b->
             ("VPair", (a,b))))),
  ("head", ("VFunc",\("VPair",(a,b))->a)),
  ("tail", ("VFunc",\("VPair",(a,b))->b)),
  ("ispair", ("VFunc", \(tag,data)->("VAtomic",tag=="VPair"))),
  ("isfunc", ("VFunc", \(tag,data)->("VAtomic",tag=="VFunc")))
 ]
```

Figure 6.9: An interpreter for $Ef_{RTTI}$ written in *Ef*.

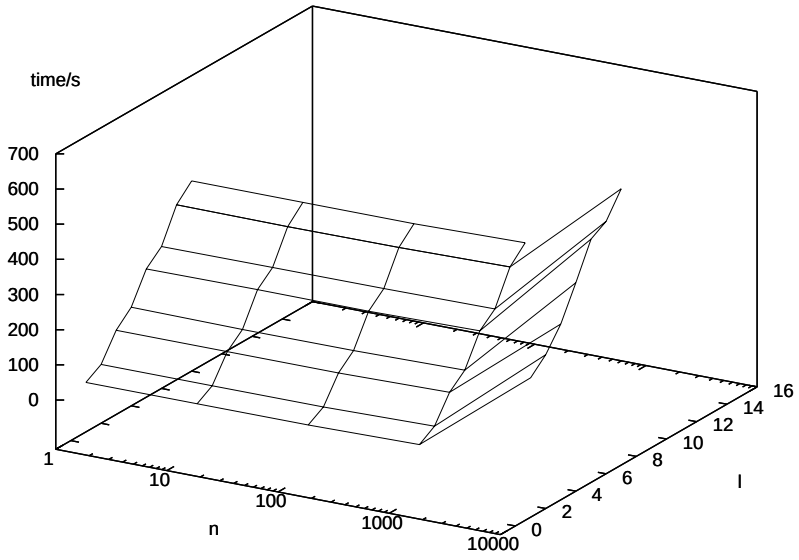
```
eval_lk where

zip x@(x1:xs) y@(y1:ys) = if x==[] then [] else (x1,y1):zip xs ys
lookup x ((n,v):nvs) = if x==n then v else lookup x nvs
foldl f z l@(l1:ls) = if l==[] then z else foldl f (f z l1) ls
map f x@(x1:xs) = if x==[] then [] else f x1:map f xs
x@(x1:xs) ++ y = if x==[] then y else x1:xs++y
length = foldl (\x _->x+1) 0

eval_lk = eval[]
evale env sexp@(h:t) =
 if ispair sexp then
  if h == "quote" then quote (head t)
  else if h == "lambda" then
    let formalArgs = head t
        body = head (tail t)
    in if length formalArgs == 1 then
       (\actualArg -> evale ((head formalArgs, actualArg):env) body) else
       (\actualArgs -> evale (zip formalArgs actualArgs++env) body)
  else if h == "let" then
    let body = head t
        decls = tail t
        add_decl env (name,exp) = (name,evale env exp):env
        env'  = foldl add_decl env decls
    in evale env' body
  else if h=="letrec" then
    let body = head t
        decls = tail t
        add_decl env (name,exp) = (name,evale env exp)
        env' = (map (add_decl env') decls) ++ env
    in evale env' body
  else
    if length t == 1 then
    (evale env h) (evale env (head t)) else
    (evale env h) (map (evale env) t)
 else lookup sexp env

prim_lk = [
  ("add", uncurry 2 (+)),  ("sub", uncurry 2 (-)),
  ("mul", uncurry 2 (*)),  ("eq",  uncurry 2 (==)),
  ("head", head),          ("tail", tail),
  ("cons", uncurry 2 (:)), ("ispair", ispair),
  ("if",  uncurry 3 (\a b c->if a then b else c))
 ]

uncurry n f args@(arg1:argss) =
  if n==1 then f arg1
  else uncurry (n-1) (f arg1) argss

quote a@(h,t) =
  if ispair a then (quote h,quote t) else
  if a=="NIL" then []
  else if a=="t" then True
  else if a=="f" then False
  else a
```

Figure 6.10: An interpreter for LispKit Lisp written in $Ef_{RTTI}$.

```
(letrec eval_ef
  (eval_ef . (lambda (exp) (eval (quote NIL) exp)))
  (eval . (lambda (env exp)
    (letrec
        (if (eq tag (quote ELit)) a
        (if (eq tag (quote EVar)) (lookup a env)
        (if (eq tag (quote EApply)) ((eval env a1) (eval env a2))
        (if (eq tag (quote EPrim)) (lookup a prims)
        (if (eq tag (quote EPair)) (cons (eval env a1) (eval env a2))
        (if (eq tag (quote ELambda))
          (lambda (arg) (eval (cons (cons a1 arg) env) a2))
        (if (eq tag (quote ELet))
          (letrec (eval env' a2)
            (env' . (append
               (map (lambda (name_exp)
                 (cons (head name_exp)
                       (eval env' (tail name_exp)))) a1)
               env
            )))
        (quote undefined_tag))))))))
      (tag . (head exp))
      (a   . (tail exp))
      (a1  . (head a))
      (a2  . (tail a))
    )))
  (lookup . (lambda (x nvs)
    (let (if (eq x name) value (lookup x nvss))
      (nv    . (head nvs))
      (name  . (head nv))
      (value . (tail nv))
      (nvss  . (tail nvs))
    )
  ))
  (prims .
    (cons (cons (quote +) (curry2 add))
    (cons (cons (quote -) (curry2 sub))
    (cons (cons (quote *) (curry2 mul))
    (cons (cons (quote if) (curry3 if))
    (cons (cons (quote ==) (curry2 eq))
    (cons (cons (quote :) (curry2 cons))
    (cons (cons (quote head) head)
    (cons (cons (quote tail) tail)
    (cons (cons (quote ispair) ispair)
    (quote NIL))))))))))

  (curry2 . (lambda (f) (lambda (a) (lambda (b) (f a b)))))
  (curry3 . (lambda (f) (lambda (a) (lambda (b) (lambda (c) (f a b c))))))
  (map lambda (f x)
    (if (eq x (quote NIL)) (quote NIL) (cons (f (head x)) (map f (tail x)))))
  (foldr . (lambda (f z l) (if (eq l (quote NIL)) z (f (head l) (foldr f z (tail l))))))
  (append . (lambda (x y) (foldr cons y x)))
)
```

Figure 6.11: An interpreter for $Ef_{RTTI}$ written in LispKit Lisp.

| | | $n$ | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 10 | 100 | 1000 | 10000 |
| | 1 | 0.71s | 0.68s | 0.86s | 5.88s | 256.48s |
| | 2 | 27.01s | 26.97s | 27.23s | 33.58s | 297.23s |
| | 3 | 99.61s | 99.37s | 99.52s | 104.88s | 359.93s |
| $l$ | 4 | 140.43s | 137.73s | 138.36s | 143.40s | 452.79s |
| | 5 | 224.48s | 226.39s | 225.70s | 230.50s | 551.81s |
| | 6 | 263.34s | 264.72s | 265.39s | 268.05s | 577.56s |
| | 7 | 357.46s | 356.09s | 357.55s | 362.69s | 645.64s |
| | 8 | 400.59s | 400.23s | 400.10s | 406.68s | |

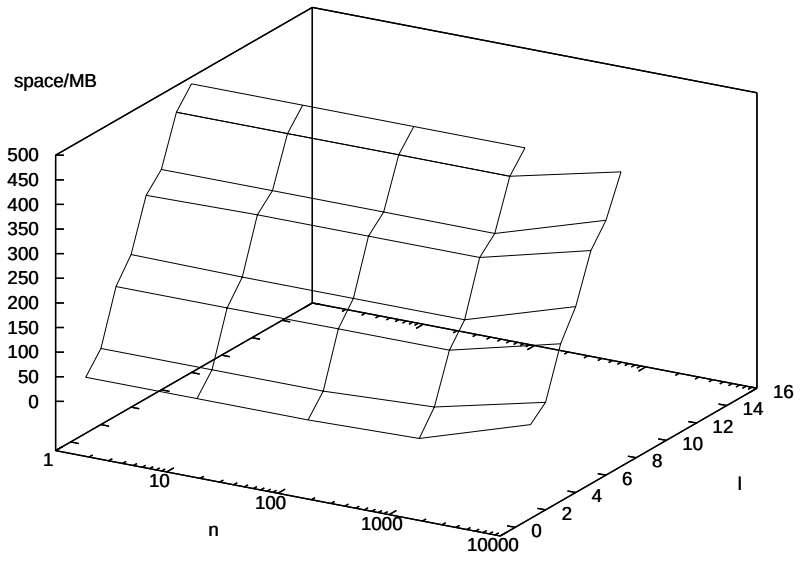



*Time[* `lktower` *l n ]*

```
even x = x `mod` 2 == 0

lk_il l =
  if l==2 then eval_lift eval_lk else
  if even l then lk_il (l-1) `apply` eval_lk_prs_lift
            else lk_il (l-1) `apply` eval_ef_prs_lift

lktower l n =
  if l==1 then eval_prs_lift addup_prs `apply` ("VAtomic",n) else
  if even l then lk_il l `apply` addup_lk_prs_lift `apply` ("VAtomic",n)
            else lk_il l `apply` addup_prs_lift `apply` ("VAtomic",n)
```

Table 6.9: Completely lazy evaluation of a hetergeneous *Ef* / LispKit Lisp tower of interpreters (time).

| | | $n$ | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 10 | 100 | 1000 | 10000 |
| | 1 | 13.6MB | 13.6MB | 13.6MB | 19.3MB | 90.7MB |
| | 2 | 54.6MB | 54.6MB | 54.6MB | 65.5MB | 118.5MB |
| | 3 | 162.7MB | 162.7MB | 163.4MB | 163.4MB | 219.6MB |
| $l$ | 4 | 210.0MB | 207.4MB | 207.4MB | 207.4MB | 277.5MB |
| | 5 | 312.5MB | 316.3MB | 316.2MB | 316.3MB | 373.7MB |
| | 6 | 347.4MB | 347.6MB | 347.6MB | 347.4MB | 417.5MB |
| | 7 | 446.1MB | 446.0MB | 446.1MB | 446.0MB | 498.1MB |
| | 8 | 486.0MB | 486.0MB | 486.0MB | 485.9MB | |




*Space[* `lktower` *l n ]*

Table 6.10: Completely lazy evaluation of a hetergeneous *Ef* / LispKit Lisp tower of interpreters (space).

encoded values using these constructor functions directly, then building a tower including such interpreters would result in multiple encodings. Instead the approach taken is to handle the encoding of values once at the bottom of the tower and not in every occurrence of a LispKit Lisp interpreter in the tower.

To achieve this one-off handling of value encoding, an enhanced version of *Ef*, $Ef_{RTTI}$, is implemented in *Ef* first. It is within this enhanced version that the interpreter for LispKit Lisp is written.

Figures 6.9, 6.10 and 6.11 show the implementations of $Ef_{RTTI}$ in *Ef*, LispKit Lisp in $Ef_{RTTI}$, and $Ef_{RTTI}$ in LispKit Lisp.

Tables 6.9 and 6.10 show the time and space requirements for an interpretive tower consisting of one `eval_efrtti`, and a tower of alternating `eval_lk` and `eval_ef`.

The results indicate, as before, that the interpretive overhead is additive and not multiplicative. There are some aspects of implementation of LispKit Lisp worthy of mention. The definition of `zip` is unusual in that it only tests to see if its first argument is `[]`, not both as is more usual. The function `zip` is being used inside multi-argument function definitions to zip together variables and their values. If `zip` tests its second argument this introduces an overhead that is incurred every time the function that `zip` is helping to build is called.

It can be seen that the curried primitives in *Ef* are implemented using the non-curried primitives in LispKit Lisp and vice-versa. The experiments demonstrate that this implementation of primitives has no adverse effect on execution speed, the interpretive overhead is still additive not multiplicative.

It is necessary to handle the implementation of 1-ary functions in LispKit Lisp slightly differently from the implementation of n-ary functions. Functions of arity two or greater receive their arguments in a list. If this was the case also for 1-ary functions, then 1-ary functions would always expect their argument as a singleton list. This would apply also to functions written in *Ef* and interpreted by the *Ef* interpreter written in LispKit Lisp. If a tower of such interpreters were built, the placing of arguments in singleton lists would compound, resulting in a multiplicative interpretive overhead.

| | addup | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| layers | 1 | 3 | 10 | 30 | 100 | 300 | 1000 | 3000 | 10000 |
| 0 | 0.16s | 0.16s | 0.16s | 0.23s | 0.51s | 1.54s | 7.51s | 38.61s | 349.91s |
| 1 | 1.32s | 1.31s | 1.34s | 1.53s | 2.22s | 4.97s | 15.86s | 68.23s | 426.67s |
| 2 | 43.16s | 43.18s | 43.28s | 43.83s | 45.82s | 54.89s | 85.40s | 201.61s | — |
| 3 | 204.18s | 204.88s | 205.08s | 207.31s | 229.06s | 279.29s | — | — | — |
| 4 | — | — | — | — | — | — | — | — | — |

Table 6.11: *Ef* optimal evaluation of a tower of interpreters (time).

| | addup | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| layers | 1 | 3 | 10 | 30 | 100 | 300 | 1000 | 3000 | 10000 |
| 0 | 8.8MB | 8.8MB | 8.8MB | 11.8MB | 13.1MB | 14.0MB | 19.7MB | 44.6MB | 109.6MB |
| 1 | 15.5MB | 15.5MB | 15.5MB | 13.6MB | 16.5MB | 22.5MB | 34.1MB | 83.6MB | 215.6MB |
| 2 | 97.5MB | 97.5MB | 97.5MB | 97.5MB | 97.5MB | 113.7MB | 146.1MB | 242.1MB | — |
| 3 | 318.2MB | 318.2MB | 318.2MB | 318.2MB | 350.6MB | 409.6MB | — | — | — |
| 4 | — | — | — | — | — | — | — | — | — |

Table 6.12: *Ef* optimal evaluation of a tower of interpreters (space).

| | addup | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| layers | 1 | 3 | 10 | 30 | 100 | 300 | 1000 | 3000 | 10000 |
| 0 | 0.04s | 0.03s | 0.05s | 0.04s | 0.05s | 0.05s | 0.05s | 0.08s | 0.25s |
| 1 | 0.06s | 0.10s | 0.13s | 0.27s | 0.73s | 2.02s | 7.05s | | |
| 2 | 11.98s | 16.60s | 33.27s | — | — | — | — | — | — |
| 3 | — | — | — | — | — | — | — | — | — |

Table 6.13: BOHM optimal evaluation of a tower of interpreters (time).

| | addup | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| layers | 1 | 3 | 10 | 30 | 100 | 300 | 1000 | 3000 | 10000 |
| 0 | 2.1MB | 2.1MB | 2.1MB | 2.1MB | 2.1MB | 2.1MB | 2.2MB | 2.4MB | 2.9MB |
| 1 | 2.6MB | 2.8MB | 3.5MB | 5.4MB | 12.2MB | 31.5MB | 99.0MB | | |
| 2 | 169.2MB | 216.8MB | 383.4MB | — | — | — | — | — | — |
| 3 | — | — | — | — | — | — | — | — | — |

Table 6.14: BOHM optimal evaluation of a tower of interpreters (space).

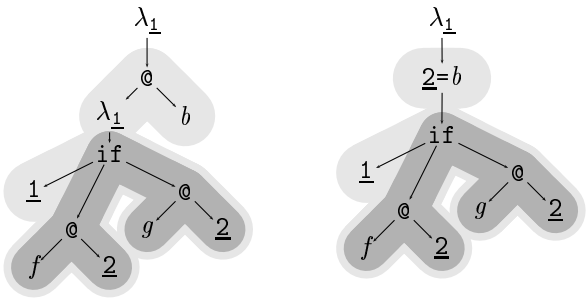



Figure 6.12: A *stuck* substitution.

## 6.5 Optimal evaluation of towers of interpreters

It would be easy to assume that an optimal evaluator would pass the tower of interpreters test. However neither the optimal *Ef* nor BOHM do.

Compared with the completely lazy *Ef*, both the optimal *Ef* and BOHM perform badly. The optimal *Ef* appears to be having some specializing effect although each additional layer slows down execution by a factor of two to three. BOHM appears not to have any specializing effect at all.

In the case of the optimal *Ef* this is due to memo-tables being shared to a greater degree than they are with the completely lazy *Ef*. A simple example demonstrating the negative effect of this greater degree of sharing can be seen in Figure 6.12.

Using a completely lazy reduction order, the substitution $\underline{2} = \texttt{b}$ will be pushed through the `if` before the substitution binding a value for $\underline{1}$ reaches the $\underline{1}$ in the conditional part of the `if`. Using *Ef*'s optimal reduction order the $\underline{2} = \texttt{b}$ substitution does not proceed until a substitution binding $\underline{1}$ has substituted through both the $\underline{2} = \texttt{b}$, and the `if` making it possible for the `if` to be reduced. Reduction of either `f` $\underline{2}$ or `g` $\underline{2}$ will then proceed. When further reduction becomes blocked on $\underline{2}$, the $\underline{2} = \texttt{b}$ binding substitutes through the nodes between itself and the blocked node. Some of these nodes result from the substitution binding $\underline{1}$. It is the substitution of these new nodes that causes the memo-table belonging to $\underline{2} = \texttt{b}$ to grow very large. The problem is that the function in Figure 6.12 may be applied to any number of values. As each of these substitute through $\underline{2} = \texttt{b}$, new nodes beneath $\underline{2} = \texttt{b}$ are created. The memo-table belonging to $\underline{2} = \texttt{b}$ grows larger every time the function is applied, resulting in progressively slower execution.

```
EConst a = ["EConst",a]
EVar a = ["EVar",a]
EOp op arg1 arg2 = ["EOp",op,arg1,arg2]

SAssign l r = ["SAssign",l,r]
SIf c t e = ["SIf",c,t,e]
SGoto l = ["SGoto",l]
```

Figure 6.13: Constructor functions for embedding flowchart parse trees within *Ef*.

```
power_prs =
  [("main",
    [SAssign "result" (EConst 1)
    ,SGoto "loop"
    ]
   )
  ,("loop",
    [SIf (EOp (==) (EVar "n") (EConst 0))
      []
      [SAssign "result" (EOp (*) (EVar "result") (EVar "x"))
      ,SAssign "n" (EOp (-) (EVar "n") (EConst 1))
      ,SGoto "loop"
      ]
    ]
   )
  ]
```

Figure 6.14: The power program written in a flowchart language.

```
evalProg names prog init_values = jump "main" init_values where

  jump label = lookup label prog'
  prog' = map (\(label,stmts)->(label,evalStmt stmts)) prog
  deref n1 = deref' names where
    deref' (n:ns) (v:vs) = if n1==n then v else deref' ns vs
  assign n1 v1 = assign' names where
    assign' (n:ns) (v:vs) = if n1==n then (v1:vs) else v:assign' ns vs

  evalStmt stmts@([tag,a,b,c]:rest) values =
    if stmts==[] then values else
    if tag=="SAssign" then
      let value = evalExp b values
          values' = assign a value values
      in evalStmt rest values' else
    if tag=="SGoto" then jump a values else
    if tag=="SIf" then if evalExp a values
                      then evalStmt b values
                      else evalStmt c values
    else "undefined"

  evalExp [tag,a,b,c] values =
    if tag == "EConst" then a else
    if tag == "EVar" then deref a values else
    if tag == "EOp" then a (evalExp b values) (evalExp c values)
    else "undefined"
```

Figure 6.15: A flowchart interpreter written in *Ef*.

```
ones = 1:ones
take n (l:ls) = if n==0 then [] else l:take (n-1) ls
map f = mapf where mapf x@(x1:xs) = if x==[] then [] else f x1:mapf xs
power1 n x = head (evalProg ["result","n","x"] power_prs [0,n,x])
power2 n x = head (power2' [1,n,x]) where
  power2' [result,n,x] = if n==0
                         then [result,n,x] else
                         power2' [x*result,n-1,x]
```

Figure 6.16: Power functions with and without layers of interpretation.

## 6.6 Specializing a flowchart interpreter

To demonstrate the completely lazy specialization of a strict interpreter, a flowchart interpreter has been written. Unlike the programs written for $Ef_{case}$ and LispKit Lisp, there is no concrete syntax for this language. Instead parse trees for the interpreter are constructed using the constructor functions in Figure 6.13. An example program written for the flowchart interpreter is shown in Figure 6.14.

Figure 6.15 shows the interpreter. The function `evalProg` is called with the list of names of variables used in a flowchart program, the parse tree of the flowchart program, and a list of the initial values of the variables in the flowchart program. Splitting the environment into separate lists of names and values is a standard trick used to make programs more amenable to specialization [45]. The interpreter uses the knot-tying technique to bound the number of calls to `evalStmt`. This ensures that specialization dependent on the parse tree is finite. It is also necessary to ensure that `names` is substituted *before the knot is tied*. This ensures that specialization dependent on `names` is finite.

It can be verified that the interpretive layer has been eliminated in three ways:

1. Look in the heap: If the power function resulting from specialization is partially applied to any number greater than 1, then the heap representation of the new power function contains no reference to any environment variables or parse tree constructs.

2. Modify the primitives in the *Ef* evaluator, in particular the equality test on strings: After specialization no further string comparisons are performed. This was the most useful technique when writing the flowchart interpreter. The implications of binding `names` after `prog` in the definition of the function `evalProg` (rather than before as shown) were not immediately appreciated. Having the string comparison test print out successful comparisons made it

| | | $m$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 5 | 10 | 20 | 50 | 100 | 200 | 500 | 1000 |
| | 1 | 0.55s | 0.52s | 0.53s | 0.54s | 0.56s | 0.65s | 0.85s | 1.37s | 3.55s | 8.75s |
| | 2 | 0.56s | 0.56s | 0.55s | 0.56s | 0.56s | 0.66s | 0.89s | 1.37s | 3.66s | 9.37s |
| | 5 | 0.54s | 0.51s | 0.52s | 0.53s | 0.61s | 0.70s | 0.90s | 1.54s | 4.02s | 10.20s |
| | 10 | 0.53s | 0.58s | 0.56s | 0.59s | 0.61s | 0.81s | 0.99s | 1.76s | 4.19s | 10.34s |
| $n$ | 20 | 0.58s | 0.55s | 0.58s | 0.61s | 0.64s | 0.87s | 1.30s | 2.26s | 6.08s | 14.45s |
| | 50 | 0.65s | 0.68s | 0.70s | 0.78s | 0.86s | 1.34s | 2.23s | 4.05s | 11.05s | 24.88s |
| | 100 | 0.84s | 0.91s | 0.88s | 1.04s | 1.32s | 2.22s | 3.85s | 7.41s | 20.14s | 44.70s |
| | 200 | 1.29s | 1.32s | 1.46s | 1.74s | 2.24s | 4.22s | 7.72s | 15.40s | 40.32s | 88.19s |
| | 500 | 3.13s | 3.33s | 3.82s | 4.71s | 6.69s | 12.78s | 23.75s | 45.97s | 124.05s | 265.64s |
| | 1000 | 7.71s | 8.06s | 9.41s | 11.87s | 16.90s | 32.87s | 62.65s | 124.66s | 337.08s | — |

Table 6.15: *Time[* `take` $m$ `(map (power1` $n$`) ones)`*]*

| | | $m$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 5 | 10 | 20 | 50 | 100 | 200 | 500 | 1000 |
| | 1 | 0.39s | 0.39s | 0.38s | 0.40s | 0.39s | 0.48s | 0.71s | 1.19s | 3.49s | 8.57s |
| | 2 | 0.37s | 0.36s | 0.37s | 0.33s | 0.40s | 0.50s | 0.67s | 1.26s | 3.55s | 9.04s |
| | 5 | 0.33s | 0.37s | 0.34s | 0.42s | 0.41s | 0.57s | 0.77s | 1.29s | 3.95s | 9.89s |
| | 10 | 0.37s | 0.39s | 0.37s | 0.36s | 0.42s | 0.56s | 0.86s | 1.55s | 4.63s | 11.31s |
| $n$ | 20 | 0.39s | 0.36s | 0.38s | 0.47s | 0.48s | 0.65s | 1.15s | 2.12s | 6.03s | 14.49s |
| | 50 | 0.44s | 0.43s | 0.45s | 0.55s | 0.62s | 1.09s | 1.92s | 3.72s | 10.44s | 24.68s |
| | 100 | 0.52s | 0.51s | 0.59s | 0.75s | 0.95s | 1.85s | 3.55s | 7.36s | 19.44s | 43.12s |
| | 200 | 0.77s | 0.81s | 0.91s | 1.23s | 1.81s | 3.78s | 7.58s | 14.98s | 41.10s | 89.94s |
| | 500 | 1.69s | 1.84s | 2.33s | 3.28s | 5.00s | 10.45s | 20.34s | 41.55s | 107.80s | 263.08s |
| | 1000 | 4.16s | 4.65s | 6.27s | 8.71s | 13.93s | 30.54s | 60.62s | 125.76s | 341.28s | — |

Table 6.16: *Time[* `take` $m$ `(map (power2` $n$`) ones)`*]*

quite clear that the parse tree had been specialized away, but the environment had not.

3. Measure the reduction times: Tables 6.15 and 6.16 shows the reduction times using the functions in Figure 6.16.

The reduction times shown in Tables 6.15 and 6.16 demonstrate that the combined overheads of specializing the flowchart interpreter to the syntax tree of the power program, and specializing the resulting power function to the desired power, are very small. Small as they may be, the overheads are not constant. The difference between Tables 6.15 and 6.16 is shown in Table 6.17. The most striking characteristic of this difference is the apparent volatility in the difference between the execution times, for longer execution runs. This difference ranges approximately between 16s and -4s. This volatility is not due to random fluctuations in the timing of experiments. Variations between repeated runs of the same experiment are typically around 1%. The volatility is due largely to the unpredictable nature of garbage collections. In some cases this effect results in executions of the power

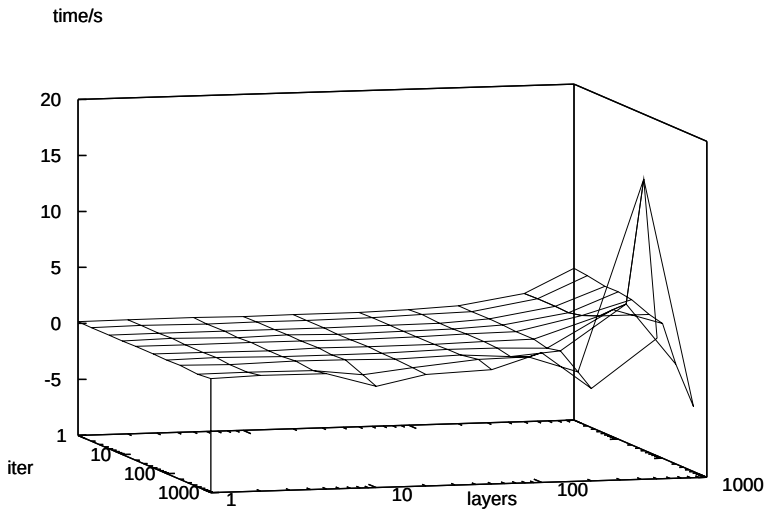



Table 6.17: *Time[* `take` $m$ `(map (power1` $n$`) ones)`*]* - *Time[* `take` $m$ `(map (power2` $n$`) ones)`*]*

function implemented in the interpreted flowchart language running faster than the power function directly implemented in *Ef*.

A second trend is that the difference in time taken to specialize the power function to a given value varies with the value to which the power function is specialized. The directly implemented power function can be specialized faster. This is due to infinite unfolding. Some of the substitution nodes used in specializing the flowchart interpreter get caught up in the infinite unfolding in the resulting power function. These substitutions arise from applications such as `evalStmt a values` being reduced before a value for `values` is substituted.

A similar effect can be seen in the towers of interpreters experiments. The first interpreter in a tower of interpreters is specialized much more quickly than the remaining interpreters. However the effect is not cumulative.

When a solution to the infinite-unfolding problem is found, these slightly odd artifacts will no longer appear.

It would be possible to write an interpreter in the flowchart language. However since the knot-tying trick is not possible in programs written in the flowchart language, the amount of specializing required would not be bounded by the size of the parse trees being interpreted.

## 6.7 BOHM examples

There is very little literature on the kinds of programs and styles of programming that make an optimal evaluator useful. The approach to optimal evaluation presented in this thesis prompts the observation that optimal evaluation is useful when a multi-argument function is applied to a combination of arguments and the benefits of specializing the function with respect to its first argument and to its second argument are wanted simultaneously. This occurs with expressions such as `map (\x-> map (\y-> f (x,y)) [0..]) [0..]`. Although in this example the values to which `y` is bound are independent of `x`, in general these values could be dependent on `x`.

The only other existing optimal evaluator for the $\lambda$-calculus, is the Bolgna Optimal Higher-order Machine (BOHM) [10, 11], discussed further in §2.11. The literature on BOHM contains the only descriptions of programs executed with an optimal evaluator. Many of the programs used to test BOHM are based on Church numerals. Of the four other based programs presented in the BOHM literature, (Prime, Transclos, Mergesort and Tartaglia), three are tamed by full laziness (Prime, Transclos and Mergesort) and three are tamed by complete laziness (Prime, Mergesort and Tartaglia). These programs are presented in §6.7.1 to §6.7.4, essentially copied, but converted to *Ef*'s syntax. The reduction times, space requirements and number of $\beta$-reductions are presented for several reduction schemes with different degrees of sharing.

The differences in the numbers of $\beta$-reductions performed by the optimal *Ef* and BOHM are partly due to the way BOHM handles top-level expressions, and partly due to the way *Ef* handles primitive functions. (*Ef* translates expressions such as `1+2` to `(+) 1 2` and represents `(+)` as `\x y->x+y` whereas BOHM instantiates applications of primitive functions directly in place). The number of additional $\beta$-reductions required for *Ef* is bounded by the number of syntactic occurrences of primitive functions.

### 6.7.1 Prime

```
-- The following program computes Erathostenes' sieve.

-- starting approximation function
constOne n = 1

-- (min g n m) is the n-th input value k of g (larger than m)
-- such that (g k)<>0
min g n m = if (g m) == 0 then min g n (m+1)
            else if n == 0 then m
            else min g (n-1) (m+1)

-- (minIn g n) is the n-th input value k of g such that (g k) <> 0
minIn g n = min g n 1

-- criv computes the next approximation function in
-- Erathostenes' sieve
criv n g x =
 let a = minIn g n
 in if (x `mod` a) /= 0 || (x == a) then g x else 0

-- (iterate n g f) = (g n (g n-1 (g n-2 (.... (g 1 f)...))))
iterate n g f = if n == 0 then f
                else g n (iterate (n-1) g f)

-- (prime n x) is 1 if x is prime w.r.t. the n first prime numbers,
-- and 0 otherwise
prime n = iterate n criv constOne
```

Figure 6.17: Prime numbers program.

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `prime 2 7` | 0.20s | 0.21s | 0.21s | 0.30s | 0.04s |
| `prime 2 50` | 0.21s | 0.18s | 0.21s | 0.34s | 0.04s |
| `prime 4 15` | 1.32s | 0.23s | 0.28s | 1.19s | 0.02s |
| `prime 5 3500` | 13.38s | 0.29s | 0.37s | 3.02s | 0.04s |
| `prime 6 20` | 184.74s | 0.34s | 0.47s | 6.40s | 0.06s |
| `prime 7 49` | 3214.39s | 0.43s | 0.58s | 13.22s | 0.05s |
| `prime 10 50` | — | 0.83s | 1.32s | 77.04s | 0.11s |

Table 6.18: Prime (time).

The Prime program is taken from Example 7.1 in [10]. The expression `prime n x` evaluates to `1` if `x` is prime with respect to the first `n` prime numbers, and `0` otherwise.

To understand better how the program works, consider the expression `prime 3 x`. This expression can be reduced to `criv 3 (criv 2 (criv 1 constOne)) x`.

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `prime 2 7` | 274 | 172 | 59 | 59 | 28 |
| `prime 2 50` | 275 | 173 | 59 | 59 | 28 |
| `prime 4 15` | 12191 | 701 | 82 | 82 | 49 |
| `prime 5 3500` | 146855 | 1287 | 96 | 96 | 63 |
| `prime 6 20` | 2076167 | 2125 | 112 | 112 | 79 |
| `prime 7 49` | 37370515 | 3319 | 132 | 132 | 99 |
| `prime 10 50` | — | 9619 | 212 | 212 | 179 |

Table 6.19: Prime (beta).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `prime 2 7` | 8.6MB | 8.6MB | 11.6MB | 11.7MB | 2.1MB |
| `prime 2 50` | 8.6MB | 8.6MB | 11.6MB | 11.7MB | 2.1MB |
| `prime 4 15` | 12.4MB | 11.3MB | 11.6MB | 16.0MB | 2.1MB |
| `prime 5 3500` | 18.3MB | 11.3MB | 11.6MB | 18.6MB | 2.2MB |
| `prime 6 20` | 21.1MB | 11.3MB | 13.0MB | 23.5MB | 2.3MB |
| `prime 7 49` | 31.0MB | 11.3MB | 13.0MB | 39.1MB | 2.4MB |
| `prime 10 50` | — | 11.8MB | 15.2MB | 147.1MB | 2.8MB |

Table 6.20: Prime (space).

The function `criv`, computes the `n`th approximation of Erathostenes' sieve from the `n-1`th approximation.

The function starting the approximation, `constOne`, is so approximate it considers all numbers to be prime. The function computed by `criv 1 constOne` determines whether its next argument is divisible by two, and if not, whether it satisfies `constOne`. Each successive approximation considers whether its argument is divisible by the `n`th prime number and if not whether it satisfies the `n-1`th approximation.

The reason lazy evaluation has such a hard time evaluating this program is because of the definition of `criv`. If the definition is changed to:

```
criv n g =
 let a = minIn g n in
 \x->
 if (x `mod` a) /= 0 || (x == a) then g x else 0
```

Then lazy evaluation is able to proceed unhindered.

This small change to the program makes it possible for the function `criv` to compute the `n`th prime number once before `x` is bound to a value rather than many times after `x` has been bound to a value. This change is automatically achieved by a full laziness transformation. Complete laziness manages to perform the reduction of `a` before `x` is bound.

### 6.7.2 Transclos

```
-- This file contains an example of function for computing the
-- transitive closure of a graph based on Roy-Warshall algorithm.
-- Nodes are supposed to be integers, and the graph is represented
-- by the characteristic function f of its edges, i.e. f(n,m) = 1
-- iff there is an edge from n to m (and 0 otherwise).

-- (iterate n g f) = (g n (g n-1 (g n-2 (... (g 1 f)))))
iterate n g f = if n == 0 then f
                else g n (iterate (n-1) g f)

-- Roy-Warshall's function phi
phi n g a b = let ga = g a
              in if ga n == 1 && g n b == 1 then 1
                 else ga b

-- transitive closure of a graph with n nodes
tranclos n = iterate n phi

-- the following function represents a graph where each node n
--  is connected to its predecessor
g n m = if n == m+1 then 1 else 0
```

Figure 6.18: Transclos.

The Transclos program is taken from Example 7.2 in [10]. The expression `transclos 5 g 3 2` can be reduced to

`phi 5 (phi 4 (phi 3 (phi 2 (phi 1 g)))) 3 2`.

The function `g` represents the directed graph where each node `n` is connected to node `n-1`. The function computed by `phi` $n$ `( ... (phi 1 g) ... )` represents the directed graph where nodes are connected if they are connected either directly or via any of the nodes $n..1$ in graph `g`. For a graph `g` with `n` nodes in, `transclos n g` computes the transitive closure.

The key to making the program amenable to lazy evaluation is to change the

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `transclos 5 g 3 2` | 0.24s | 0.22s | 0.28s | 0.30s | 0.01s |
| `transclos 5 g 5 4` | 0.25s | 0.23s | 0.27s | 0.40s | 0.05s |
| `transclos 10 g 2 6` | 1.59s | 0.22s | 5.59 | 0.55s | 0.03s |
| `transclos 15 g 5 10` | 63.79s | 0.29s | — | 1.36s | 0.05s |
| `transclos 18 g 17 18` | 549.89s | 0.71s | — | 2.95s | 0.06s |
| `transclos 20 g 5 15` | 1985.42s | 0.35s | — | 2.51s | 0.05s |
| `transclos 20 g 20 1` | 1622.47s | 0.72s | — | 4.07s | 0.05s |

Table 6.21: Tranclos (time).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `transclos 5 g 3 2` | 917 | 315 | 144 | 61 | 39 |
| `transclos 5 g 5 4` | 1067 | 434 | 154 | 65 | 43 |
| `transclos 10 g 2 6` | 23161 | 615 | 2639 | 84 | 62 |
| `transclos 15 g 5 10` | 1030325 | 1849 | — | 115 | 93 |
| `transclos 18 g 17 28` | 8912661 | 7671 | — | 154 | 132 |
| `transclos 20 g 5 15` | 32964849 | 2744 | — | 140 | 118 |
| `transclos 20 g 20 1` | 26738863 | 9363 | — | 170 | 148 |

Table 6.22: Tranclos (beta).

definition of `phi`:

```
phi n g a =
 let ga = g a
     gan1 = ga n==1
 in
 \b->
 if gan1 && g n b == 1 then 1 else ga b
```

This change ensures that evaluation of `ga n==1` is shared across multiple bindings of `b`. Full laziness achieves this transformation automatically. It is interesting to note that this example demonstrates complete laziness achieving a less effective degree of sharing than full laziness.

When evaluating the transclos program with a completely lazy evaluator, the expression `ga n==1` is substituted by bindings for `n`, `g` and `a` and reduced as far as possible before bindings for `b` are substituted. However this doesn't enable `ga n==1` to be fully reduced before various bindings for `b` are substituted. The reason for this is that the variable `a` is bound to a variable, and *this* variable is only bound after various bindings for `b` have duplicated the expression `ga n==1`.

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `transclos 5 g 3 2` | 8.6MB | 8.6MB | 11.6MB | 11.6MB | 2.1MB |
| `transclos 5 g 5 4` | 8.6MB | 8.6MB | 11.6MB | 12.6MB | 2.1MB |
| `transclos 10 g 2 6` | 11.8MB | 11.3MB | 22.9MB | 12.6MB | 2.1MB |
| `transclos 15 g 5 10` | 20.0MB | 11.4MB | — | 16.5MB | 2.2MB |
| `transclos 18 g 17 18` | 20.5MB | 11.8MB | — | 17.5MB | 2.3MB |
| `transclos 20 g 5 15` | 21.1MB | 11.4MB | — | 17.8MB | 2.2MB |
| `transclos 20 g 20 1` | 20.9MB | 11.8MB | — | 22.5MB | 2.3MB |

Table 6.23: Tranclos (space).

### 6.7.3 Mergesort

```
-- This file contains a mergesort algorithm operating over arrays
-- of integers represented as functions, i.e. a[i] = a(i).

-- merge of two "arrays"
merge f1 f2 i =
 let f11 = f1 1; f21 = f2 1
 in if f11 < f21
    then if i == 1 then f11 else merge (\x->f1 (x+1)) f2 (i-1)
    else if i == 1 then f21 else merge f1 (\x->f2 (x+1)) (i-1)

-- mergesort
mergesort f m n =
 if m == n then (\x->if x == 1 then f m else 10000)
 else let half = (m+n) `div` 2
        f1 = mergesort f m half
        f2 = mergesort f (half+1) n
      in merge f1 f2

-- examples of "arrays". In all examples, integers are
-- initially presented in reverse order.
n20 x = 21-x; n40 x = 41-x; n50 x = 51-x; n60 x = 61-x; n70 x = 71-x

test1 = mergesort n20 1 20 10;    test2 = mergesort n20 1 20 20
test3 = mergesort n40 1 40 15;    test4 = mergesort n40 1 40 30
test5 = mergesort n40 1 40 40;    test6 = mergesort n50 1 50 25
test7 = mergesort n50 1 50 40;    test8 = mergesort n50 1 50 50
test9 = mergesort n60 1 60 60
```

Figure 6.19: Mergesort.

| Test | | | *Ef* | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `test1` | 4.30s | 0.59s | 0.85s | 5.17s | 0.07s |
| `test2` | 20.46s | 0.83s | 1.35s | 11.49s | 0.10s |
| `test3` | 51.77s | 0.94s | 1.62s | 13.34s | 0.13s |
| `test4` | 357.70s | 1.66s | 3.11s | 32.28s | 0.20s |
| `test5` | 698.62s | 2.29s | 4.59s | 55.69s | 0.31s |
| `test6` | 440.01s | 1.64s | 3.14s | 32.48s | 0.19s |
| `test7` | 1431.91s | 2.48s | 4.84s | 61.13s | 0.29s |
| `test8` | 2402.23s | 3.29s | 6.52s | 92.05s | 0.44s |
| `test9` | — | 4.53s | 9.16s | 153.75s | 0.51s |

Table 6.24: Mergesort (time).

The mergesort example is taken from Example 7.3 of [10]. The function `mergesort` takes as arguments an array to sort, and two indices into the array indicating the region to be sorted. Arrays are represented by functions taking an array index as their argument and evaluating to the contents of the array. When

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `test1` | 48082 | 3297 | 228 | 167 | 131 |
| `test2` | 241104 | 7399 | 392 | 207 | 171 |
| `test3` | 632291 | 7607 | 391 | 263 | 227 |
| `test4` | 4447842 | 17585 | 709 | 333 | 297 |
| `test5` | 8579516 | 26382 | 1016 | 377 | 341 |
| `test6` | 5488237 | 16540 | 661 | 349 | 313 |
| `test7` | 17878176 | 28543 | 1040 | 421 | 385 |
| `test8` | 29967694 | 39856 | 1416 | 465 | 429 |
| `test9` | — | 56175 | 1866 | 559 | 523 |

Table 6.25: Mergesort (beta).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `test1` | 15.1MB | 12.0MB | 14.5MB | 22.7MB | 2.5MB |
| `test2` | 19.6MB | 13.4MB | 15.9MB | 37.9MB | 2.9MB |
| `test3` | 19.7MB | 13.6MB | 16.9MB | 35.8MB | 2.9MB |
| `test4` | 19.7MB | 14.6MB | 18.0MB | 68.5MB | 3.9MB |
| `test5` | 19.7MB | 15.6MB | 21.3MB | 111.1MB | 5.0MB |
| `test6` | 20.5MB | 14.8MB | 18.0MB | 70.5MB | 3.8MB |
| `test7` | 20.5MB | 16.0MB | 21.2MB | 115.5MB | 5.1MB |
| `test8` | 21.7MB | 16.0MB | 21.2MB | 151.8MB | 6.4MB |
| `test9` | — | 17.7MB | 27.3MB | 239.1MB | 8.2MB |

Table 6.26: Mergesort (space).

`mergesort` is applied to four arguments, the first three are arguments to `mergesort` and the fourth is an argument to the array computed by `mergesort`.

The key to making the mergesort program amenable to lazy evaluation is to change the `merge` function:

```
merge f1 f2 =
 let f11 = f1 1; f21 = f2 1
     merge1 = merge (\x->f1 (x+1)) f2
     merge2 = merge f1 (\x->f2 (x+1))
 in \i-> if f11 < f21
         then if i == 1 then f11 else merge1 (i-1)
         else if i == 1 then f21 else merge2 (i-1)
```

This change is performed automatically by a full laziness transformation. Without this change, the arrays computed by `merge` are recomputed every time they are indexed.

Complete laziness achieves the same effect by reducing the expressions that are computationally independent of `i` before a binding of `i` is substituted though the body of `merge`.

### 6.7.4 Tartaglia

```
-- row 0
init x = if x == 1 then 1 else 0

-- the function "eval" evaluates an input function f in the interval 0-n
eval f x = eval'
           where eval' n = if n == 0 then 0
                           else if x == n then (f n)
                           else (eval' (n-1))

-- next row in tartaglia's triangle
next f x = f (x-1) + f x

-- tartaglia m n gives the n-th element in the m-th row of
-- tartaglia's triangle
tartaglia m =
 if m == 0 then init
 else \x->eval (next (tartaglia (m-1))) x (m+1)
```

Figure 6.20: Tartaglia.

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `tartaglia 9 5` | 1.57s | 1.13s | 0.35s | 1.59s | 0.05s |
| `tartaglia 13 7` | 19.72s | 13.37s | 0.47s | 4.15s | 0.04s |
| `tartaglia 17 9` | 276.88s | 189.33s | 0.62s | 8.18s | 0.09s |
| `tartaglia 20 10` | 1983.01s | 1378.92s | 0.79s | 13.39s | 0.12s |
| `tartaglia 23 12` | — | — | 1.00s | 19.17s | 0.14s |
| `tartaglia 35 18` | — | — | 2.53s | 70.20s | 0.40s |
| `tartaglia 40 20` | — | — | 3.66s | 112.13s | 0.58s |

Table 6.27: Tartaglia (time).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `tartaglia 9 15` | 21072 | 16332 | 102 | 57 | 27 |
| `tartaglia 13 7` | 302603 | 233853 | 156 | 65 | 35 |
| `tartaglia 17 9` | 4414984 | 3415848 | 226 | 73 | 43 |
| `tartaglia 20 10` | 32164160 | 25040982 | 288 | 79 | 49 |
| `tartaglia 23 12` | — | — | 361 | 85 | 55 |
| `tartaglia 35 18` | — | — | 739 | 109 | 79 |
| `tartaglia 40 20` | — | — | 938 | 119 | 89 |

Table 6.28: Tartaglia (beta).

The Tartaglia example is taken from Example 7.4 in [10]. The expression `tartaglia m x` computes the `x`th element on the `m`th row of Tartaglia's triangle [76] also known as Pascal's triangle [66] and Yang Hui's triangle [83].

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `tartaglia 9 5` | 11.8MB | 11.9MB | 11.6MB | 15.3MB | 2.2MB |
| `tartaglia 13 7` | 16.7MB | 15.2MB | 12.3MB | 20.2MB | 2.3MB |
| `tartaglia 17 9` | 20.0MB | 20.1MB | 12.3MB | 24.4MB | 2.5MB |
| `tartaglia 20 10` | 20.8MB | 19.9MB | 14.1MB | 29.9MB | 2.8MB |
| `tartaglia 23 12` | — | — | 14.1MB | 38.1MB | 3.0MB |
| `tartaglia 35 18` | — | — | 15.9MB | 107.2MB | 4.8MB |
| `tartaglia 40 20` | — | — | 16.6MB | 157.3MB | 6.1MB |

Table 6.29: Tartaglia (space).

The `eval` function in the Tartaglia program performs a form of memoization when evaluated with a completely lazy evaluator. It takes a function `f`, an argument, `x`, and an upper limit, `n`, on the domain of `f`. To compute its result, `eval` creates a chain of conditionals comparing the value of `x` with numbers from `n` downto 0, when a match is found `f` is computed with that found value.

The definition of `eval` contains `if x==n then f n else` ... . The use of `f n` instead of `f x` is crucial! Although the variable `x` is bound before `n` in the definition of `eval`, in the location where `eval` is called, a value for `n` is bound while `x` is bound to the `\x->` in the definition of `tartaglia`. It is beneath this `\x->` that the memoizing chain of conditionals will be built. As lazy and fully lazy evaluators do not perform reductions under $\lambda$'s, they are not able to perform memoization in this way.

To enable lazy and fully lazy evaluation to benefit from the same memoization, `eval` can be rewritten (somewhat more simply):

```
eval f =
 let table=map f (from 0)
 in \x n-> if x==0 || x>n then 0 else table!!x
```

Where `from`, `!!`, and `map` are defined in the usual way:

```
from n = n:from (n+1)
(l1:ls) !! n = if n==0 then l1 else ls!!(n-1)
map f x@(x1:xs) = if x==[] then [] else f x1:map f xs
```

### 6.7.5 Church numerals

```
I x = x

zero  f x = x
one   f x = f x
two   f x = f(f x)
three f x = f(f(f x))
...

succ n x y = x (n x y)
add m n x y = m x (n x y)
mult n m x = n (m x)
pair x y z = z x y
fst x y = x
snd x y = y

nextfact p = let n1 = p fst
                 n2 = succ (p snd)
             in pair (mult n1 n2) n2

fact n = n nextfact (pair one zero) fst

nextfibo p = let n1 = p fst
                 n2 = p snd
             in pair (add n1 n2) n1

fibo n = n nextfibo (pair zero one) fst
```

Figure 6.21: Church numeral programs.

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `f one` | 0.43s | 0.42s | 0.43s | 0.46s | 0.05s |
| `f two` | 0.42s | 0.40s | 0.45s | 0.45s | 0.04s |
| `f three` | 0.41s | 0.44s | 0.45s | 0.40s | 0.04s |
| `f four` | 3.68s | 3.76s | 32.38s | 0.46s | 0.05s |
| `f five` | — | — | — | 0.46s | 0.06s |
| `f six` | — | — | — | 0.54s | 0.05s |
| `f seven` | — | — | — | 0.55s | 0.07s |
| `f eight` | — | — | — | 0.65s | 0.07s |
| `f nine` | — | — | — | 1.02s | 0.18s |
| `f ten` | — | — | — | 1.82s | 0.63s |
| `f eleven` | — | — | — | 3.63s | 2.39s |
| `f twelve` | — | — | — | 7.35s | 10.39s |
| `f thirteen` | — | — | — | 15.86s | 47.55s |
| `f fourteen` | — | — | — | 35.46s | 228.50s |
| `f fifteen` | — | — | — | 77.21s | 984.82s |

Table 6.30: `f x = x two two I I` (time).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `f one` | 16 | 16 | 16 | 14 | 14 |
| `f two` | 45 | 45 | 37 | 19 | 19 |
| `f three` | 534 | 534 | 292 | 24 | 24 |
| `f four` | 131111 | 131111 | 65599 | 29 | 29 |
| `f five` | — | — | — | 34 | 34 |
| `f six` | — | — | — | 39 | 39 |
| `f seven` | — | — | — | 44 | 44 |
| `f eight` | — | — | — | 49 | 49 |
| `f nine` | — | — | — | 54 | 54 |
| `f ten` | — | — | — | 59 | 59 |
| `f eleven` | — | — | — | 64 | 64 |
| `f twelve` | — | — | — | 69 | 69 |
| `f thriteen` | — | — | — | 74 | 74 |
| `f fourteen` | — | — | — | 79 | 79 |
| `f fifteen` | — | — | — | 84 | 84 |

Table 6.31: `f x = x two two I I` (beta).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `f one` | 11.3MB | 11.4MB | 11.6MB | 11.6MB | 2.2MB |
| `f two` | 11.3MB | 11.4MB | 11.6MB | 11.6MB | 2.2MB |
| `f three` | 11.3MB | 11.4MB | 11.6MB | 11.6MB | 2.2MB |
| `f four` | 13.5MB | 13.5MB | 47.2MB | 11.6MB | 2.2MB |
| `f five` | — | — | — | 11.6MB | 2.2MB |
| `f six` | — | — | — | 11.6MB | 2.2MB |
| `f seven` | — | — | — | 12.1MB | 2.2MB |
| `f eight` | — | — | — | 12.1MB | 2.2MB |
| `f nine` | — | — | — | 13.4MB | 2.2MB |
| `f ten` | — | — | — | 14.6MB | 2.3MB |
| `f eleven` | — | — | — | 18.4MB | 2.4MB |
| `f twelve` | — | — | — | 23.5MB | 2.6MB |
| `f thirteen` | — | — | — | 35.3MB | 3.0MB |
| `f fourteen` | — | — | — | 62.1MB | 3.9MB |
| `f fifteen` | — | — | — | 114.6MB | 5.6MB |

Table 6.32: `f x = x two two I I` (space).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `f one` | 0.40s | 0.43s | 0.43s | 0.41s | 0.03s |
| `f two` | 0.40s | 0.42s | 0.44s | 0.48s | 0.04s |
| `f three` | — | — | — | 0.49s | 0.04s |
| `f four` | — | — | — | 0.85s | 0.20s |
| `f five` | — | — | — | 10.91s | 44.59s |
| `f six` | — | — | — | 266.57s | — |

Table 6.33: `f x = x x x I I` (time).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `f one` | 10 | 10 | 10 | 10 | 10 |
| `f two` | 45 | 45 | 37 | 19 | 19 |
| `f three` | — | — | — | 40 | 40 |
| `f four` | — | — | — | 79 | 79 |
| `f five` | — | — | — | 142 | 142 |
| `f six` | — | — | — | 235 | — |

Table 6.34: `f x = x x x I I` (beta).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `f one` | 11.3MB | 11.4MB | 11.6MB | 11.6MB | 2.2MB |
| `f two` | 11.3MB | 11.4MB | 11.6MB | 11.6MB | 2.2MB |
| `f three` | — | — | — | 11.6MB | 2.2MB |
| `f four` | — | — | — | 13.2MB | 2.3MB |
| `f five` | — | — | — | 26.7MB | 3.7MB |
| `f six` | — | — | — | 260.7MB | — |

Table 6.35: `f x = x x x I I` (space).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `fact one I I` | 0.39s | 0.46s | 0.39s | 0.41s | 0.05s |
| `fact three I I` | 0.41s | 0.42s | 0.44s | 0.46s | 0.06s |
| `fact five I I` | 0.45s | 0.45s | 0.47s | 0.47s | 0.03s |
| `fact seven I I` | 1.14s | 0.73s | 3.62s | 0.51s | 0.07s |
| `fact nine I I` | 60.70s | 23.46s | — | 0.59s | 0.04s |
| `fact ten I I` | 607.02s | 239.11s | — | 0.67s | 0.05s |
| `fact twenty I I` | — | — | — | 2.03s | 0.09s |

Table 6.36: Factorials (time).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `fact one I I` | 28 | 28 | 28 | 28 | 30 |
| `fact three I I` | 80 | 77 | 64 | 53 | 55 |
| `fact five I I` | 540 | 402 | 292 | 82 | 84 |
| `fact seven I I` | 17848 | 11963 | 7756 | 115 | 117 |
| `fact nine I I` | 1227476 | 818408 | — | 152 | 154 |
| `fact ten I I` | 12113890 | 8076032 | — | 172 | 174 |
| `fact twenty I I` | — | — | — | 427 | 429 |

Table 6.37: Factorials (beta).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `fact one I I` | 11.3MB | 11.4MB | 11.6MB | 11.6MB | 2.2MB |
| `fact three I I` | 11.3MB | 11.4MB | 11.6MB | 11.6MB | 2.2MB |
| `fact five I I` | 11.3MB | 11.4MB | 11.6MB | 11.6MB | 2.2MB |
| `fact seven I I` | 11.6MB | 11.8MB | 16.7MB | 12.0MB | 2.2MB |
| `fact nine I I` | 28.4MB | 34.1MB | — | 12.1MB | 2.2MB |
| `fact ten I I` | 137.8MB | 107.6MB | — | 12.1MB | 2.2MB |
| `fact twenty I I` | — | — | — | 15.8MB | 2.2MB |

Table 6.38: Factorials (space).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `fibo one I I` | 0.37s | 0.38s | 0.36s | 0.34s | 0.05s |
| `fibo four I I` | 0.37s | 0.34s | 0.37s | 0.38s | 0.04s |
| `fibo seven I I` | 0.37s | 0.41s | 0.40s | 0.44s | 0.03s |
| `fibo ten I I` | 0.40s | 0.40s | 0.54s | 0.60s | 0.04s |
| `fibo thirteen I I` | 0.50s | 0.50s | 1.11s | 1.57s | 0.06s |
| `fibo sixteen I I` | 0.92s | 0.83s | 4.57s | 7.30s | 0.13s |
| `fibo nineteen I I` | 2.61s | 2.56s | 22.73s | 37.46s | 0.40s |

Table 6.39: Fibonacci (time).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| s | | $\beta$ | | | |
| `fibo one I I` | 26 | 26 | 26 | 26 | 28 |
| `fibo four I I` | 85 | 85 | 71 | 61 | 63 |
| `fibo seven I I` | 232 | 232 | 166 | 104 | 106 |
| `fibo ten I I` | 747 | 747 | 461 | 179 | 181 |
| `fibo thirteen I I` | 2822 | 2822 | 1604 | 390 | 392 |
| `fibo sixteen I I` | 11505 | 11505 | 6339 | 1177 | 1179 |
| `fibo nineteen I I` | 48180 | 48180 | 26290 | 4404 | 4406 |

Table 6.40: Fibonacci (beta).

| Test | *Ef* | | | | BOHM |
|---|---|---|---|---|---|
| | Lazy | Full | Complete | Optimal | |
| `fibo one I I` | 11.5MB | 11.3MB | 11.5MB | 11.5MB | 2.2MB |
| `fibo four I I` | 11.5MB | 11.3MB | 11.5MB | 11.5MB | 2.2MB |
| `fibo seven I I` | 11.5MB | 11.3MB | 11.5MB | 11.5MB | 2.2MB |
| `fibo ten I I` | 11.5MB | 11.3MB | 12.0MB | 12.1MB | 2.2MB |
| `fibo thirteen I I` | 11.5MB | 11.8MB | 14.2MB | 15.8MB | 2.2MB |
| `fibo sixteen I I` | 11.8MB | 12.4MB | 18.8MB | 24.0MB | 2.4MB |
| `fibo nineteen I I` | 13.0MB | 15.8MB | 49.3MB | 72.6MB | 3.3MB |

Table 6.41: Fibonacci (space).

The Church numeral experiments are taken from §9.5 of [10]. The Church numeral representation of a number $n$ is a function which applies its first argument $n$-times to its second argument. Examples are shown in Figure 6.21. By supplying `inc` and `0` as the arguments (where `inc` is `\x->x+1`), it is possible to convert a Church numeral into the number it represents. The time taken to perform this conversion is proportional to the number represented.

In the examples shown the arguments used to force the evaluation of the Church numerals are `I I`. For lazy and fully lazy evaluation, the use of `inc 0` or `I I` makes little difference. For optimal evaluation however, the ability to reduce `x I` to `I`, where `x` is a Church numeral, makes all the difference.

If the Church numerals `n` and `m` represent the numbers $n$ and $m$, then the function computed by `n m` represents the number $m^n$. The number represented by the Church numerals computed by `x x x`, is $x^{x^x}$. Thus the numbers represented by the Church numerals computed in Figure 6.33 are: 1, 16, 19683, $4.29 \times 10^9$, $2.98 \times 10^{17}$ and $1.03 \times 10^{28}$.

To understand why optimal evaluation has such an advantage over lazy, fully lazy and completely lazy reduction orders, it is useful to consider the normal-order reduction of `two two I I`.

```
0:  (\f x->f(f x))(\f x->f(f x))(\y->y)(\y->y)
1:  (\x f->\x->f(f x))((\f x->f(f x))x))(\y->y)(\y->y)
2:  (\f x->f(f x))((\f x->f(f x))(\y->y))(\y->y)
3:  (\x->(\f x->f(f x))(\y->y)((\f x->f(f x))(\y->y)x))(\y->y)
4:  (\f x->f(f x))(\y->y)((\f x->f(f x))(\y->y)(\y->y))
5:  (\x y->y)((\y->y)x))((\f x->f(f x))(\y->y)(\y->y))
6:  (\y->y)((\y->y)((\f x->f(f x))(\y->y)(\y->y)))
7:  (\y->y)((\f x->f(f x))(\y->y)(\y->y))
8:  (\f x->f(f x))(\y->y)(\y->y)
9:  (\x->(\y->y)((\y->y)x))(\y->y)
10: (\y->y)((\y->y)(\y->y))
11: (\y->y)(\y->y)
12: \y->y
```

This can be viewed more concisely as:

```
0:  two two I I
2:  two (two I) I
4:  two I (two I I)
6:  I (I (two I I))
7:  I (two I I)
8:  two I I
10: I (I I)
11: I I
12: I
```

This normal-order evaluation takes 12 $\beta$-reductions, a lazy evaluator takes 11, as the application of `two I` in $\beta$-reduction 5 and 9 is shared. This application results in `(\x->I(I x))`. Clearly there would be benefit in performing the `I` applications in-place under the `\x->`. Optimal evaluation achieves this and saves two $\beta$-reductions as the `I`'s are reduced once before `(\x->I(I x))` is applied and not twice afterward. Completely lazy evaluation however does not achieve this sharing. The reason for this is that the application `two I` is never formed in the completely lazy evaluation. The problem is that completely lazy evaluation specializes the first `two` to the second `two` resulting in `four`, preventing the application `two I` ever coming into existence.

The reductions performed by completely lazily evaluating `two two` are:

```
0: (\f x->f(f x))(\g y->g(g y))
1: \x->(\g y->g(g y))((\g y->g(g y))x)
2: \x->(\g y->g(g y))(\y->x(x y))
3: \x y->(\y->x(x y))((\y->x(x y))y)
4: \x y->(\y->x(x y))(x(x y))
5: \x y->x(x(x(x y)))
```

To avoid showing the complicated sharing mechanisms, which for the present discussion would not help, the reductions above are not shown in the order the completely lazy evaluator performs them. The order shown above is the order in which an evaluator with the impossible foresight to know which arguments and which redexes in a function body can be reduced without performing unneeded reductions. The important factor here is *which* reductions are performed, not the order.

In the reduction of `two two I I`, reducing `two two` to `four` does not save any work, as this function is only used once. In the evaluation of `two two two I I` however terms such as `two two` are applied more than once and so are worth specializing. There is a conflict between specializing `two two`, and leaving it unspecialized so as to specialize `two I` later. The benefits from specializing `two two` are small compared to specializing `two I`. To reduce terms of the form `x x x I I` (where `x` is a Church numeral) effectively, nothing as complex as optimal evaluation is required. Just ensuring the reduction of `x I` to `I` is the only specializing reduction performed is sufficient.

In the Fibonacci example, the optimal evaluators have no great advantage over the other reduction orders. The reason for this can be seen in the way in which the Church numerals are combined. In the other examples the Church numeral operations have been exponentiation and multiplication. For both these operations multiple copies of Church numerals are made. However in the Fibonacci example

the only operation performed on Church numerals is addition. Inspection of the definition of `add` reveals that it does not make multiple copies of its arguments, so there is little sharing for an optimal evaluator to exploit.

## 6.8 Summary

The experiments presented in this chapter were chosen firstly to demonstrate the unique capabilities of a completely lazy evaluator, and secondly to demonstrate the apparent correctness of the optimal implementation of *Ef*.

The experiments chosen to test the optimal implementation of *Ef* include all the experiments previously used to demonstrate the power of BOHM. Analysis of these experiments and further testing of these experiments with fully lazy and completely lazy evaluators reveal how little is understood about the potential uses of an optimal evaluator.

This chapter has demonstrated how the *completely lazy Ef* can eliminate layers of interpretation, both for functional layers and imperative layers. It has been demonstrated that the optimal *Ef* reduces terms in essentially the same number of $\beta$-reductions as BOHM. A contribution to the understanding of when an *optimal* evaluator might be useful has been made by demonstrating that current implementations do not eliminate layers of interpretation.

Complete laziness has some interesting properties. It has never been this easy to eliminate layers of interpretation before. It also has some serious drawbacks, as absolutely everything is specialized. In partial evaluation the decision of what to specialize and what to residualize is crucial; if too much is specialized the partial evaluator may not terminate, if too little is specialized; there may not be much improvement in the specialized program. By performing the specialization only when needed, the specialize-residualize decision is changed from a semantic decision to an operational decision. However always choosing to specialize is not sensible operationally. For any long-running program, in which recursive function calls are used a lot, the memo-tables will grow very large, and execution painfully slow.

# Chapter 7

# Conclusions

The lambda calculus supports many possible reduction strategies but does not handle sharing and cycles. Functional language implementations handle sharing and cycles but only support top-level reduction strategies. The novel results in this thesis bridge the gap between the lambda calculus and functional language implementation techniques more elegantly than has been done before. However the research was originally intended to take place very much on the language implementation side. The earliest version of *Ef* with specializing capabilities was based on the G-machine [41]. This implementation was unable to pass the tower of interpreters test. As subsequent layers of functional language implementation refinement were undone, the implementation looked increasingly like a simple lambda-calculus interpreter. It was only when the implementation reached its current state that the tower of interpreters test was passed. How the removed layers of functional language implementation refinement can best be re-applied remains to be seen.

## 7.1 Review of contributions

**Chapter 2: Background**

- *The identification of a difference between static full laziness and dynamic full laziness is original*

This seems like a subtle point, however its discovery was a real break-through for the author. Many late-night hours were spent trying to witness the 'remarkable self-optimizing properties' of Turner's combinators in a dynamic setting, before it was realized that Turner's combinators did not achieve full-laziness via some dynamic

properties, but via the static optimization rules Turner applied.

**Chapter 3: Degrees of Sharing**

- *The use of depth to delimit the scope of a function is original.*

The importance of this crucial break-through was unfortunately discovered at a very late stage of the research. This scheme was originally envisaged as a minor optimization to aid in implementing dynamic full laziness. Although identifying scope dynamically within a lambda graph has been done before [3], assigning natural numbers to the scopes so as to manipulate substitutions is original.

- *The identification of a partial ordering between some degrees of sharing is original.*

This discovery was also a major break-through. Many months were spent mistakenly thinking complete-laziness should subsume full-laziness and maintain dynamic full-laziness. Only when this expectation of complete-laziness was dropped was the tower of interpreters test passed.

- *The classification of reduction strategies in terms of: substitute-by-name, substitute-by-value and substitute-by-need is original. Their analogy to, and orthogonality with, call-by-name, call-by-value and call-by-need is original.*

This classification helps put the results in context. Although numerous ugly implementation attempts were made to solve the tower of interpreters test, it is pleasing to see that the successful technique can retrospectively be explained with reference to this function body reduction strategy classification.

- *The use of memo-tables to achieve call-by-need and substitute-by-need is original.*

The use of memo-tables was rejected at a very early stage in the research. Memo-tables are simple to understand, but they are not a pleasing way to achieve call-by-need and substitute-by-need. More than a year was spent trying to find something more elegant than memo-tables. Trying to devise a scheme which passes the tower of interpreters test without incurring some other overhead such as the infinitely growing memo-tables, feels like trying to stamp out all the bumps in a badly fitting carpet. Perhaps there is some fundamental limitation here, or perhaps some major break-through still remains to be found.

- *Explaining optimal evaluation in terms of the simultaneous specialization of a function to multiple arguments is original.*

This thesis has not attempted to find a use for optimal evaluators. The observation that optimal evaluators may be useful for simultaneous specializing of a function to multiple arguments remains to be explored further.

- *The generalization of a completely lazy evaluator to an optimal evaluator is original.*

This result was achieved more by accident than design, the intention was to find a convincing argument to explain why complete laziness could not be generalized to optimal evaluation.

The correctness of the implementation is unclear. The mechanism used to test if two substitutions can swap may be too crude to cope with the full generality of cyclic terms. The results chapter demonstrates that the implementation produces the correct results in the expected number of beta reductions. The test programs used contain cycles, and no programs are known for which the implementation fails.

**Chapter 4: Reduction Rules**

- *The notation used to represent graphs with memo-tables in a term-like fashion is original.*

Existing notations fail to suitably capture sharing. This new notation may prove to have some lasting significance.

- *The reduction rules for complete laziness and optimal evaluation are original.*

These reduction rules help demonstrate the utility of the new notation, and make precise the reduction techniques explained in the previous chapter.

**Chapter 5: Implementation**

- *The implementation of full laziness by graph transformation is original.*

Dynamic full laziness was originally thought important in passing the tower of interpreters test. However as this has since proved not the be the case, the implementation of full laziness by graph transformation has no great significance.

- *The implementation of complete laziness is the first ever.*

The conciseness of the implementation helps convey how elegant the underlying ideas are.

- *The implementation of optimal evaluation with memo-tables is original.*

The optimal implementation of *Ef* is relatively concise. However questions about its correctness still remain.

**Chapter 6: Results**

- *The first ever evaluator to pass the tower of interpreters test is demonstrated.*

The solid experimental evidence makes it clear the tower of interpreters test has been passed. The overhead is indeed additive and not multiplicative. However there is much scope for significantly reducing this additive overhead.

- *Examples demonstrate that interpreters used in a tower of interpreters may introduce additional features and the interpretive overhead is still eliminated.*

The results are not just an experimental artifact but do indeed work for many language implementations.

- *Existing implementations of optimal evaluators are shown to not pass the tower of interpreters test.*

This result was a surprise. The author had assumed an optimal evaluator would pass the tower of interpreters test, and that the potential benefit of an evaluator with a lesser degree of sharing would be in the opportunities for various constant-factor compiler optimizations.

- *The specializing effect of complete laziness is shown to be inheritable by an imperative language as well as functional languages.*

This result helps demonstrate the wider applicability of the lazy specialization technique. Whether it would be possible to add some language construct to the imperative language such as an explicit memo-table construct so as to enable suitable knot-tying interpreters written in the imperative language to be specialized away is still an open question.

- *Numerous test programs are executed with lazy, fully lazy, completely lazy and optimal evaluators, and the results analysed.*

These experiments have demonstrated how little is known about what optimal evaluators might be useful for. It is not the aim of this thesis to advocate the use of optimal evaluation.

## 7.2 Evaluation

The aim of this research was to contribute to functional language implementation techniques. However the techniques developed currently are not suitable for general use. In retrospect it is clear that the theoretical underpinnings of language implementations were lacking. This research will help provide those underpinnings.

## 7.3 Future work

### Annotations

The most promising direction for future research is the investigation of annotation schemes to control which reductions under lambdas are performed. The annotations used by partial evaluators are not suitable as these annotations cannot be inherited from the specializing language to the language being interpreted. The author has recently accidentally discovered that well placed `seq` function applications can dramatically alleviate the overhead of memo-tables. Ideally it should be possible to avoid using memo-tables at all. Perhaps some form of strictness annotation on function bodies could be used to good effect. Just as strictness annotations give the programmer the freedom to choose between call-by-need and call-by-value, some variant on strictness annotations could be used to enable the programmer to choose between substitute-by-name and substitute-by-value on different parts of a function body. Beyond annotations, it may be fruitful to explore how far binding time analysis can be automated.

### Types

Writing an interpreter in a typed language typically results in an explicit universal type representation. This will introduce a multiplicative overhead if the specializa-

tion techniques presented in this thesis are used. Perhaps Hughes' type specialization [40] or dependent types [16] can come to the rescue.

### Efficient complete laziness

Whether it will ever be possible to implement a completely lazy evaluator able to pass the tower of interpreters test and not have undesirable slow-down inefficiencies such as those seen when using memo-tables is an open question. Implementing a completely-lazy evaluator has been useful in exploring one end of a spectrum of implementation possibilities between never specializing function bodies and always specializing function bodies. It would be appealing to have an evaluator able to pass the tower of interpreters test and still be no more than a constant factor slower than conventional evaluators. From this starting point programmers could consider use the of *residualize* annotations as purely a constant-factor speed-up issue. The author suspects that such a goal may be impossible, and that programmers will still start from the other end of the spectrum and add *specialize* annotations to achieve speed-ups exponential in the number of layers of interpretation.

### Optimal evaluation

This thesis has not tried to justify the use of an optimal evaluator. However some interesting questions have been raised. Is it possible for an optimal evaluator to pass the tower of interpreters test? Is it possible to do this without being more than a constant factor slower than a conventional evaluator? Achieving this would clearly surpass achieving efficient complete laziness, it is not even clear efficient complete laziness is achievable. There may be some benefit to exploring the sort of programs optimal evaluators are good for. The programs which have previously been used to demonstrate the abilities of optimal evaluators did not provide evidence that there was a use for optimal evaluators. Perhaps exploring the issue of specializing a function along multiple dimensions will produce examples demonstrating the potential use for an optimal evaluator.

## 7.4 Final remark

The author is confident that the results in this thesis will contribute to a usable lazily specializing language. However until such an implementation has been achieved the

promise of the results in this thesis remain unproven. Only time will tell how significant the results prove to be.

# Bibliography


[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[2] Zena Ariola and Arvind. Properties of a first-order functional language with sharing. *Theoretical Computer Science*, 146:69–108, 1995.

[3] Zena Ariola and Stefan Blom. Lambda calculi plus letrec. Technical Report IR-434, Department of Mathematics and Computer Science, Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam, 1997.

[4] Zena Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, May 1997.

[5] Zena M. Ariola, J. W. Klop, J. R. Kennaway, F. J. de Vries, and M. R. Sleep. Syntactic definitions of undefined: On defining the undefined. In *TACS 94*, Sendai, Japan, 1994.

[6] Arvind, Vinod Kathail, and Keshav Pingali. Sharing of computations in functional language implementations. In *Proceeding of the International Workshop on High-Level Computer Architecture*, 1984. Also in IFL 1985, published as Chalmers PMG Report 17.

[7] Arvind, J-W. Maessen, R.S. Nikhil, and J.E. Stoy. A lambda calculus with letrecs and barriers. Technical Report 395, Computation Structures Group, LFCS, MIT, January 1997.

[8] A. Asperti, J. Chroboczek, C. Giovannetti, C. Laneve, P. Gruppioni, and A. Naletto. *The BOHM functional interpreter*. Department of Mathematics of the University of Bologna, Bologna, Italy, October 1995. ftp://ftp.cs.unibo.it/pub/asperti/bohm1.1.tar.gz.

[9] Andrea Asperti. $\delta \circ !\epsilon = 1$: Optimizing optimal $\lambda$-calculus implementations. In Jieh Hsiang, editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA-95)*, volume 914 of *LNCS*, pages 102–116, Kaiserslautern, Germany, April 1995. Springer-Verlag.

[10] Andrea Asperti, Cecilia Giovannetti, and Andrea Naletto. The bologna optimal higher-order machine. *Journal of Functional Programming*, 6(6):763–810, November 1996.

[11] Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*. Number 45 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, June 1998.

[12] Andrea Asperti and Cosimo Laneve. Interaction systems II: The practice of optimal reductions. *Theoretical Computer Science*, 159:191–244, 1996.

[13] Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer Verlag, September 1985.

[14] Lennart Augustsson. SMALL – a small interactive functional system. Technical report, Programming Methodology Group, University of Goteborg and Chalmers University of Technology, December 1986.

[15] Lennart Augustsson. *Haskell B. user's manual*, 1993.

[16] Lennart Augustsson. Cayenne — a language with dependent types. Technical report, Department of Computer Science, Chalmers University of Technology, 1998.

[17] Lennart Augustsson and Thomas Johnsson. The Chalmers lazy-ML compiler. *The Computer Journal*, 2(32):127–141, April 1989.

[18] Hendrik Pieter Barendregt. *The lambda calculus: its syntax and semantics*. North-Holland, Amsterdam, revised edition, 1984.

[19] Henk P. Barendregt, J. R. W. Glauert M. C. J. D. van Eekelen, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *Proc. Conference on*

*Parallel Architecture and Languages Europe (PARLE '87)*, LNCS 259, pages 141–158, Eindhoven, The Netherlands, 1987. Springer-Verlag.

[20] H.P. Barendregt, J.R. Kennaway, J.W. Klop, and M.R. Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75:191–231, 1987.

[21] Lennart Beckman, Anders Haraldson, Osten Oskarsson, and Erik Sandwell. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.

[22] Stefan Blom. *Term Graph Rewriting: syntax and semantics*. PhD thesis, Vrije Universiteit, Amsterdam, 2001. http://www.cwi.nl/~sccblom/publications/.

[23] W. H. Burge. *Recursive Programming Techniques*. Addison-Wesley Publishing Company, Reading, Mass, 1975.

[24] Alonzo Church. The calculi of lambda-conversion. *Annals of Mathematical Studies*, 6, 1941.

[25] William Clinger, Jonathan Rees, et al. *Revised* $^4$ *Report on the Algorithmic Language Scheme*, November 1991.

[26] Nicolaas Govert de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Math*, 34:381–92, 1972.

[27] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation : Applications, Languages and Tools*, volume 2. World Scientific, October 1999.

[28] John Henry Field. On laziness and optimality in lambda interpreters: tools for specification and analysis. In *Proceedings of the seventeenth annual ACM symposium on Principles of Programming Languages*, pages 1–15. ACM, January 1990.

[29] John Henry Field. *Incremental Reduction in the Lambda Calculus and Related Reduction Systems*. PhD thesis, Cornell University, 1991.

[30] Yoshihiko Futamura. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems Computers Controls*, 2(5):721–728, August 1971.

[31] Jean-Yves Girard. Linear logic: Its syntax and semantics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 1–42. Cambridge University Press, 1995. ftp://lmd.univ-mrs.fr/pub/girard/Synsem.ps.Z.

[32] Georges Gonthier, Martin Abadi, and Jean-Jacques Levy. The geometry of optimal lambda reduction. In *Symposium on Principles of Programming Languages*, pages 15–26. ACM, 1992.

[33] Peter Henderson, Geraint Jones, and Simon Jones. The LispKit manual. Technical Report PRG-32, Oxford University Computing Laboratory, Programming Research Group, 1983.

[34] Peter Henderson and J. H. Morris. A lazy evaluator. In *Third annual ACM Symposium on Principles of Programming Languages*, pages 95–103, Atlanta, GA, January 1976. ACM.

[35] Carsten Kehler Holst. Improving full laziness. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming*, Workshops in Computing, pages 71–82. Springer-Verlag, 1990.

[36] Carsten Kehler Holst and Carsten Krogh Gomard. Partial evaluation is fuller laziness. *SIGPLAN Notices*, 26(9):223–233, September 1991.

[37] John Hughes. Super-Combinators. In *Lisp and Functional Programming*, pages 1–10. ACM, August 1982.

[38] John Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Oxford, July 1983.

[39] John Hughes. Why functional programming matters. Technical Report PMG report 16, Chalmers University of Technology, Gotumteborg, November 1984.

[40] John Hughes. Type specialisation for the lambda-calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *LNCS*. Springer-Verlag, February 1996.

[41] Thomas Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984. Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction.

[42] Thomas Johnsson. Lambda-lifting – transforming programs to recursive equations. In *Functional Programming and Computer Architecture*, volume 201 of *LNCS*, pages 190–203. Springer Verlag, 1985.

[43] Mark P. Jones. *The Hugs 98 User Manual*, 1999.

[44] Neil Jones, P. Sestoft, and H. Sondergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[45] Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glck, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 216–237. Springer-Verlag, 1996.

[46] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. http://www.dina.kvl.dk/~sestoft/pebook/pebook.html.

[47] Fairouz Kamareddine and Alejandro Ríos. A $\lambda$-calculus á la de bruijn with explicit substitutions. In *7th international symposium on Programming Languages: Implementations, Logics and Programs*, pages 45–62. Springer-Verlag, 1995. LNCS 982.

[48] Fairouz Kamareddine and Alejandro Ríos. Relating the $\lambda\sigma$- and $\lambda$s-styles of explicit substitutions. *Journal of Logic and Computation*, 10(3):349–380, June 2000.

[49] K. Kaneko and Masato Takeichi. Relationship between lambda hoisting and fully lazy lambda lifting. *Journal of Information Processing*, 15(4):564–569, 1992.

[50] Vinod Kumar Kathail. *Optimal Interpreters for Lambda-calculus Based Functional Programming Languages*. PhD thesis, MIT, May 1990.

[51] Stephen Cole Keene. *Mathematical Logic*. Wiley, New York; London, 1967.

[52] Richard Kennaway and Ronan Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(4):602–626, 1988.

[53] Yves Lafont. Interaction nets. In *Proceedings of the 17th Symposium on Principles of Programming Languages*, pages 95–108. ACM Press, January 1990.

[54] John Lamping. An algorithm for optimal lambda calculus reduction. In *Symposium on Principles of Programming Languages*, pages 16–30. ACM, 1990.

[55] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[56] John Launchbury. A strongly-typed self-applicable partial evaluator. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, LNCS 523, pages 145–164. ACM, August 1991.

[57] John Launchbury. A natural semantics for lazy evaluation. *ACM Symposium on Principle of Programming Languages*, 20:144–154, January 1993.

[58] Jean-Jacques Lévy. *Réductions Correctes et Optimales dans le Lambda Calcul*. Thèse de Doctorat d'Etat, University of Paris VII, 1978.

[59] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.

[60] Lionello A. Lombardi. Incremental computation. *Advances in Computers*, 8:247–333, 1967.

[61] Ian Mackie. Yale: Yet another lambda evaluator based on intetaction nets. In *In Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, September 1998.

[62] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, May 1998.

[63] Donald Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, April 1968.

[64] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. Wiley, Chichester, 1992.

[65] Chris Okasaki. *Purely Functional Data Structures*. Cambridge Univeristy Press, 1998.

[66] Blaise Pascal. Trait du triangle arithmtique (Treatise on the arithmetic triangle), 1654.

[67] Simon Peyton Jones, Will Partain, and Andre Santos. Let-floating: Moving bindings to give faster programs. In *SIGPLAN International Conference on Functional Programming*, pages 1–12. ACM, May 1996.

[68] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[69] Simon L. Peyton Jones et al. *Report on the Programming Language Haskell 98. A Non-strict, Purely Functional Language*. http://haskell.org/, February 1999.

[70] Simon L. Peyton Jones and David R. Lester. A modular, fully lazy lambda lifter in Haskell. *Software Practice and Experience*, 21(5):479–506, 1991.

[71] Niklas Rojemo. Highlights from nhc - a space-efficient Haskell compiler. In *Functional Programming Languages and Computer Architecture*, pages 282–292. ACM, June 1995.

[72] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Technical Report PRG-6, Oxford University Computing Laboratory, Programming Research Group, 1977.

[73] Joseph Stoy. *Dentational Semenatics : The Scott-Strachey approach to programming language theory*. MIT Press, Cambridge, Mass; London, 1977.

[74] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 203–217. SIGPLAN, ACM, June 1997.

[75] Masato Takeichi. Lambda-hoisting — a transformation technique for fully lazy evaluation of functional programs. *New Generation Computing*, 5(4):377–391, 1988.

[76] Niccolo Tartaglia. *General trattato di numeri et misure (General treatise of numbers and measures)*, volume 1. Curtio Troiana, Venice, 1556.

[77] Peter Thiemanm. Cogen in six lines. In *International Conference on Functional Programming*, pages 180–189. ACM, May 1996.

[78] Michael Thyer. Lazy specialization. In *Draft Proceedings of the 9th International Workshop on Implementation of Functional Languages*, pages 489–499, St. Andrews, Scotland, September 1997.

[79] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[80] David A. Turner. Another algorithm for bracket abstraction. *The Journal of Symbolic Logic*, 44(2):267–271, June 1979.

[81] David A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[82] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. DPhil thesis, Oxford, 1971.

[83] Hui Yang. Cheng chu tong bian ben mo (Alpha and omega of variations on multiplication and division), 1275.