Adaptive Evaluation of Non-Strict Programs

Robert Jonathan Ennals
King's College

Dissertation submitted for the degree of Doctor of Philosophy
University of Cambridge

July 2, 2004

Declaration

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university. Further, no part of the dissertation has already been or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed 60,000 words. This total includes tables and footnotes, but excludes appendices, bibliography, and diagrams.

Material from Chapters 2, 3 and 4 has been presented in a paper I co-authored with my supervisor [EP03b]. Chapter 5 is largely the same as a paper that I co-authored with my supervisor and with Alan Mycroft [EPM03]. Chapter 11 is largely identical to another paper that I co-authored with my supervisor [EP03a]. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text.

Robert Ennals, July 2, 2004

Abstract

Most popular programming languages are *strict*. In a strict language, the binding of a variable to an expression coincides with the evaluation of the expression.

Non-strict languages attempt to make life easier for programmers by decoupling expression binding and expression evaluation. In a non-strict language, a variable can be bound to an unevaluated expression, and such expressions can be passed around just like values in a strict language. This separation allows the programmer to declare a variable at the point that makes most logical sense, rather than at the point at which its value is known to be needed.

Non-strict languages are usually evaluated using a technique called Lazy Evaluation. Lazy Evaluation will only evaluate an expression when its value is known to be needed. While Lazy Evaluation minimises the total number of expressions evaluated, it imposes a considerable bookkeeping overhead, and has unpredictable space behaviour.

In this thesis, we present a new evaluation strategy which we call Optimistic Evaluation. Optimistic Evaluation blends lazy and eager evaluation under the guidance of an online profiler. The online profiler observes the running program and decides which expressions should be evaluated lazily, and which should be evaluated eagerly. We show that the worst case performance of Optimistic Evaluation relative to Lazy Evaluation can be bounded with an upper bound chosen by the user. Increasing this upper bound allows the profiler to take greater risks and potentially achieve better average performance.

This thesis describes both the theory and practice of Optimistic Evaluation. We start by giving an overview of Optimistic Evaluation. We go on to present a formal model, which we use to justify our design. We then detail how we have implemented Optimistic Evaluation as part of an industrial-strength compiler. Finally, we provide experimental results to back up our claims.

Acknowledgments

First of all, I would like to thank my supervisor, Simon Peyton Jones, who has provided invaluable support, inspiration, and advice. Enormous thanks must go to Alan Mycroft, who has acted as my lab supervisor. I am also extremely grateful for the support of Microsoft Research, who have provided me not only with funding and equipment, but also with a supervisor, and a building full of interesting people.

I would also like to thank Jan-Willem Maessen and Dave Scott, who provided suggestions on this thesis, and Manuel Chakravarty, Karl-Filip Faxén, Fergus Henderson, Neil Johnson, Alan Lawrence, Simon Marlow, Greg Morisett, Nick Nethercote, Nenrik Nilsson, Andy Pitts, Norman Ramsey, John Reppy, Richard Sharp, and Jeremy Singer, who have provided useful suggestions during the course of this work.

This work builds on top of years of work that many people have put into the Glasgow Haskell Compiler. I would like to thank all the people at Glasgow, at Microsoft, and elsewhere, who have worked on GHC over the years. Special thanks must go to Simon Marlow, who was always able to help me when I ran into deep dark corners in the GHC runtime system.

Contents

CHAPTER 1

Introduction

Avoiding work is sometimes a waste of time.

Non-strict languages are elegant, but they are also slow. Optimistic Evaluation is an implementation technique that makes them faster. It does this by using a mixture of Lazy Evaluation and Eager Evaluation, with the exact blend decided by an online profiler.

1.1 Lazy Evaluation

> *Lazy Evaluation is like leaving the washing-up until later. If you are never going to use your plates again then you can save considerable time by not washing them up. In the extreme case you may even avoid washing up a plate that is so dirty that it would have taken an infinite amount of time to get clean.*
>
> *However, if you are going to need to use your plates again, then leaving the washing up until later will waste time. By the time you eventually need to use your plates, the food on them will have stuck fast to the plate and will take longer to wash off. You will also have to make space in your kitchen to hold all the washing up that you have not yet done.*

Non-strict languages aim to make life easier for programmers by removing the need for a programmer to decide if and when an expression should be evaluated. If a programmer were to write the following:

$$\mathbf{let}\ x = E\ \mathbf{in}\ E'$$

then a strict language such as C [KR88] or ML [MTHM97] would evaluate the **let** expression *eagerly*. It would evaluate the E to a value, bind x to this value, and then evaluate E'.

By contrast, a non-strict language such as Haskell [Pey03] or Clean [BvEvLP87, NSvEP91] will not require that E be evaluated until x is known to be needed. Non-strict languages are usually evaluated using an evaluation strategy called *Lazy Evaluation*.[1] If the evaluator evaluates the **let** *lazily* then it will bind x to a description of how to evaluate E, only evaluating E if x is found to be needed.

In a strict language, the `let` declaration does not just say what x means—it also says that x should be evaluated now. This forces the programmer to declare x at a point at which it is known that E is needed; otherwise the work done to evaluate E might be wasted. However, *the best point for x to be evaluated might not be the most logical place for x to be declared*. Programmers are forced to make a compromise between elegance and efficiency in their placement of declarations.

Non-strict languages avoid this problem by distinguishing between the point at which a variable is declared and the point at which it is evaluated. It is argued that this gives programmers more freedom, and allows them to write more elegant programs than they would in a strict language [Hug89].

Unfortunately, this beauty comes at a cost. If the evaluator is not going to evaluate an expression immediately, then it must create a data structure describing how to produce the value later. Producing this data structure takes time and space. As a result of this, non-strict languages are usually significantly slower than strict languages.

1.2 Mixing Eager and Lazy Evaluation

> *Rather than leaving all the washing up until later, or washing up everything now, it might be best to wash up some plates now, but leave some other plates for later. I might decide to wash up the plates that I know I am going to need tomorrow. Alternatively, I might wash up only those plates which I know are fairly clean and so won't take very long to wash up.*
>
> *Unfortunately, it can be difficult to know for certain that a plate will definitely be needed or that it will definitely be easy to clean. I may have used my favourite plate every day this year, but if I got hit by a bus tomorrow then I would never need to use it again. Similarly, although my plate has always been easy to wash up in the past, I can't be sure that it won't be impossible to wash up this time.*

[1] The only non-lazy implementation of a non-strict language of which we are aware is Eager Haskell, which we discuss in Section 13.2.

One way for an implementation to avoid the cost of Lazy Evaluation is to simply not use it. If E is very cheap to evaluate, or is always needed, then it is safe to evaluate x at the point at which it is declared. We can thus arrange to use Eager Evaluation in such cases, and save Lazy Evaluation for the cases where it is necessary.

It is all very well to say "we will only use Lazy Evaluation when it is necessary", but how does a language implementation find out where Lazy Evaluation can be safely replaced by Eager Evaluation?

Most existing compilers for non-strict languages use a static analysis called strictness analysis (Section 13.1.2) to find expressions that are guaranteed to be needed.[2] Another approach is cheapness analysis (Section 13.1.3) which finds expressions that are guaranteed to be cheap to evaluate.

While both these analyses work well they, like all static analyses of undecidable properties, are limited by the need to be conservative. If a static analysis decides to evaluate an expression eagerly, then the analysis must be 100% sure that the expression is either cheap or needed. If the expression turned out to be expensive and unnecessary, then this could cause the program to execute much more slowly than otherwise, or perhaps even to not terminate. This requirement to be conservative can be very frustrating. Often one will have an expression that one is "pretty sure" could be safely evaluated eagerly, but because one is not 100% sure one has to evaluate it lazily. A further problem is that such analyses miss expressions that are "usually used and usually cheap".

In practice this conservatism causes static analyses to lazily evaluate many more expressions than necessary. Figure 1.1 summarises the proportion of lazy **let** evaluations left behind by GHC's [PHH+93] strictness analyser [PP93] in a suite of test programs, that our profiler believes should be evaluated eagerly. This graph is explained in more detail in Section 12.4.4.

Static analyses, and other approaches to mixed eager/lazy evaluation are discussed in more depth in Chapter 13. The implementation of Lazy Evaluation is discussed further in Chapter 7.

1.3 Optimistic Evaluation

> *Perhaps a better approach is to wash up "optimistically". As days go by, I learn which plates are likely to be easy to wash up and which ones are likely to be needed again. I will soon learn that, while paper plates are usually not needed again, my favourite dinner plate usually is. If a plate turns out to be harder to wash up than I expected then I can stop washing it up and move onto something else.*

[2] In this introductory chapter, we are fairly sloppy in distinguishing between static program expressions and dynamic expression instances. We will be more formal in later chapters.

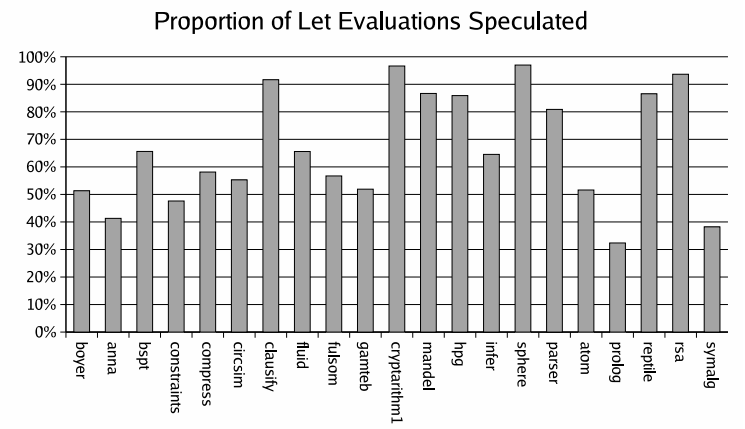


Figure 1.1: Percentage of lazy $\mathbf{let}$ evaluations left behind by GHC's strictness analyser that we believe should be evaluated eagerly

In this thesis, we introduce Optimistic Evaluation, a new evaluation strategy for non-strict programs. Optimistic Evaluation avoids the curse of conservatism by using an online profiler to decide how best to blend lazy and eager evaluation, rather than using a static analysis. Every $\mathbf{let}$ expression is compiled such that it can evaluate either eagerly, or lazily, depending on the state of a dynamically changing structure called the *speculation configuration*. An online profiler monitors the behaviour of the program as it runs, and updates the speculation configuration to take account of what it has seen.

Optimistic Evaluation is a *speculative evaluation strategy*. It evaluates expressions without knowing whether their values are needed or whether evaluation will have to be aborted. When Optimistic Evaluation evaluates the right hand side[3] of a $\mathbf{let}$ eagerly, we say that it is *speculating* the $\mathbf{let}$, and refer to the evaluation of the right hand side as a *speculation* or a *speculative evaluation*.

Unlike a static analysis, Optimistic Evaluation is not conservative. An expression will be evaluated eagerly if the profiler judges that this will *probably* make the program go faster, rather than only if this will *definitely* not make the program go slower. If the profiler makes a bad decision, and causes the program to attempt to eagerly evaluate an expensive expression, then the profiler will spot that the evaluation has gone on for too long, and arrange for it to be aborted (more details in Section 2.2).

The purpose of this thesis is to explore the concept of Optimistic Evaluation, develop the theory behind it, demonstrate that it can be practically implemented, and show that it gives considerable (approx 20%) performance improvements relative to the best performing compiler previously available.

[3]For the expression '$\mathbf{let}\ x = E\ \mathbf{in}\ E'$' we say that x is the *binder*, E is the *right hand side*, and E' is the *body*. While these terms are quite widespread, they can be confusing because, although the right hand side is written to the right of the binder, it is written to the left of the body.

1.4 Structure of this Thesis

This thesis is divided into four parts:

- **Part I:** An overview of Optimistic Evaluation
- **Part II:** The theory behind Optimistic Evaluation
- **Part III:** The implementation of Optimistic Evaluation in the Glasgow Haskell Compiler
- **Part IV:** Performance results, related work, and conclusions.

All terms defined in this thesis can be found in the index.

1.5 Contributions

This thesis makes the following contributions:

- We introduce Optimistic Evaluation, a novel approach to evaluating non-strict programs, which dynamically explores the space between Lazy Evaluation and Eager Evaluation (Chapter 2).
- We give a concrete semantics for Optimistic Evaluation and prove that it is sound with respect to Lazy Evaluation (Chapter 4).
- We give a novel cost semantics for non-strict programs and prove that this semantics correctly models the costs experienced by a conventional operational semantics (Chapter 5).
- We use this cost model to motivate the design of an online profiler (Chapter 6).
- We show that, given certain safe assumptions, we can guarantee that such a profiler can bound the worst case performance of Optimistic Evaluation, relative to Lazy Evaluation. This is an important property if Optimistic Evaluation is to be used as a “plug-in” replacement for Lazy Evaluation. (Chapter 6).
- We describe a real, practical implementation of Optimistic Evaluation that is able to correctly execute arbitrary Haskell programs. (Part III)
- We demonstrate that Optimistic Evaluation allows the use of debugging techniques that would not otherwise be practical. (Chapter 11)

- We demonstrate significant performance increases (approx 20%) compared to our baseline compiler. Given that our compiler has been tuned for many years and was itself the fastest compiler available for the Haskell language, this is a considerable achievement. (Chapter 12)

- Optimistic Evaluation opens up a considerable design space. We have explored part of this space, and have compared several different approaches.

Part I

Overview

CHAPTER 2

Optimistic Evaluation

Optimistic Evaluation consists of several components that come together to produce a functioning system:

- Each **let** expression can evaluate either eagerly or lazily (Section 2.1). Which of these it does depends on the state of a run-time adjustable *switch*, one for each **let** expression. The set of all such switches is known as the *speculation configuration*.

- If the evaluator decides that a speculation has gone on for too long then it will *abort* the speculation. The evaluator will resume by evaluating the body of the **let** that spawned the aborted speculation. (Section 2.2).

- Recursively generated structures such as infinite lists can be generated in *chunks* of several elements, thus reducing the cost of laziness, while avoiding evaluating too much of the structure in one go. (Section 2.3).

- *Online Profiling* learns which **let**s are expensive to evaluate, or are rarely used (Chapter 3). The profiler modifies the speculation configuration so that expensive and rarely used **let** definitions are not speculated.

A more formal treatment of these concepts is provided in Chapter 4, while more implementation details are provided in Chapter 8.

2.1 Switchable Let Expressions

Our compiler reduces the full Haskell language to a restricted subset in which all complex expressions must be bound by $\mathbf{let}$ expressions (Section 4.1). Each $\mathbf{let}$ expression takes the following form:

$$\mathbf{let}\ x = E\ \mathbf{in}\ E'$$

Such a $\mathbf{let}$ can be evaluated either eagerly or lazily:

- **Eager:** Evaluate E immediately. Bind x to the value that E evaluates to.

- **Lazy:** Do not evaluate E immediately. Bind x to a *thunk* in the heap, containing all the information needed to evaluate E later.

Our implementation decides at runtime whether to evaluate a given $\mathbf{let}$ eagerly or lazily.[1] The code generator translates the $\mathbf{let}$ expression above into code that behaves logically like the following:

$\mathbf{if}(switch_{237} \neq 0)\{$
  $x :=$ result of evaluating E
$\}\mathbf{else}\{$
  $x :=$ thunk to compute E when demanded
$\}$
evaluate E'

Our current implementation associates one *static switch* (in this case switch_{237}) with each $\mathbf{let}$. There are many other possible choices. For example, for a function like `map`, which is used in many different contexts, it might be desirable for the switch to take the calling context into account. We have not explored these context-dependent possibilities because the complexity costs of dynamic switches seem to overwhelm the uncertain benefits. In any case, GHC's aggressive inlining tends to reduce this particular problem. We discuss such switches further in Section 4.2.2. If E was large, then this compilation scheme could result in code bloat because two blocks of code are generated for E. We explain ways to avoid this problem in Section 8.1.6.

If a $\mathbf{let}$ is evaluated eagerly, then we say that the $\mathbf{let}$ is *speculated*, we refer to the time spent evaluating the right hand side of the $\mathbf{let}$ as a *speculation*, we say that the speculation was *spawned* by the $\mathbf{let}$, and we say that the $\mathbf{let}$ is the *source* of the speculation.

[1] If strictness analysis is applied beforehand, then some $\mathbf{let}$ expressions may be hard-coded as eager.

2.2 Abortion

Optimistic Evaluation aims to improve performance by evaluating expressions eagerly even when it does not know for sure that they will be cheap or needed.[2] An obvious consequence of this strategy is that it will sometimes evaluate expressions that are expensive and unneeded. It is essential that the evaluator has a way to recover from such mistakes in order to avoid non-termination; this is the role of *abortion.*

If the evaluator detects that a speculation has been going on for a long time, then it aborts all active speculations (they can of course be nested), resuming after the **let** that started the outermost speculation.[3]

Detecting when a speculation has been running for too long can be done in several ways; the choice is not important, so long as it imposes minimal overheads on normal execution. One approach is to have periodic sample points which look at the state of the running program. If a speculation remains active for two consecutive sample points then the evaluator considers the speculation to have gone on for too long. Our real implementation is more complicated, as discussed in Section 10.1.

There are also several ways in which a speculation can be aborted. Our current scheme is similar to the suspension system used for handling asynchronous exceptions in Haskell [MPMR01]. A suspension is created in the heap containing the state of the aborted speculation. If the result of this speculation is found to be needed, then the speculation will be unfrozen and resumed from the point where it left off. The implementation of abortion is described in detail in Section 10.2. Abortion turns out to be a fairly rare event, so it does not need to be particularly efficient (see the statistics in Section 12.4.3).

Abortion alone is enough to guarantee correctness, i.e. that the program will deliver the same results as its lazy counterpart. Indeed, previous work (discussed in Section 13.2) has evaluated a non-strict language using only eager evaluation and abortion. However *abortion alone is not sufficient to guarantee reasonable performance*—for that we need online profiling, which we describe in Chapter 3.

[2] We formalise the concepts of *cheap* and *needed* in Chapter 6.

[3] It is not always necessary to abort all active speculations. See Section 6.3.1.

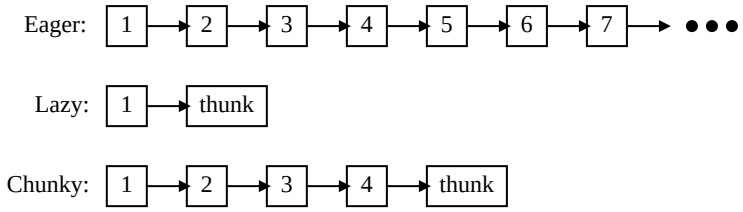


Figure 2.1: Results of evaluating `rest` eagerly, lazily, or chunkily

2.3 Chunky Evaluation

Programs in non-strict languages ofter work with infinite data structures [Hug89]. Chunky Evaluation is a feature of Optimistic Evaluation that allows such structures to be evaluated in *chunks*, giving better performance than if they were evaluated entirely eagerly or entirely lazily. Consider the following Haskell program, which generates an infinite stream of integers.

$$ints\ n = n : ints\ (n + 1)$$

In the core language used by our compiler, function arguments can only be variables. This example will thus be desugared to the following:

$$\begin{aligned} &ints\ n = \\ &\quad \textbf{let}\ n' = n + 1 \quad \textbf{in} \\ &\quad \textbf{let}\ rest = ints\ n' \quad \textbf{in} \\ &\quad (n : rest) \end{aligned}$$

The function $ints$ generates an infinite list of integers, starting from n. Let us assume that the first million elements of this list are needed. How should $rest$ be evaluated? Figure 2.1 illustrates several alternatives. If the evaluator always evaluates $rest$ eagerly then it will call $ints$ recursively forever and so will not terminate. However always evaluating $rest$ lazily is not very good either. In this case, the evaluator will have to bear the overhead of creating one million thunks, even though all but one of them describe evaluations that are needed.

A better alternative is what we call *chunky evaluation*. Under chunky evaluation $rest$ is evaluated eagerly *up to some limit* and then evaluated lazily. If $rest$ is set to evaluate chunkily then $ints$ will produce its result list in chunks. Consider the case where the list is generated in chunks of 4 elements. Chunky Evaluation of $ints$ will only recurse 4 levels deep and so will terminate; however 4 times fewer thunks will be created.

The size of the chunks used by chunky evaluation is set by the profiler and can adapt according to dynamic demand. The profiler aims to set the chunk size such that performance is maximised. The chunk size will thus increase if more elements are used and decrease if fewer elements are used.

This chunky behaviour can be useful even for finite lists that are entirely needed. Non-strict programmers often use a generate-and-filter paradigm, relying on laziness to avoid creating a very large intermediate list. Even if the compiler knew that the intermediate list would be completely evaluated, it would sometimes be a bad plan to evaluate it eagerly as the intermediate data structure might be too big to fit in the cache or even to big to fit in main memory. In such cases, Chunky Evaluation will give much better performance.

Our compiler implements chunky evaluation by limiting the depth to which speculation may be nested. Consider what happens in the program above if speculation of $rest$ is limited to a depth of 4. In this case, outermost evaluation of $rest$ will be speculative and will itself start a second, nested, speculation of $rest$. This second speculation of $rest$ will start a third speculation, which will in turn start a fourth speculation. However, because the speculation depth limit for $rest$ is 4, there can be no fifth nested speculation and so the next attempt to evaluation $rest$ will be lazy. The four nested speculations will then return, each building a cons cell. The result is the chunky structure shown in Figure 2.1.

The code for a **let** now behaves semantically like the following (replacing the code given in Section 2.1):

$$
\begin{array}{l}
\mathbf{if}(\mathbf{SpecDepth} < \mathrm{limit}_{237})\{ \\
\quad \mathbf{SpecDepth} := \mathbf{SpecDepth} + 1 \\
\quad x := \text{value of } E \\
\quad \mathbf{SpecDepth} := \mathbf{SpecDepth} - 1 \\
\}\mathbf{else}\{ \\
\quad x := \text{thunk for } E \\
\} \\
\text{evaluate } E'
\end{array}
$$

where $\mathbf{SpecDepth}$ is the number of nested speculations that we are currently inside and $limit_{237}$ is an integer *depth limit* that determines how deeply this particular **let** can be speculated.

2.4 Probability of a Nested Speculation being Used

We can justify depth limited speculation by appealing to an intuitive notion of probability. We can label every **let** in a program with the probability that its right hand side will be needed, given that its body is needed. If speculative evaluations are nested, then we multiply their probabilities together. Consider the following program:

$$
\begin{array}{rcl}
f\ x & = & \mathbf{let}\ y = g\ x\ \mathbf{in}\ E \\
g\ x & = & \mathbf{let}\ z = h\ x\ \mathbf{in}\ E'
\end{array}
$$

Imagine that the probabilities for y and z are $\frac{1}{2}$ and $\frac{3}{4}$ respectively. What then is the probability of the right hand side of z being needed, given that E is needed? This is the probability that y is needed, given that E is needed, and that z is needed, given that E' is needed. If we assume that these two probabilities are independent, then we can multiply them together, giving $\frac{3}{8}$.

In practice, things are not so simple. The probabilities for y and z are unlikely to be independent and so the compound probability is unlikely to be exactly equal to the product of the two; however the intuitive principle of more deeply nested speculations being less likely to be needed than less deeply nested ones does seem to hold. A similar notion of probability is used to justify task priorities in several parallel systems (Section 13.3).

2.5 Exceptions and Errors

It is important that Optimistic Evaluation deals correctly with exceptions and errors. Consider the following function:

$$
\begin{array}{l}
f\ x = \\
\quad \textbf{let} \\
\qquad y = \mathit{error}\ \text{“}\texttt{urk}\text{”} \\
\quad \textbf{in} \\
\qquad \textbf{if}\ x\ \textbf{then}\ y\ \textbf{else}\ 12
\end{array}
$$

In Haskell, the $error$ function prints an error message and halts the program. Optimistic Evaluation may evaluate y without knowing whether y is actually needed. It is obviously unacceptable to print `"urk"` and halt the program because Lazy Evaluation would not do that if x is $False$. The same issue applies to exceptions of all kinds, including divide-by-zero and black-hole detection [MLP99].

In GHC, $error$ raises a catchable exception, rather than halting the program [PRH+99]. The exception-dispatch mechanism tears frames off the stack in the conventional way. The only change needed is to modify this existing dispatch mechanism to recognise a speculative-return frame, and return to it with a thunk that will re-raise the exception. A **let** thus behaves rather like a **catch** statement, preventing exceptions raised by speculative evaluation of its right hand side from escaping. This concept is formalised in Section 4.2.

2.6 Unsafe Input/Output

Optimistic Evaluation is only safe because Haskell is a *pure* language: evaluation has no side effects, and thus evaluation order does not affect the result of a program. Input/Output is safely partitioned using the `IO` monad [Wad95, PW93, Wad97], so there is no danger of speculative computations performing I/O. However, Haskell programs sometimes make use of impure I/O,

using the "function" $unsafePerformIO$. This "function" has the following type signature:

$$unsafePerformIO : IO\ \alpha \rightarrow \alpha$$

We initially believed that speculation of $unsafePerformIO$ was safe. Our argument was that, by using the $unsafePerformIO$ function, one was asserting that the IO behaved like a pure function and could be evaluated at any time without affecting the semantics of the program. However, while speculating $unsafePerformIO$ is indeed safe, aborting it is not. Consider the following example:[4]

$$\begin{array}{l}
\textbf{let}\ x = unsafePerformIO\ (\textbf{do} \\
\qquad\qquad acquire_lock(lock) \\
\qquad\qquad y \leftarrow \text{use locked object} \\
\qquad\qquad release_lock(lock) \\
\qquad\qquad return\ y \\
\qquad)\\
\textbf{in} \\
\qquad \ldots
\end{array}$$

This example binds x to the result of an imperative procedure in the IO Monad. The imperative procedure acquires a lock, does some work, and then releases the lock. It is not safe to abort the procedure while it is holding the lock because this would put the system into a state in which the lock was held by a suspended computation. There is no guarantee that a suspended computation will ever be resumed and thus the lock may never be released—causing a deadlock.

A similar problem is that code which expects to be executed atomically can find itself being interleaved with other IO. Consider the following example:

$$\begin{array}{l}
\textbf{let}\ x = unsafePerformIO\ (\textbf{do} \\
\qquad\qquad writeIORef\ ref\ 1 \\
\qquad\qquad v \leftarrow readIORef\ ref \\
\qquad\qquad return\ v \\
\qquad)\\
\textbf{in} \\
\qquad \ldots
\end{array}$$

In this example, the imperative procedure writes the value 1 to an imperative reference and then reads this value back again. If the program has only a single thread of execution, then we would expect x to always have the value 1. However, if $unsafePerformIO$ is speculated, then it

[4] This example uses *do-notation* [Pey01]. The indented lines after the **do** keyword are imperative commands in the IO Monad.

may be suspended between the call to $writeIORef$ and the call to $readIORef$. By the time the procedure is is resumed, another procedure may have written a different value to ref, causing a value other than 1 to be returned.

It is for these reasons that Optimistic Evaluation does not allow uses of $unsafePerformIO$ inside speculative evaluations. Any speculation which attempts to call $unsafePerformIO$ will abort immediately.

CHAPTER 3

Online Profiling

Optimistic Evaluation saves pennies by risking pounds. Each speculation saves the relatively small cost of creating a thunk, but risks the potentially huge cost of evaluating an unneeded expression. The purpose of the online profiler is to ensure that the pennies saved outweigh the (hopefully few) pounds wasted.

In this chapter we give a rough overview of how online profiling works. A more detailed theoretical analysis is given in Chapter 6 while more details on practical implementation are given in Chapter 9.

3.1 The Need for Profiling

While abortion ensures that all speculations must terminate, it does not ensure that evaluation is efficient. It turns out that with the mechanisms described so far, some programs run faster, but some run dramatically slower. For example the `constraints` program from the NoFib benchmark suite [Par92] runs over 150 times slower (Section 12.3.1). Detailed investigation shows that these programs build many moderately-expensive thunks that are seldom used. These thunks are too cheap to trigger abortion, but nevertheless aggregate to waste massive amounts of time (and space).

One obvious solution is this: trigger abortion very quickly after starting a speculative evaluation, thereby limiting the size of a speculation and thus limiting wasted work. Unfortunately, this would ignore the large proportion (See Section 12.4.5) of speculations that are moderately expensive, but whose values are almost always needed. It would also restrict chunky evaluation because a chunk can be fairly expensive.

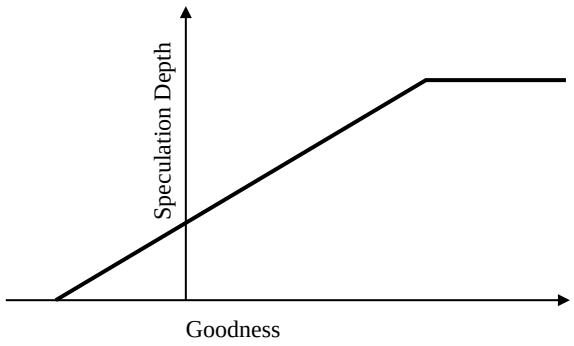


Figure 3.1: The speculation depth for a **let** depends on its estimated goodness

If the evaluator is to exploit such opportunities, it needs to have an accurate estimate of the amount of work that is actually being wasted by speculation of a **let**, taking into account not only the costs of speculations, but also which speculations are actually needed. This is the role of our online profiler.

3.2 Goodness

For each **let** in the program, our profiler maintains an estimate of its *goodness*. Goodness is defined as:

$$goodness \quad = \quad savedWork - wastedWork$$

where $savedWork$ is the amount of work that has been saved so far by speculating the **let** and $wastedWork$ is the amount of work wasted so far by speculating the **let**. Going back to our previous analogy, $savedWork$ is the pennies that have been saved and $wastedWork$ is the pounds that have been wasted.

In practice it is not possible to know at runtime exactly how much work has been wasted by speculation of a **let** as the profiler cannot know exactly which of the expressions evaluated so far will turn out to be needed. It is however possible for the profiler to cheaply compute an overestimate of $wastedWork$, and thus a safe underestimate of $goodness$.

We start by describing an idealised implementation in which all **let** expressions are profiled all of the time and in which a counter is maintained for every **let**, holding the current estimate of its goodness.

3.2.1 Using Goodness to Guide Speculation

If the evaluator has an estimate of the goodness of a **let**, then it can use this to guide its speculation of that **let**. Figure 3.1 gives a possible function from goodness to speculation depth limit (Section 2.3). As the estimated goodness of a **let** increases, we increase the depth limit and so

speculate it more. Similarly, as the estimated goodness of a **let** decreases we decrease the depth limit and so speculate it less.

Once the goodness of a **let** falls below a defined cutoff point, the **let** will become entirely lazy and will waste no more work. This allows us to place a bound on the amount of work that a **let** can waste before it becomes entirely lazy, as we explain in Section 6.3.3.

3.2.2 Calculating Saved Work

Every time a **let** is speculated, the evaluator saves the work required to build and manage a thunk. The evaluator keeps track of the work saved in this way by maintaining a $savedWork$ counter for each **let** and incrementing this counter every time the **let** is speculated.[1]

By speculating a **let**, the evaluator may also change the amount of work that the garbage collector has to do. We discuss this in Section 9.3.

3.3 Calculating Wasted Work

Speculation of a **let** will waste work if the right hand side of the **let** would not have been needed by Lazy Evaluation. Our calculation of wasted work relies on the concepts of *ventures*, *local work*, and *blame*, which we describe in the following subsections.

3.3.1 Ventures

We use the term *venture* to refer to the region of time during which work was done to produce a value for a particular heap binding.[2] To understand this concept, it may help to consider the following example:

$$
\begin{array}{l}
\textbf{let } x = \\
\qquad \textbf{let } y = E_y \textbf{ in} \\
\qquad \textbf{let } z = E_z \textbf{ in} \\
\qquad E_x \\
\textbf{in} \\
\qquad \ldots
\end{array}
$$

If x, y and z are all speculated, then the evaluation of this expression will contain the ventures illustrated in Figure 3.2. The ventures for y and z take place during the venture for x. When a venture starts, it will place a return frame on the stack; this frame will remain on the stack until the venture completes. If ventures are nested then we consider all ventures with a return frame

[1] Of course, if the **let** is itself evaluated inside another speculation then that **let** might not otherwise have been speculated. This issue is addressed in Section 3.3.5.

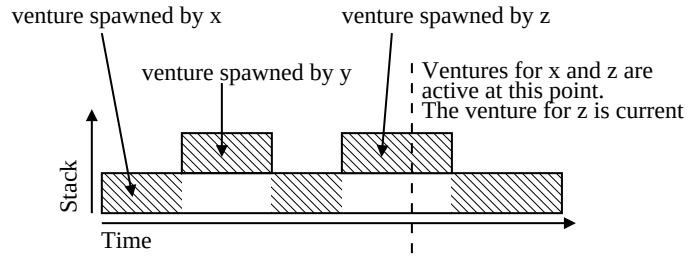


Figure 3.2: A venture represents the time during which work is done to produce a value for a particular binding

on the stack to be *active* and consider the innermost venture to be *current*. In Figure 3.2 the return frame for the current venture is shaded.

Every venture is either the unique *root venture* which computes the return value for the program, a *speculation* which is speculatively evaluating the right hand side of a $\mathbf{let}$, or a *thunk venture* which is evaluating a thunk whose value was demanded by another venture. In this thesis, we will often write "x", to mean "a venture that is producing a value for the right hand side of the $\mathbf{let}$ identified by the binder x".

3.3.2 Work

We use the term *work* to refer to any reasonable measure of execution cost, such as time or heap allocation.[3] We discuss a formal model of work in Chapter 5 and practical measurement of work in Chapter 9.

3.3.3 Local Work

The *local work* done in a venture is a measure of the work done by the venture, excluding any work done in enclosed ventures. All work done in the lifetime of a program is included in the local work of exactly one venture. In Figure 3.2 the local work of a venture is the area of that venture that is shaded.

3.3.4 Blame

The *blame* for a venture is the amount of work that the profiler has chosen to assign to that venture. Like local work, all work done in the lifetime of a program is included in the blame of exactly one venture. Unlike local work, blame may be moved between ventures as the program runs. If work is known to be needed by Lazy Evaluation, then it will be blamed on the root

[2] By "produce a value", we mean "to evaluate it to weak head normal form".

[3] By "heap allocation" we mean the total number of bytes of memory allocated during the lifetime of the program. Haskell, like most purely functional languages, has to allocate a new heap cell for every value produced and so allocates heap almost continually.

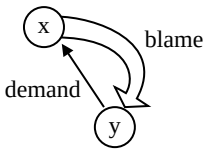


Figure 3.3: When y demands x, x's blame is passed to y

venture, otherwise it will be blamed on the first speculation whose execution required that the work be done. The $wastedWork$ counter that we maintain for each $\mathbf{let}$ is the sum of the blames for all ventures that were spawned from that $\mathbf{let}$.

Blame behaves like a linear object [Wad90b, Bak95, TWM95]. It can be passed between ventures, but cannot be duplicated or deleted. If the result of a venture is needed by another venture then the blame of the demanded venture will be transferred to the demanding venture. The justification for this is that the work done in the demanded venture was necessary in order to produce the result for the demanding venture. If the result of a venture has not been used by any other venture, then its blame will be the sum of the local work for that venture and any other work that was blamed on that venture while it executed.

To illustrate how blame works, consider the following program (illustrated by Figure 3.3):

$$\begin{array}{ll} \mathbf{let}\ x = E & \mathbf{in} \\ \mathbf{let}\ y = x + 1 & \mathbf{in} \\ 4 & \end{array}$$

If x and y are both speculated then the local work done to evaluate x is the work done to evaluate E, while the local work done to evaluate y is simply the work required to perform an addition. When speculation of x completes, the profiler will initially assume that it was not needed, and blame x for the evaluation of E. When y is speculated, it will demand the value of x, causing the profiler to transfer all blame from x into y. When the program completes, the blame for x will be zero, and the blame for y will be the work required to evaluate E plus the work required to do an addition.

Our blame allocation system can attribute more blame to a venture than is "fair". Consider the following example (illustrated by Figure 3.4):

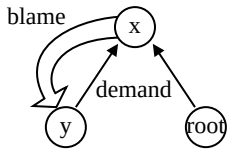


Figure 3.4: x's blame is passed to y rather than to the root computation and so is incorrectly regarded as being wasted

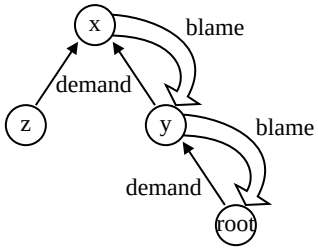


Figure 3.5: x's blame will be passed to either y or z depending on evaluation order.

$$\begin{array}{ll} \textbf{let } x = E & \textbf{in} \\ \textbf{let } y = x + 1 & \textbf{in} \\ x & \end{array}$$

The local work for x is blamed on y and so is considered to be wasted. But in fact, x is needed by the root venture, so the "real" work wasted by speculating y is tiny (just incrementing a value). Fortunately it is safe to overestimate the amount of work wasted, so we simply accept this approximation.

Similarly, the blame assigned to a venture will depend on the order of evaluation. Consider the following example (illustrated in Figure 3.5):

$$\begin{array}{ll} \textbf{let } x = E & \textbf{in} \\ \textbf{let } y = x + 1 & \textbf{in} \\ \textbf{let } z = x + 1 & \textbf{in} \\ y & \end{array}$$

If y and z are both speculated then y will demand x before z does and so y will be blamed for x's work rather than z. This seems unfair as y and z have the same definition. However, this does not affect the safety of our profiler, and so again we simply accept it.

For a more rigorous definition of work and blame, refer to Chapter 6. For more details of the implementation of this technique, refer to Chapter 9.

3.3.5 The Cost of a Let

It is important that the cost of evaluating an enclosed $\mathbf{let}$ expression should include the cost of building a thunk for that $\mathbf{let}$, even if that $\mathbf{let}$ was actually speculated. Consider for example:

$$\begin{array}{l}\mathbf{let}\ x = \\ \quad\quad \mathbf{let}\ y = 1 + 2\ \mathbf{in} \\ \quad\quad 3 \\ \mathbf{in} \\ \quad\quad \ldots\end{array}$$

If y is speculated then the work required to build a thunk has been saved; however it is speculation of y that has saved the work and not speculation of x. When y is speculated it will increment its $savedWork$ counter to record the fact that the work was saved. If the cost of evaluating x did not include the thunk cost then this saving would be counted twice, causing the profiler to overestimate goodness.

To illustrate what can go wrong if this principle is not followed, consider the following example:

$$\begin{array}{l}f\ x = \\ \quad\quad \mathbf{let}\ y = f\ x\ \mathbf{in}\ 3\end{array}$$

Evaluating y speculatively is clearly bad. It will lead to an expensive speculation which is not needed. Any safe profiling strategy should thus decrease y's goodness whenever it is speculated.

Consider however what would happen if a naive profiler assumed that a call to f did not incur the cost of building a thunk for y. In this case, the wasted work for y would only be the work required to call f and then immediately return 3. However this work is likely to be less than the work that will be added to the $savedWork$ counter when y is speculated. As a result, y will appear to be saving more work than it was wasting, and so its goodness would increase.

By contrast, a correct profiler would ensure that the work done by f included the cost of building a thunk for y, even if y was speculated. This work would cancel out the work recorded as saved and so ensure that y had negative goodness.

3.4 Burst Profiling

It would be inefficient to profile every speculation that took place, so we instead profile a random selection of speculations. The runtime of a program is divided up into a series of *periods*. Each period starts at a *boundary point* and lasts until the next boundary point. Any speculations that start during a period are considered to belong to that period. This is illustrated in Figure 3.6.

A random selection of periods is profiled. When a period is profiled, all speculations that

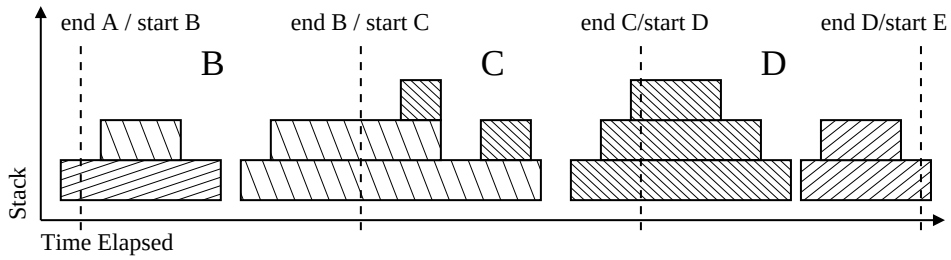


Figure 3.6: Every computation belongs to exactly one profi le period

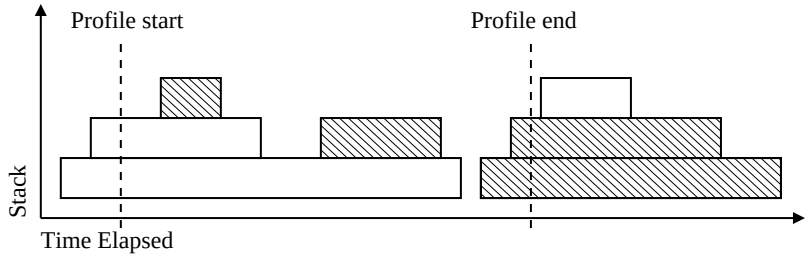


Figure 3.7: Only those computations that start during a profi led period will be profi led.

belong to that period will be profiled (Figure 3.7). If every period has a probability p of being profiled, then it follows that every speculation also has a probability p of being profiled.

The burst profiler assumes that the program behaves in the same way when being profiled as it does when it is not being profiled. It is thus able to build up a reasonable estimate of the goodness of any **let**. Over time, it becomes increasingly unlikely that a **let** can be wasting work while having a positive estimated goodness. In the limit, as runtime tends to infinity, the estimate of relative goodness produced by burst profiling should be the same as the estimate that would be produced by a continuous profiler. This convergence is accelerated if the profiler starts off by profiling everything, and gradually backs off as it gains confidence (Section 9.4.3).

While burst profiling does not make it impossible for Optimistic Evaluation to perform very badly relative to Lazy Evaluation, it does make it vanishingly unlikely that it will do so.

Part II

Theory

CHAPTER 4

Semantics

In this chapter we present a formal model of Optimistic Evaluation. This model provides an insight into the way that Optimistic Evaluation works, and serves as the basis for the more detailed models of Chapters 5 and 6.

- We start, in Section 4.1, by describing a simple non-strict language. This is the language that we work with for the rest of the thesis.

- In Section 4.2, we give a small-step *operational semantics* for our language. This semantics describes not only evaluation, but also abortion. We see in Part III that this semantics corresponds very closely to our real implementation of Optimistic Evaluation.

- In Section 4.3, we give a denotational semantics for our language. This semantics provides a high-level model of evaluation that is independent of evaluation strategy.

- Finally, in Section 4.4, we prove that the operational semantics of Section 4.2 is sound with respect to the denotational semantics of Section 4.3, thus proving that Optimistic Evaluation will always produce the same results as Lazy Evaluation.

Expression	E	::=	x	variable
		|	$C\ x_0 \ldots x_n$	constructor application
		|	$\mathbf{case}\ E\ \mathbf{of}\ P_0 \ldots P_n$	case analysis
		|	$\mathbf{let}\ x = E\ \mathbf{in}\ E'$	thunk creation
		|	$\lambda x.E$	function abstraction
		|	$E\ x$	function application
		|	$\mathbf{exn}$	exception or error
		|	n	integer constant
		|	$x \otimes x'$	primitive operation
Alternatives	P	::=	$C\ x_0 \ldots x_n \rightarrow E$	
ValueExp	V	::=	$C\ \alpha_0 \ldots \alpha_n \mid \lambda x.E \mid \mathbf{exn} \mid n$	

Here C ranges over constructors, x over variable names, and α over heap references (see later).

Figure 4.1: Terms of a simple language

4.1 A Simple Language

The language we work with is Haskell [Pey03]. While the externally-visible Haskell language is very complex, it can be reduced into the simple form given in Figure 4.1, and that is the language our theory deals with. This language is similar to the augmented *Core* [TtGT01] language[1]which GHC uses internally and also to A-normal form [FSDF93]. Key points to note are the following:

- A variable x is a local variable bound by an enclosing $\mathbf{let}$, $\mathbf{case}$ or λ expression.
- Each $\mathbf{let}$ is required to have a unique binder x. This allows us to refer to a $\mathbf{let}$ by its binder.
- Functions and constructors are always applied to variables, rather than to expressions. It follows that $\mathbf{let}$ is the only point at which a thunk might be created (A $\mathbf{case}$ expression scrutinises an arbitrary expression E, but it does not first build a thunk.). This restriction does not limit the expressiveness of the language because a $\mathbf{let}$ expression can be used to give a name to an argument of a function or constructor.
- We represent booleans using the nullary constructors $True$ and $False$.
- $\otimes$ ranges over primitive operations, such as $+, -, /, ==, <$ etc.

[1]Core should not be confused with the Kernel language used in the Haskell report. Unlike the Kernel, the Core requires that function arguments be variables.

- Exceptions and errors (see Section 2.5) are handled exactly as described in [PRH+99]. An exception $\mathbf{exn}$ is considered to be a value, rather than a control operator.

- $\mathbf{let}$ is not recursive. That is E cannot make reference to x. If recursion is desired, then the user can define a fixed point function within the language. For example, Y can be written as:

$$Y \equiv \lambda f.\ \mathbf{let}\ y = (\lambda x.\ \mathbf{let}\ z = x\ x\ \mathbf{in}\ f\ z)\ \mathbf{in}\ y\ y$$

  In the real language used by our compiler, recursive $\mathbf{let}$s are of course allowed; however omitting them simplifies our formal presentation. In particular, it avoids us having to deal with speculations that demand their own values.

4.2 Operational Semantics

We describe program execution using a small step operational semantics. This semantics specifies the sequence of states that the virtual machine will pass through as execution proceeds. The main transition relation, $\longrightarrow$, takes the form:

$$S \longrightarrow_{\Sigma} S'$$

meaning that the virtual machine will transform the state S into the state S' in one step, under speculation configuration Σ. The speculation configuration is updated sporadically by the profiler, rather than by the the evaluation rules.[2]

The semantics we present is similar to the STG Machine [Pey92] and to the semantics for Lazy Evaluation given by Sestoft [Ses97], being lower level than Sestoft, but higher level than the STG Machine. Unlike Sestoft, we evaluate expressions to references to values, rather than to values themselves. This allows us to be precise about the closures that are present in the heap. While this leads to a semantics that is more complex than Sestoft's, it causes our semantics to be a very accurate model of our real implementation; this is important, given that we are interested in modelling performance.

In the subsections that follow, we formalise Optimistic Evaluation in more detail. In Section 4.2.1 we explain the structure of the program state S. In Section 4.2.2 we give the rules that define the evaluation relation '$\longrightarrow$'. In Section 4.2.4 we use the evaluation relation '$\longrightarrow$' to define a semantics for *bounded speculation*. In Section 4.2.3 we give a semantics for abortion. Finally, in Section 4.2.6 we define the restricted language that is used for the rest of Part II.

[2] We give a semantics for this profiler in Section 6.2.5.

State	S	::=	$\Gamma; c; s$	Program state with heap, command, and stack
Command	c	::=	E	evaluate E
		|	$\triangledown\alpha$	return α
		|	$\circledcirc\alpha$	demand the value of α
Heap	Γ	::=	$\{\alpha_i \mapsto K_i\}_0^n$	
Closure	K	::=	$(\!\lvert V \rvert\!)$	value closure, with value V
		|	$\langle\alpha\rangle$	indirection, with indirectee α
		|	E	thunk, with body E
		|	$\alpha \angle l$	suspended stack frame
Stack	s	::=	$[\,]$	empty stack
		|	$f : s$	stack with topmost frame f
Frame:	f	::=	$\{x\}E$	speculative return frame
		|	$\#\alpha$	update frame
		|	l	local frame
LocalFrame	l	::=	$\{P_i\}_0^n$	Case Match
		|	$@\alpha$	Function application
		|	$\otimes\alpha$	Primop, awaiting first argument
		|	$n\otimes$	Primop, awaiting second argument
Config	Σ	::=	$\{x_i \mapsto n_i\}_0^m$	Speculation configuration.

Figure 4.2: Syntactic forms for states

4.2.1 The Program State

The structure of program states is given in Figure 4.2 and is commented on below:

- Γ represents the *heap*. The heap is a function mapping *heap references* to *closures*. If Γ maps a heap reference α to a closure K, then K describes how one can obtain a value for α. A closure is either a *value closure*, a *thunk*, an *indirection*, or a *suspended stack frame*.

  - The value of a value closure $(\!\lvert V \rvert\!)$ is V.
  - The value of an indirection $\langle\alpha\rangle$ is the value of the closure referenced by α.
  - The value of a thunk E is the value one obtains by evaluating E.
  - The value of a suspended stack frame is discussed in Section 4.2.3.

  Runtime expressions may contain heap references in place of variables. We write $E[\alpha/x]$ to denote the expression formed by replacing all instances of the variable x with the heap reference α.

- c represents a *command*. The command says what the program is currently doing. If the command is E then the program is evaluating the expression E. If the command is $\triangledown\alpha$

then the program is returning α, which is a reference to a value. If the command is $\circledcirc\alpha$ then the program is demanding the value of α.

- s is a *stack* of *stack frames* f (c.f. [Gus98]), each of which represents work still to be done.

  - A *speculative return frame* contains work that should be done once a speculation has been completed or aborted. If the virtual machine returns to a speculative return frame, $\{x\}E$, it will bind x to the result of the speculation and then evaluate E.

    We define the *speculation depth* of a state to be the number of speculations that are active, or equivalently, the number of speculative return frames on the stack.

  - An *update frame* represents a thunk that needs to be updated with its value. If the virtual machine returns to an update frame, $\#x$, it will replace the thunk with an indirection to its value.

  - A *case return frame* contains a set of $\mathbf{case}$ alternatives. If the virtual machine returns to a $\mathbf{case}$ return frame, $\{P_i\}_0^n$, it will select the appropriate alternative, P_j, and evaluate it.

  - A *function application frame* contains an argument to pass to the returned function. If the returned value is not a function then the program will fail.

  - A *primop frame* gives the state of a partially applied primitive operation. If the stack frame is '$\otimes\alpha$' then the primop is waiting for its first argument. If the stack frame is '$n\otimes$' then the first argument evaluated to n and the primop is waiting for its second argument.

  A subset of stack frames are *local frames*. Local frames describe work to be done within the current venture (see Section 3.3.1) while other stack frames represent venture boundaries. We develop these ideas further in Chapters 5 and 6. While the letter f can refer to any kind of stack frame, the letter l can only refer to a local frame.

- Σ represents the *speculation configuration*. Σ maps a $\mathbf{let}$ (identified by its binder) to its *depth limit*: the maximum speculation depth at which it can be speculated. If the $\mathbf{let}$ is always lazy then the depth limit will be 0.

4.2.2 Evaluation Transitions

The transition rules for $\longrightarrow$ are given in Figure 4.3. The first ten rules describe conventional Lazy Evaluation (cf. [Ses97]) while the last two describe speculation:

Evaluate a value constant:

(VAL) $\Gamma; V; s \longrightarrow_\Sigma \Gamma[\alpha \mapsto (\!|V|\!)]; \triangledown\alpha; s$ where α is fresh

Demand the value of a closure:

(VAR) $\Gamma; \alpha; s \longrightarrow_\Sigma \Gamma; \circledcirc\alpha; s$

(DEM1) $\Gamma[\alpha \mapsto (\!|V|\!)]; \circledcirc\alpha; s \longrightarrow_\Sigma \Gamma[\alpha \mapsto (\!|V|\!)]; \triangledown\alpha; s$

(DEM2) $\Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha; s \longrightarrow_\Sigma \Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha'; s$

(DEM3) $\Gamma[\alpha \mapsto E]; \circledcirc\alpha; s \longrightarrow_\Sigma \Gamma; E; (\#\alpha : s)$

(RESUME) $\Gamma[\alpha \mapsto \alpha' \angle l]; \circledcirc\alpha; s \longrightarrow_\Sigma \Gamma; \circledcirc\alpha'; (l : \#\alpha : s)$

(UPD) $\Gamma; \triangledown\alpha; (\#\alpha' : s) \longrightarrow_\Sigma \Gamma[\alpha' \mapsto \langle\alpha\rangle]; \triangledown\alpha; s$

Function Application:

(APP1) $\Gamma; E\ \alpha; s \longrightarrow_\Sigma \Gamma; E; (@\alpha : s)$

(APP2) $\Gamma[\alpha \mapsto (\!|\lambda x.E|\!)]; \triangledown\alpha; (@\alpha' : s) \longrightarrow_\Sigma \Gamma[\alpha \mapsto (\!|\lambda x.E|\!)]; E[\alpha'/x]; s$

Case Expression:

(CASE1) $\Gamma; \mathbf{case}\ E\ \mathbf{of}\ \{P_i\}_0^n; s \longrightarrow_\Sigma \Gamma; E; (\{P_i\}_0^n : s)$

(CASE2) $\Gamma[\alpha \mapsto (\!|C\ \{\alpha_i\}_0^n|\!)]; \triangledown\alpha; (\{P_i\}_0^n : s) \longrightarrow_\Sigma \Gamma[\alpha \mapsto (\!|C\ \{\alpha_i\}_0^n|\!)]; E[\{\alpha_i/x_i\}_0^n]; s$ where $P_k = C\ \{x_i\}_0^n \to E$

Primitive Operation:

(OP1) $\Gamma; \alpha \otimes \alpha'; s \longrightarrow_\Sigma \Gamma; \circledcirc\alpha; (\otimes\alpha' : s)$

(OP2) $\Gamma[\alpha \mapsto (\!|n|\!)]; \triangledown\alpha; (\otimes\alpha' : s) \longrightarrow_\Sigma \Gamma[\alpha \mapsto (\!|n|\!)]; \circledcirc\alpha'; (n\otimes : s)$

(OP3) $\Gamma[\alpha' \mapsto (\!|n'|\!)]; \triangledown\alpha'; (n\otimes : s) \longrightarrow_\Sigma \Gamma[\alpha' \mapsto (\!|n'|\!)]; n \tilde{\otimes} n'; s$

Exception:

(EXN1) $\Gamma[\alpha \mapsto (\!|\mathbf{exn}|\!)]; \triangledown\alpha; (@\alpha' : s) \longrightarrow_\Sigma \Gamma[\alpha \mapsto (\!|\mathbf{exn}|\!)]; \triangledown\alpha; s$

(EXN2) $\Gamma[\alpha \mapsto (\!|\mathbf{exn}|\!)]; \triangledown\alpha; (\{P_i\}_0^n : s) \longrightarrow_\Sigma \Gamma[\alpha \mapsto (\!|\mathbf{exn}|\!)]; \triangledown\alpha; s$

(EXN3) $\Gamma[\alpha \mapsto (\!|\mathbf{exn}|\!)]; \triangledown\alpha; (n\otimes : s) \longrightarrow_\Sigma \Gamma[\alpha \mapsto (\!|\mathbf{exn}|\!)]; \triangledown\alpha; s$

(EXN4) $\Gamma[\alpha \mapsto (\!|\mathbf{exn}|\!)]; \triangledown\alpha; (\otimes\alpha' : s) \longrightarrow_\Sigma \Gamma[\alpha \mapsto (\!|\mathbf{exn}|\!)]; \triangledown\alpha; s$

Lazy Evaluation of a `let`:

(LAZY) $\Gamma; (\mathbf{let}\ x = E\ \mathbf{in}\ E'); s \longrightarrow_\Sigma \Gamma[\alpha \mapsto E]; E'[\alpha/x]; s$ if $\Sigma(x) \leq \mathit{specDepth}(s)$ and α is fresh

Speculative Evaluation of a `let`:

(SPEC1) $\Gamma; (\mathbf{let}\ x = E\ \mathbf{in}\ E'); s \longrightarrow_\Sigma \Gamma; E; (\{x\}E' : s)$ if $\Sigma(x) > \mathit{specDepth}(s)$

(SPEC2) $\Gamma; \triangledown\alpha; (\{x\}E : s) \longrightarrow_\Sigma \Gamma; E[\alpha/x]; s$

Figure 4.3: Operational semantics: evaluation

- Rule *(VAL)* evaluates a value V by creating a value closure $(\!|V|\!)$ in the heap and returning a reference to it.
- Rule *(VAR)* evaluates a heap reference α by demanding the value of the closure it references.
- Rule *(DEM1)* demands the value of a value closure. A value closure is already fully evaluated, so *(DEM1)* returns immediately.
- Rule *(DEM2)* demands the value of an indirection by demanding the value of the closure that the indirection points to.
- Rule *(DEM3)* demands the value of a thunk by evaluating its body. The thunk is removed from the heap, and an update frame is pushed. When evaluation of the thunk body completes, the update frame will trigger rule *(UPD)* which will replace the thunk with an indirection to its value, allowing the result of the evaluation to be shared.
- Rules *(DEM1)* and *(DEM2)* could be subsumed into rule *(DEM3)* by considering the value closure $(\!|V|\!)$ to be the thunk V and the indirection $\langle\alpha\rangle$ to be the thunk α. However, such a virtual machine would create unnecessary closures and push unnecessary update frames. The inclusion of rules *(DEM1)* and *(DEM2)* also mirrors the behaviour of our real implementation.
- Rule *(RESUME)* resumes a suspended evaluation. This rule is discussed in Section 4.2.3.
- Rule *(APP1)* evaluates a function application $E\ \alpha$ by evaluating the function E and placing a function application frame on the stack. When evaluation of the function completes, the function application frame will trigger an application of rule *(APP2)*, causing the returned function to be applied to α.
- Rule *(CASE1)* evaluates a **case** expression by evaluating the scrutinee and pushing the **case** alternatives on the stack. When evaluation of the scrutinee completes, rule *(CASE2)* will evaluate the appropriate **case** alternative.
- Rule *(OP1)* evaluates a primitive operation by pushing a primop frame and evaluating the first argument. When the first argument is evaluated, rule *(OP2)* will evaluate the second argument. When both arguments are evaluated, rule *(OP3)* will perform the primitive operation (denoted by $\tilde{\otimes}$).
- If an exception is raised, then rules *(EXN1)*, *(EXN2)*, *(EXN3)*, and *(EXN4)* unwind the stack until a speculative return frame is reached.[3]

[3]This language has no catch expressions and thus no catch frames. This corresponds to the behaviour of

$$
\begin{array}{lrcl}
\textit{(!EXP)} & \Gamma; E; s & \rightsquigarrow & \Gamma[\alpha \mapsto E]; \circledcirc\alpha; s \\
\textit{(!RET)} & \Gamma; \triangledown\alpha; s & \rightsquigarrow & \Gamma; \circledcirc\alpha; s \\
\textit{(!SPEC)} & \Gamma; \circledcirc\alpha; (\{x\}E : s) & \rightsquigarrow & \Gamma; E[\alpha/x]; s \\
\textit{(!UPD)} & \Gamma; \circledcirc\alpha; (\#\alpha' : s) & \rightsquigarrow & \Gamma[\alpha' \mapsto \langle\alpha\rangle]; \circledcirc\alpha; s \\
\textit{(!ABORT)} & \Gamma; \circledcirc\alpha; (l : s) & \rightsquigarrow & \Gamma[\alpha' \mapsto \alpha\angle l]; \circledcirc\alpha'; s \\
 & & & \text{where } \alpha' \text{ is fresh}
\end{array}
$$

Figure 4.4: Operational semantics : abortion

The last three rules implement the **let** construct. Rule *(LAZY)* evaluates the **let** using conventional Lazy Evaluation, creating a thunk for the right hand side of the **let**. Rules *(SPEC1)* and *(SPEC2)* implement speculative evaluation. The semantics chooses between the two approaches based on the values of $\Sigma(x)$ and $specDepth(s)$, where $\Sigma(x)$ is the depth limit for x and $specDepth(s)$ is the current speculation depth.

- Rule *(LAZY)* is used if the depth limit, $\Sigma(x)$, is less than or equal to the current speculation depth $specDepth(s)$. The rule builds a thunk α in the heap, binds x to α, and then evaluates the body of the **let**.
- Rule *(SPEC1)* is used if $\Sigma(x)$ is greater than the current speculation depth $specDepth(s)$. The rule begins speculative evaluation of the right-hand side E. A speculative return continuation is pushed, causing the body of the **let** to be evaluated once speculative evaluation of the right hand side has completed.
- Rule *(SPEC2)* is used once the speculation completes. The rule binds x to the returned reference and proceeds to evaluate the **let** body. It should be noted that neither *(SPEC1)* nor *(SPEC2)* allocate any closures in the heap.

4.2.3 Abortion Transitions

The process of abortion is described by the $\rightsquigarrow$ rules given in Figure 4.2.3. These rules allow a program to back out of a speculation which has turned out to be too expensive:

- Rule *(!EXP)* creates a thunk in the heap for the current expression. This rule is rather like rule *(VAL)*, except that rule *(VAL)* turns a value into a value closure, while rule *(!EXP)* turns an arbitrary expression into a thunk.

Haskell, in which pure expressions can raise exceptions but not catch them; exceptions can only be caught in the IO Monad [Wad95, PW93, Wad97].

$$\textit{(RUN)}\quad \frac{\Gamma; c; s \quad \longrightarrow_\Sigma \quad \Gamma'; c'; s'}{\Gamma; c; s; t \quad \curvearrowright_\Sigma \quad \Gamma'; c'; s'; t+1} \quad \text{if } t \leq MAXTIME \quad \text{and } specDepth(s) > 0$$

$$\textit{(RESET)}\quad \frac{\Gamma; c; s \quad \longrightarrow_\Sigma \quad \Gamma'; c'; s'}{\Gamma; c; s; t \quad \curvearrowright_\Sigma \quad \Gamma'; c'; s'; 0} \quad \text{if } specDepth(s) = 0$$

$$\textit{(ABORT)}\quad \frac{\Gamma; c; s \quad \rightsquigarrow \quad \Gamma'; c'; s'}{\Gamma; c; s; t \quad \curvearrowright_\Sigma \quad \Gamma'; c'; s'; t} \quad \text{if } t > MAXTIME \quad \text{and } specDepth(s) > 0$$

Figure 4.5: Operational semantics : bounded speculation

- Rule *(!RET)* converts a return command, $\triangledown\alpha$, into a demand command, $\circledcirc\alpha$. Like rule *(!EXP)* the only purpose of this rule is to simplify the state in preparation for other rules.
- Rule *(!SPEC)* starts evaluating the body of a $\mathbf{let}$, despite the fact that its right hand side has not finished evaluating. This rule captures the essence of abortion.
- Rule *(!UPD)* removes an update frame from the stack, binding its updatee, x', to a partially evaluated result.
- Rule *(!ABORT)* removes a local frame from the stack and moves it to a suspension closure in the heap, together with the identifier α. If the value of this suspension frame is demanded then rule *(RESUME)* will place the frame back onto the stack and demand α again.

4.2.4 Bounded Speculation

As discussed in Section 2.2, Optimistic Evaluation will abort any speculation that goes on for too long. To formalise this, we introduce a new transition relation, $\curvearrowright$:

$$\Gamma; c; s; t \;\curvearrowright_\Sigma\; \Gamma'; c'; s'; t'$$

The components of the state are the same as for $\longrightarrow$, except that we add a new component t, representing the *speculation time*. The speculation time is the number of steps that have been taken since the virtual machine last started evaluating speculatively. The rules for bounded speculation are given in Figure 4.5. If the program has been evaluating speculatively for less than $MAXTIME$ steps then rule *(RUN)* evaluates it as normal. If the program stops speculating then rule *(RESET)* resets t to 0. If the program evaluates speculatively for more that $MAXTIME$ steps then rule *(ABORT)* will apply abortion transitions until all speculations have been aborted.

We give an improved version of this semantics in Section 6.3.1.

4.2.5 Well Formed States

We say that a state $\Gamma; E; s$ is *well formed* if no heap reference α is bound in both the heap Γ and the stack s, and additionally no heap reference α is bound by more than one update frame in the stack s. An initial state will always be well formed because its heap and stack are both empty. We can observe that the evaluation and abortion rules given in Figures 4.3 and 4.2.3 preserve well-formedness.

4.2.6 Exceptions and Case Statements

The language presented in Section 4.1 includes exceptions and **case** statements. In the semantics given in Figure 4.3 we showed that neither of these pose any real problems for Optimistic Evaluation.

In the chapters that follow, we will work with a simplified language that has neither **case** statements, nor exceptions. The revised expression form is thus the following:

Expression	E	::=	x	variable
		|	$\mathbf{let}\ x = E\ \mathbf{in}\ E'$	thunk creation
		|	$\lambda x.E$	function abstraction
		|	$E\ x$	function application
		|	n	integer constant
		|	$x \otimes x'$	primitive operation

ValueExp	V	::=	$\lambda x.E \mid n$

The stack frames remain the same, except that **case** return frames are removed.

Neither **case** statements nor exceptions increase the expressiveness of the language: both features can be encoded using the features of this simplified language. By removing these features from the language, we are able to reduce the size of the proofs and semantic models that we present in subsequent chapters. We claim that introducing **case** or exceptions would present no significant problems.

4.3 Denotational Semantics

While an operational semantics formalises the way in which a program executes, it does not give a clear view of what a program actually means. This purpose is better served by a denotational semantics. Our denotational semantics makes no reference to the order of evaluation or to whether expressions are evaluated or not. It thus allows us to present a considerably simpler model of what our language means.

Perhaps more critically, by proving that our operational semantics is sound with respect to our denotational semantics, we can demonstrate that changing the evaluation order by means of the speculation configuration Σ has no effect on the meaning of the program (Section 4.4.1).

The semantics we give here is influenced by that used by Launchbury [Lau93], and the work of Abramsky [Abr90] and Ong [Ong88]. For a background in Domain Theory, refer to Abrasmsky and Jung [AJ94].

4.3.1 Semantic Functions

We introduce semantic functions $\mathcal{E}[\![-]\!]$, $\mathcal{C}[\![-]\!]$, $\mathcal{H}[\![-]\!]$, $\mathcal{K}[\![-]\!]$, $\mathcal{S}[\![-]\!]$ and $\mathcal{M}[\![-]\!]$ that map the syntactic objects of Section 4.2 to their semantic meanings:

$$\begin{array}{rcl}
\mathcal{E}[\![-]\!] & : & \mathit{Expression} \to \mathit{Env} \to \mathit{Value} \\
\mathcal{C}[\![-]\!] & : & \mathit{Command} \to \mathit{Env} \to \mathit{Value} \\
\mathcal{H}[\![-]\!] & : & \mathit{Heap} \to \mathit{Env} \to \mathit{Env} \\
\mathcal{K}[\![-]\!] & : & \mathit{Closure} \to \mathit{Env} \to \mathit{Value} \\
\mathcal{S}[\![-]\!] & : & \mathit{Stack} \to \mathit{Env} \to \mathit{Value} \to \mathit{Env} \\
\mathcal{L}[\![-]\!] & : & \mathit{LocalFrame} \to \mathit{Env} \to \mathit{Value} \to \mathit{Value} \\
\mathcal{M}[\![-]\!] & : & \mathit{State} \to \mathit{Value}
\end{array}$$

where Value and Env are defined in Sections 4.3.2 and 4.3.4 respectively. These semantic functions can be understood as follows:

- $\mathcal{E}[\![E]\!]_\rho$ is the value that the expression E would evaluate to in the environment ρ.
- $\mathcal{C}[\![c]\!]_\rho$ is the value that the command c would return in the environment ρ.
- $\mathcal{H}[\![\Gamma]\!]_\rho$ is the environment containing the meanings of the bindings in the heap Γ. Each binding is has its meaning taken relative to the environment ρ.
- $\mathcal{K}[\![K]\!]_\rho$ is the value that the closure K would evaluate to in the environment ρ.
- $\mathcal{S}[\![s]\!]_\rho\ v$ is the environment containing the meanings of the bindings in the stack s, given that the value v has been returned to it. This environment will contain a binding for the return value and a binding for each update frame in the stack.
- $\mathcal{L}[\![l]\!]_\rho\ v$ is the value that the local stack frame l would produce if v was returned to it.
- $\mathcal{M}[\![\Gamma; c; s]\!]$ is the value that the virtual machine state '$\Gamma; c; s$' would terminate with if executed.

We explain the significance of each semantic function in more detail in the following sections.

4.3.2 Values

The domain[4] of semantic values is a solution to the following domain equation:

$$Value = (Value \rightarrow Value) \oplus \mathbb{Z}_{\perp}$$

A semantic value is either a continuous function from a value to a value, an integer, or the bottom value $\perp$. $\perp$ is used to denote non-termination or an undefined value.

4.3.3 Identifiers

An identifier (Id) is either a local variable (x) or a heap reference (α). The denotational semantics does not distinguish between the two, considering both to be abstract symbols that can be mapped to values. Within the denotational semantics, we will write x to refer to an identifier, irrespective of whether it is a variable or a heap reference.

4.3.4 Environments

An *environment* ρ is a function that maps identifiers to their values:

$$Env \quad \stackrel{\text{def}}{=} \quad Id \rightarrow Value$$

We write $(x_1 \mapsto v_1, x_2 \mapsto v_2, \ \ldots)$ to denote the environment that maps x_1 to v_1, x_2 to v_2 etc, and that maps all other identifiers to $\perp$. We write $\rho[x \mapsto v]$ to denote the environment that is like ρ but which maps x to v.

4.3.5 Meanings of Expressions

The semantic function $\mathcal{E}[\![-]\!]$ takes a syntactic expression E and an environment ρ and maps them to the value that E would terminate with in the environment ρ. This function is continuous with respect to ρ. We define $\mathcal{E}[\![-]\!]$ as follows:

$$\mathcal{E}[\![-]\!] \ : \ Expression \rightarrow Env \rightarrow Value$$

$$\begin{array}{lcl}
\mathcal{E}[\![x]\!]_\rho & = & \rho(x) \\
\mathcal{E}[\![\mathbf{let}\ x = E\ \mathbf{in}\ E']\!]_\rho & = & \mathcal{E}[\![E']\!]_{\rho[x \mapsto \mathcal{E}[\![E]\!]_\rho]} \\
\mathcal{E}[\![\lambda x.E]\!]_\rho & = & \lambda v.\mathcal{E}[\![E]\!]_{\rho[x \mapsto v]} \\
\mathcal{E}[\![E\ x]\!]_\rho & = & \mathcal{E}[\![E]\!]_\rho\ (\rho(x)) \\
\mathcal{E}[\![x \otimes x']\!]_\rho & = & \rho(x) \mathbin{\tilde{\otimes}} \rho(x')
\end{array}$$

[4]Here *domain* means a set equipped with complete partial order (having limits of all ω chains) and having a least element $\perp$.

We write $\tilde{\otimes}$ to denote the mathematical function represented by $\otimes$.

The following results are proved by induction on the structure of expressions:

Theorem 4.3.1 (Equivalent Identifiers)
If two identifiers x and x' have the same meaning in the environment ρ then we can freely substitute x' for x in an expression E without changing the meaning of E in ρ.

$$\rho(x) = \rho(x') \;\Rightarrow\; \mathcal{E}[\![E[x'/x]]\!]_\rho = \mathcal{E}[\![E]\!]_\rho$$

Theorem 4.3.2 (Unused Identifiers)
If an identifier x is not mentioned in the expression E then the meaning of E is unaffected by the presence of a binding for this identifier in the environment ρ.

$$x \notin \mathrm{vars}(E) \;\Rightarrow\; \mathcal{E}[\![E]\!]_\rho = \mathcal{E}[\![E]\!]_{\rho[x \mapsto v]}$$

4.3.6 Meanings of Commands and Closures

The denotational semantics does not distinguish between commands and expressions. The form of a command is merely a direction to the virtual machine telling it what to do next, and so is ignored by the denotational semantics. We can thus trivially define a semantic function $\mathcal{C}[\![-]\!]$ that gives meanings to commands:

$$\begin{array}{lcl} \mathcal{C}[\![\triangledown x]\!]_\rho & = & \mathcal{E}[\![x]\!]_\rho \\ \mathcal{C}[\![\circledcirc x]\!]_\rho & = & \mathcal{E}[\![x]\!]_\rho \\ \mathcal{C}[\![E]\!]_\rho & = & \mathcal{E}[\![E]\!]_\rho \end{array}$$

The denotational meaning of a closure is similarly trivial. Value closures and indirections have the same meanings as their corresponding expressions, while the meaning of a suspension is the meaning of the stack frame, using the $\mathcal{L}[\![-]\!]$ function defined is Section 4.3.8:

$$\begin{array}{lcl} \mathcal{K}[\![(\!|V|\!)]\!]_\rho & = & \mathcal{E}[\![V]\!]_\rho \\ \mathcal{K}[\![(x)]\!]_\rho & = & \mathcal{E}[\![x]\!]_\rho \\ \mathcal{K}[\![x \angle f]\!]_\rho & = & \mathcal{L}[\![f]\!]_\rho\ \rho(x) \end{array}$$

4.3.7 Meanings of Heaps

Meanings are given to heaps using the semantic function $\mathcal{H}[\![-]\!]$. $\mathcal{H}[\![\Gamma]\!]_\rho$ is the environment formed by evaluating all the closures in Γ under the environment ρ. If a heap Γ maps x to a closure K then $\mathcal{H}[\![\Gamma]\!]_\rho$ will map x to $\mathcal{K}[\![K]\!]_\rho$.

$$\mathcal{H}[\![-]\!] \;:\; \mathit{Heap} \to \mathit{Env} \to \mathit{Env}$$

$$\mathcal{H}[\![x_1 \mapsto E_1,\ \ldots\ , x_n \mapsto E_n]\!]_\rho = (x_1 \mapsto \mathcal{K}[\![E_1]\!]_\rho,\ \ldots\ , x_n \mapsto \mathcal{K}[\![E_n]\!]_\rho)$$

We observe that $\mathcal{H}[\![-]\!]$ is continuous with respect to ρ.

4.3.8 Meanings of Stacks

The meaning of a stack is similar to the meaning of a heap. The semantic function $\mathcal{S}[\![-]\!]$ takes a stack s, an environment ρ, and a value v and maps them to the environment that is produced by returning v and performing all the updates on the stack:

$$\mathcal{S}[\![-]\!] \ :\ \mathit{Stack} \to \mathit{Env} \to \mathit{Value} \to \mathit{Env}$$

$$\begin{array}{lcl}
\mathcal{S}[\![[\,]]\!]_\rho\ v & = & (\epsilon \mapsto v) \\
\mathcal{S}[\![\{x\}E : s]\!]_\rho\ v & = & \mathcal{S}[\![s]\!]_\rho\ \mathcal{E}[\![E]\!]_{\rho[x \mapsto v]} \\
\mathcal{S}[\![\#x : s]\!]_\rho\ v & = & \mathcal{S}[\![s]\!]_\rho\ v\ \sqcup (x \mapsto v) \\
\mathcal{S}[\![l : s]\!]_\rho\ v & = & \mathcal{S}[\![s]\!]_\rho\ (\mathcal{L}[\![l]\!]_\rho\ v)
\end{array}$$

This definition deserves some explanation:

- The empty stack behaves like an update frame for the result of the program. It defines a mapping from ϵ (the identifier for the program result[5]) to the value returned.
- A speculative return frame returns the value of the **let** body, E, to the stack. Note that there is no requirement that v be non-$\bot$; this reflects the real behaviour of Optimistic Evaluation, in which a non-terminating speculation will be aborted and its suspension will be returned to the speculation frame.
- An update frame creates a binding from its updatee to the returned value.
- If the topmost frame is a local frame, then we use the semantic function $\mathcal{L}[\![-]\!]$ to produce a new value, and return that. We define the semantic function $\mathcal{L}[\![-]\!]$ as follows:

$$\mathcal{L}[\![-]\!] \ :\ \mathit{LocalFrame} \to \mathit{Env} \to \mathit{Value} \to \mathit{Value}$$

$$\begin{array}{lcl}
\mathcal{L}[\![@x]\!]_\rho\ v & = & v\ (\rho(x)) \\
\mathcal{L}[\![n\otimes]\!]_\rho\ n' & = & n \mathbin{\tilde{\otimes}} n' \\
\mathcal{L}[\![\otimes x]\!]_\rho\ n & = & n \mathbin{\tilde{\otimes}} \rho(x)
\end{array}$$

- The least upper bound ($\sqcup$) is only defined if the stack s does not contain any other binding for x. Fortunately, this will be the case for all well-formed states (see Section 4.2.5).

We observe that $\mathcal{S}[\![-]\!]$ is continuous with respect to both ρ and v.

[5] The reason for this choice of symbol is revealed in Section 5.2.3.

4.3.9 Meanings of States

Given the meaning of a heap, a command, and a stack, we can produce the meaning of a complete program state, as defined is Section 4.2.1. The meaning $\mathcal{M}[\![\Gamma; c; s]\!]$ of a state '$\Gamma; c; s$' is the value that that state would terminate with if executed. To obtain this return value we construct an environment containing all bindings held in the heap, Γ, and the stack, s, and select the value for ϵ from this environment:

$$\mathcal{M}[\![-]\!] \ : \ \mathit{State} \rightarrow \mathit{Value}$$

$$\mathcal{M}[\![\Gamma; c; s]\!] = (\mu\ \rho.\ \ \mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{C}[\![c]\!]_\rho)\ \epsilon$$

Here, μ is the least fixed point operator. It selects the least environment ρ (as defined by $\sqsubseteq$) such that:

$$\rho = \mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{C}[\![c]\!]_\rho$$

The purpose of the fixed point is to allow recursive bindings between the heap and stack.

The least upper bound used here may not be defined if the heap and stack both contain bindings for the same heap reference α. Fortunately this will never be the case for a well-formed state (see Section 4.2.5).

Theorem 4.3.3 (Extension of the Heap)
We observe that the meaning of a state is preserved if the heap Γ is extended with a binding for a new identifier x:

$$x \notin \mathrm{vars}(\Gamma) \wedge x \notin \mathrm{vars}(s) \wedge x \notin \mathrm{vars}(c) \ \ \Rightarrow\ \ \mathcal{M}[\![\Gamma[x \mapsto K]; c; s]\!] = \mathcal{M}[\![\Gamma; c; s]\!]$$

where $vars$ is the set of all identifiers (variables and heap identifiers) mentioned in its argument.

4.4 Soundness

If we are to use Optimistic Evaluation as a drop in replacement for Lazy Evaluation then it is essential that Optimistic Evaluation always gives the same results as Lazy Evaluation. In order to do this, it must be both *sound* and *complete*:

Sound: If a program terminates with a value, that value will be the same value that the program would have terminated with under Lazy Evaluation.

Complete: If a program would terminate under Lazy Evaluation, then it will also terminate under Optimistic Evaluation.

Theorem 4.4.1 (Soundness)
Lazy Evaluation is a special case of Optimistic Evaluation, arising when Σ maps all $\mathbf{let}$ identifiers to 0. It is thus sufficient to prove that, for any given start state, there is only one result that Optimistic Evaluation can return, irrespective of speculation configuration. That is:

$$\emptyset; E; [\,] \longrightarrow^*_{\Sigma} \Gamma; \nabla x; [\,] \quad \wedge \quad \emptyset; E; [\,] \longrightarrow^*_{\Sigma'} \Gamma'; \nabla x'; [\,] \quad \Rightarrow \quad \mathcal{E}[\![x]\!]_{\mathcal{H}[\![\Gamma]\!]} = \mathcal{E}[\![x']\!]_{\mathcal{H}[\![\Gamma']\!]}$$

We prove the following stronger property:

$$\Gamma; c; s \quad \longrightarrow_{\Sigma} \quad \Gamma'; c'; s' \quad \Rightarrow \quad \mathcal{M}[\![\Gamma; c; s]\!] \quad = \quad \mathcal{M}[\![\Gamma'; c'; s']\!]$$

If $\longrightarrow$ preserves the meaning of a state, then any final state must have same meaning as the initial state. Given that no initial state can have more than one meaning, it must be the case that all terminal states have the same meaning. Given that the meaning of a terminal state is the meaning of the value returned, it must be the case that all possible terminal states return the same value.

The proof of this property is given in Appendix A. We proceed case-wise, demonstrating that the property holds for all rules defining '$\longrightarrow$' and '$\rightsquigarrow$'.

4.4.1 Completeness

We do not provide a proof of completeness. This is partly because such a proof is quite difficult to obtain, but primarily because, for our purposes, completeness is not a particularly interesting property.

The purpose of Optimistic Evaluation is to evaluate programs more efficiently than Lazy Evaluation. It is thus not sufficient for Optimistic Evaluation to be guaranteed to terminate if Lazy Evaluation terminates — the number of steps required for it to terminate must be within some bound of the number of steps required by Lazy Evaluation.

We develop an informal proof of the efficiency of Optimistic Evaluation in Section 6.3.3. The completeness of Optimistic Evaluation follows as a corrollary of this result.

CHAPTER 5

A Cost Model for Non-Strict Evaluation

Although the speculation configuration Σ has no effect on the value that a program produces, it does have an effect on the amount of time needed for this value to be produced.

In this chapter we give a denotational semantics that formalises the cost of evaluating an expression. The costs given by this semantics are independent of evaluation strategy but relate closely to real evaluation costs—allowing such costs to be easily derived. This cost model forms the basis of the online profiling techniques that we describe in Chapter 6.

This Chapter is structured as follows:

- In Section 5.1 we explain why we need a cost model.
- In Section 5.2 we define the concept of a cost graph.
- In Sections 5.3 and 5.4 we explain how one can create a cost graph for a program.
- In Section 5.5 we introduce an operational semantics that keeps track of the amount of work it has done.
- In Section 5.6 we give denotational meanings to the states of the costed operational semantics and demonstrate that our denotational cost model accurately models the costs experienced by the operational semantics.

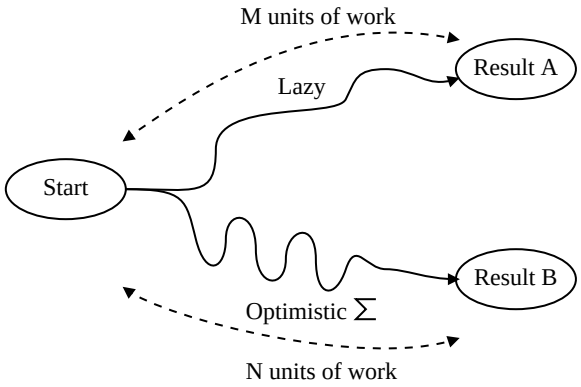


Figure 5.1: Comparing Optimistic Evaluation to Lazy Evaluation

5.1 Why We Need a Cost Model

The aim of Optimistic Evaluation is to evaluate non-strict programs faster than Lazy Evaluation. It is thus important to be able to prove that the performance of Optimistic Evaluation will never be significantly worse than that of Lazy Evaluation. Doing this turns out to be somewhat tricky.

5.1.1 The Challenges Faced by the Online Profiler

It is easy to see that there are some choices of Σ for which Optimistic Evaluation will be slower than Lazy Evaluation. For example, if our program is:

$$\mathbf{let}\ x = \texttt{expensive}\ \mathbf{in}\ 3$$

then any speculation configuration Σ that chooses to speculate x is likely to be slower than Lazy Evaluation. Similarly, it is easy to see that there are some choices of Σ for which Optimistic Evaluation will outperform Lazy Evaluation. For example, if our program is:

$$\mathbf{let}\ x = 1 + 2\ \mathbf{in}\ x + 1$$

then any speculation configuration Σ that chooses to speculate x is likely to be faster than Lazy Evaluation.

What is not easy is to tell, at runtime, whether the current value for Σ is outperforming Lazy Evaluation, and how Σ can be changed so as to improve performance. This is the task that the online profiler faces.

If we are allowed to run a program to completion, then it is easy to see how Σ compares to Lazy Evaluation; we simply run the program with Σ, run it with Lazy Evaluation, and compare the amount of work done (Figure 5.1). However the online profiler does not have this luxury; it cannot wait until the program has terminated before adjusting Σ: once the program has finished, it is too late to make it run faster. The profiler thus needs to have a way of estimating the

$\textbf{let}\ x = \texttt{<expensive>}\ \textbf{in}$
$\textbf{let}\ y = x + 3\ \textbf{in}$
$\textbf{let}\ z = x + 3\ \textbf{in}$
$\textbf{let}\ p = y + z\ \textbf{in}$
`<rest of program>`

Figure 5.2: A simple program

performance of Σ relative to Lazy Evaluation, without actually performing the Lazy Evaluation, and without waiting for the program to finish.

This job of the profiler is made yet harder by the fact that:

- It is essential that the profiler does not believe that Σ is outperforming Lazy Evaluation if it is in fact performing significantly worse than Lazy Evaluation.
- The profiler has to work with very limited information. In particular, it does not know for sure which of its computations will turn out to be needed.
- The profiler must itself impose a very low overhead on the program, lest the costs of profiling outweigh the benefits of Optimistic Evaluation.

In order to do all of this, it is necessary that we have a good understanding of the costs of non-strict programs. In particular, it is necessary that we have a solid cost model which we can use to justify and verify our online profiler.

5.1.2 The Problems with Simple Cost Models

How much does it cost to evaluate $x + 3$?

The obvious answer would be that the cost of an evaluation is the number of steps taken to perform it; however this turns out to behave poorly for non-strict languages. Consider the program in Figure 5.2:

- How expensive is y? If x is already evaluated then y will be cheap to evaluate—because it only needs to perform an integer addition. However, if x is unevaluated, then y will be expensive—because it needs to evaluate x before it can produce a result.
- How expensive is z? It seems sensible that z should have the same cost as y; but, if x is unevaluated, then one of y and z will look at x first and so appear more expensive.
- How expensive is p? Can I obtain the cost of p by combining the costs of y and z?
- Finally, is p's value actually needed? Should this affect its cost?

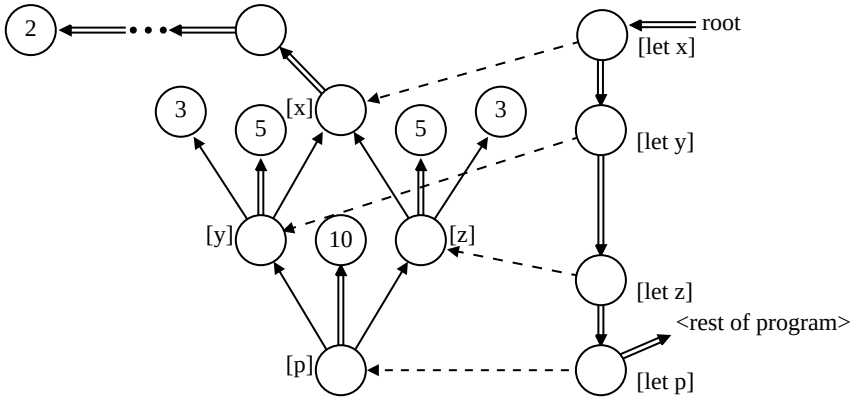


Figure 5.3: Cost graphs illustrate the dependencies between computations

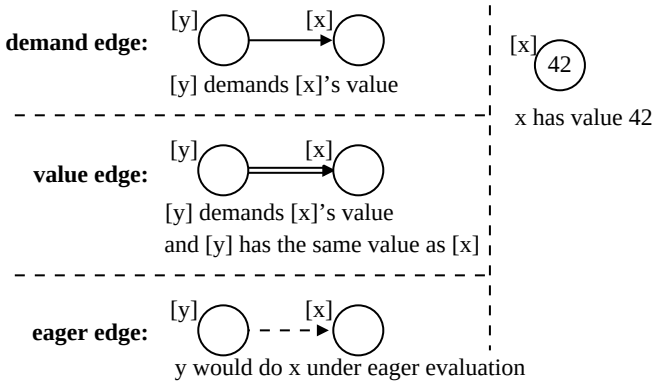


Figure 5.4: Key: Edges can be demand edges, value edges, or eager edges

In this chapter we give a new, denotational, cost semantics for a non-strict language, that takes full account of sharing. The cost model is compositional: the cost of an evaluation is obtained by combining the costs of its fragments. Furthermore the semantics is independent of evaluation strategy, so it can describe the cost of Eager Evaluation, Lazy Evaluation, or anything in between.

5.2 Cost Graphs

In our cost semantics, the meaning of a program[1] is its *cost graph*. For example, we can represent the costs involved in evaluating the program from Figure 5.2 using the cost graph shown in Figure 5.3.

A cost graph gives an unrolled trace of the computations that can be done by the program. Each node represents a computation that takes one unit of time and produces a value. An arrow from a node i to a node i' denotes the fact that the computation i depends on the value produced by the computation i'. If the arrow is thick, then the value produced by i is the same as that

[1]For the purposes of this chapter, a program is a closed expression.

produced by i'. If the arrow is dashed, then the dependency only exists under Eager Evaluation. We refer to plain links as *demand edges*, thick links as *value edges* and dashed links as *eager edges*. This is illustrated in Figure 5.4.

Computation nodes are classified into *value nodes* and *dependent nodes*:

- A *value node* is labelled with the value it produces, and does not have links to any other node. Such a node represents a computation that places a value in the program heap.

- A *dependent node* is not labelled with a value; instead it has a value edge linking it to another node. A dependent node may also have demand edges and eager edges.

The graph in Figure 5.3 tells us that p does one unit of work (to perform an addition) and depends on y, z and a computation that creates a 10 value. We can similarly see that y and z both demand the value of x, and also depend on computations that produce a 3 and a 5.

The graph also tells us the values that computations produce. We can see that p has the value 10, and y and z are both 5. In order to find the value for x, we must follow a long chain of value edges, eventually reaching a value node holding 2. In some cost graphs, the chain of value edges leading from a dependent node may be infinite. In this case, the value of the node is $\perp$, representing non-termination.

In this graph we have annotated nodes with variable identifiers (e.g. $[x]$). In reality, we must distinguish multiple instantiations of the same variable, so we need a more refined naming scheme. We discuss such a scheme in Section 5.2.3.

5.2.1 Work Sets

Given a cost graph, we can represent the work required to perform a particular evaluation using a subset of the nodes in the cost graph. Such a set is known as a *work set*.

The work required to evaluate a computation lazily is the set of nodes in the cost graph that are reachable from that computation's node using only value and demand edges. Figure 5.5 and Figure 5.6 illustrate the work required to evaluate z and y respectively.

The work required to evaluate several computations is the union of the work required to do each computation on its own. Figure 5.7 illustrates the work required to evaluate both y and z lazily.

If work has already been done, or must be done later, then we can subtract it. Figure 5.8 illustrates the work required to evaluate z, excluding the work required to evaluate y.

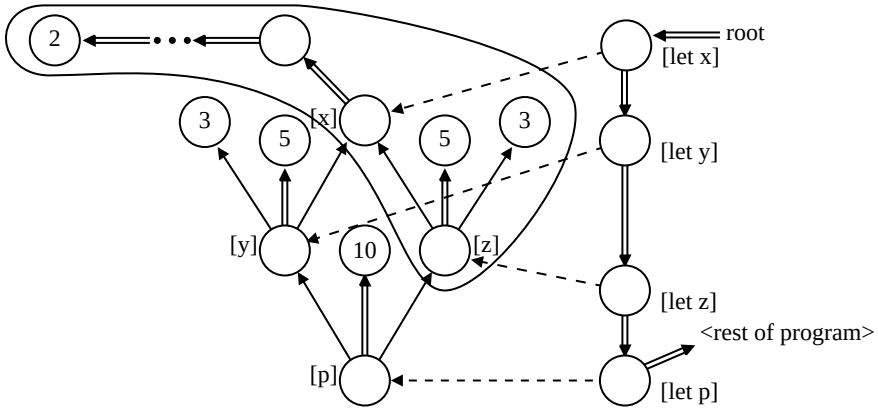


Figure 5.5: Work required to evaluate z lazily

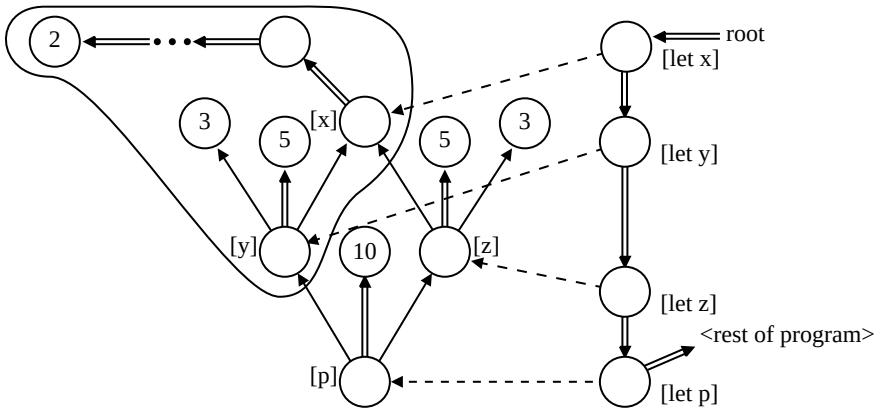


Figure 5.6: Work required to evaluate y lazily

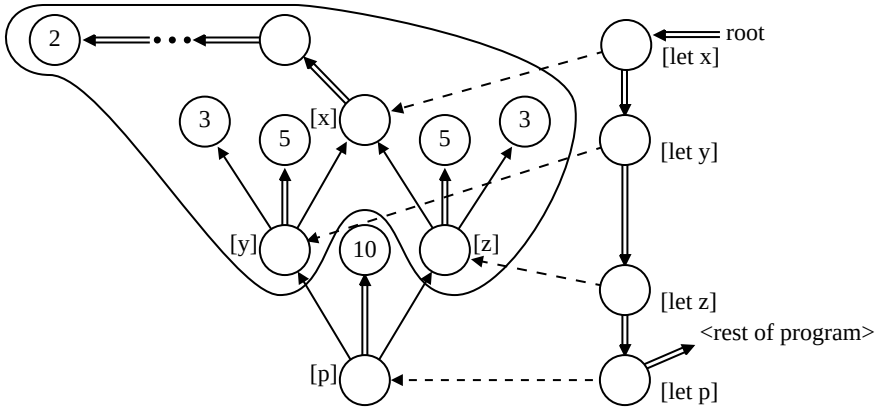


Figure 5.7: Work required evaluate both y and z lazily

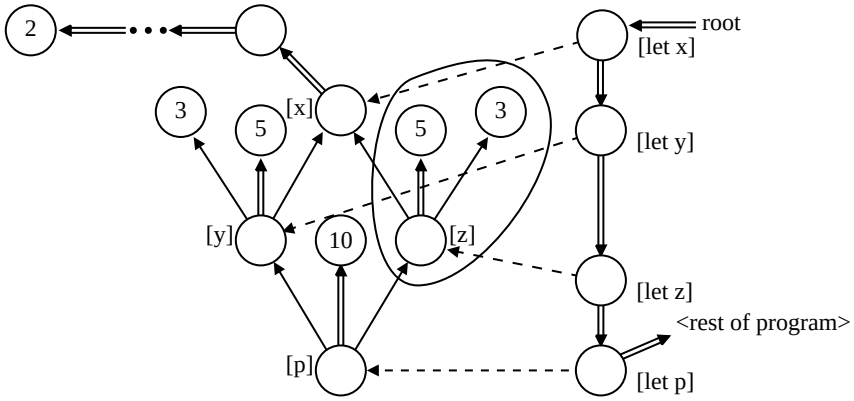


Figure 5.8: Work required evaluate z given that y has already been evaluated, or must be evaluated later

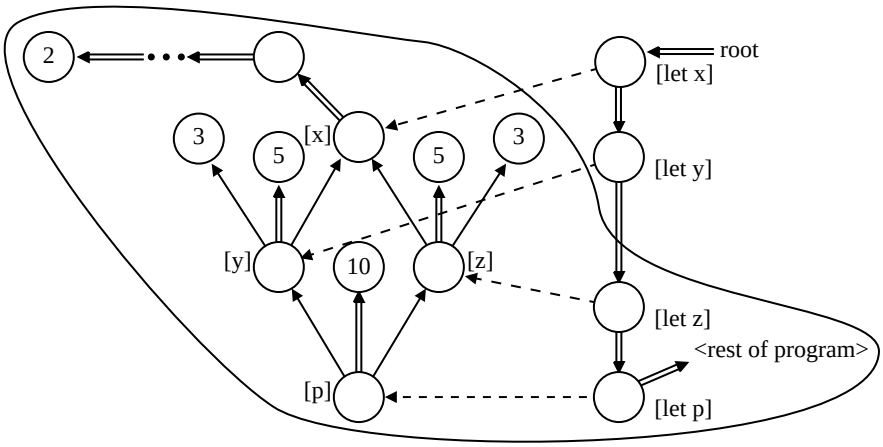


Figure 5.9: Work required to evaluate `[let z]` entirely eagerly

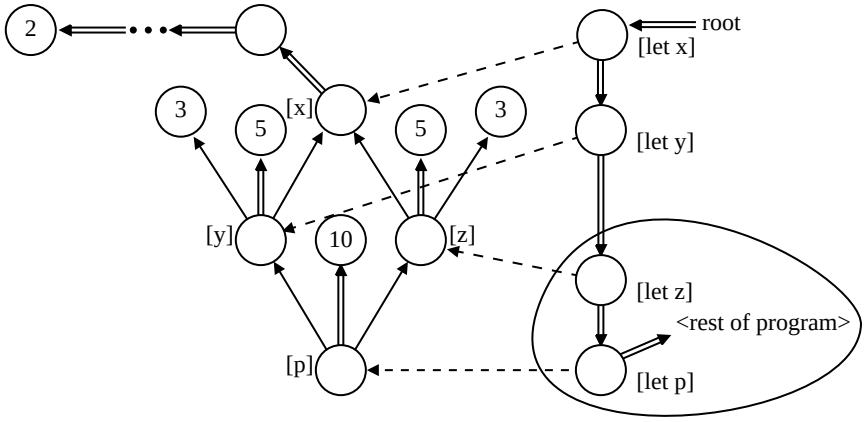


Figure 5.10: Work required to evaluate `[let z]` entirely lazily

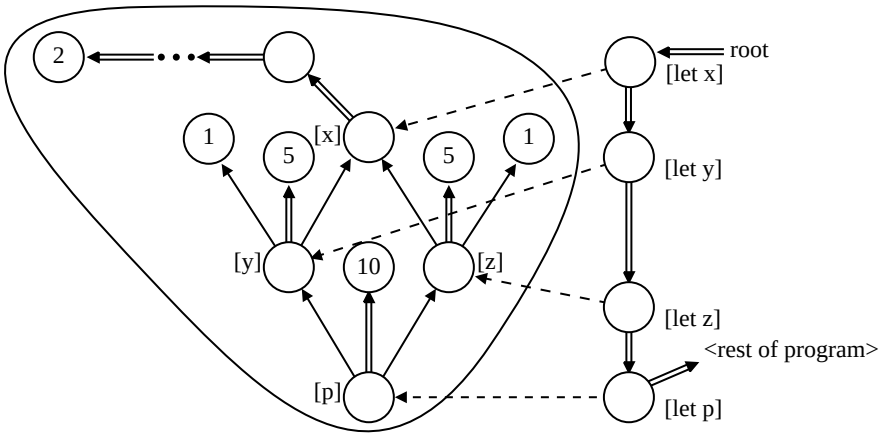


Figure 5.11: Extra work required to evaluate `[let z]` eagerly rather than lazily

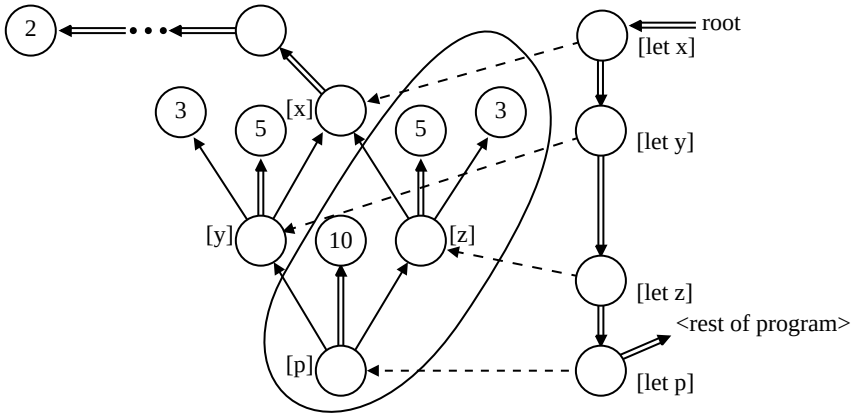


Figure 5.12: Extra work required to evaluate `[let z]` eagerly, if we know that [y] is needed by `<rest of program>`

We can also use work sets to compare evaluation strategies. Figure 5.9 illustrates the work required to evaluate '`let z`' eagerly. This is the set of all computations reachable from '`let z`', including computations reachable by eager edges. We can compare this with Figure 5.10 which shows the work required to evaluate '`let z`' lazily. By subtracting the lazy work from the eager work, we can obtain the extra work required by Eager Evaluation, which we illustrate in Figure 5.11. If it turns out that y is needed by '`<rest of program>`', then we can subtract the work required for y, and so the extra work will be that shown in Figure 5.12.

The key point to note is that work is just a set of nodes. We can take unions, intersections, differences, or do anything else that we might want to do with a set.

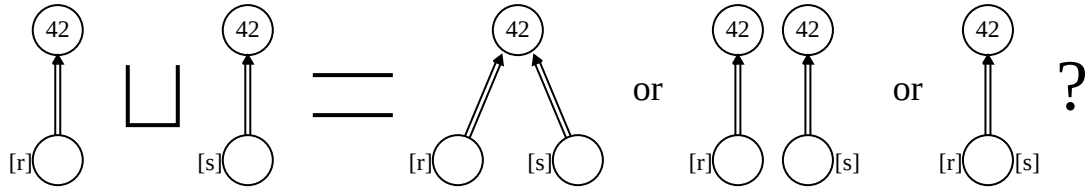


Figure 5.13: To combine two graphs, we must know which nodes are equivalent.

5.2.2 Useful Properties of Cost Graphs

Cost graphs give us a number of useful properties:

- The cost of any evaluation within a program is fully described by the cost graph for the program.
- The work required for any evaluation can be represented as a subset of the nodes in the cost graph for the program.
- The number of nodes in a work set corresponds directly to the number of steps that would be needed to perform that evaluation in a low-level operational semantics, as we show in Section 5.5.
- Work sets can be easily manipulated, allowing one to add and subtract work in a compositional way.
- Cost graphs are independent of any particular evaluation strategy.[2] For example, the cost graph in Figure 5.3 is independent of whether x, y and z were evaluated previously.

5.2.3 Giving Names to Computations

If we are to compare work sets or take their unions and intersections, then it is important that we have a standard naming scheme for nodes.

Consider the example shown in Fig.5.13. In this example we are attempting to combine the work set for evaluating r with the work set for evaluating s, creating the work set for evaluating both r and s. In order to do this, we need to know which nodes in the two sets represent the same computation. If two sets share a subcomputation then it is important that the combined set preserves this sharing. It is equally important that work sets do not consider a computation to be shared if it was in fact repeated. We prevent such problems by ensuring that *every computation within a program has a single, unique name*.

We assign every computation in a program a name that is a string of $\circ$ and $\bullet$ tokens (cf. contexts from context semantics [Mai03]). A new name can be constructed by prefixing an

[2] Provided that the strategy is a sequential hybrid of lazy and eager evaluation. Parallel strategies are not currently supported, nor are strategies that can perform reductions under lambdas.

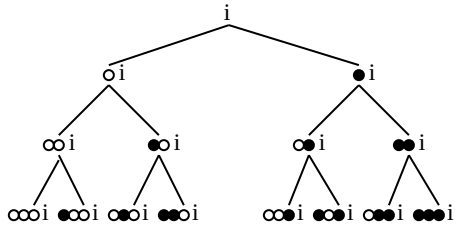


Figure 5.14: Names form a tree

existing name with additional tokens. We say that a name i is *descended* from a name i' if i' is a suffix of i. It can be helpful to visualise names as representing points within a tree, as illustrated in Figure 5.14.

We see in Section 5.3 that: *(1)* The semantic function that produces a cost graph for an evaluation is passed a root name i that it can use to create unique names for the computations it defines; and *(2)* the name of a computation represents the location it would have in the program text if all function calls were statically expanded. We see in Section 5.5 that the name of a computation also represents that point at which it would it would be evaluated in a strict language: if a name i' descends from a name i, then i' would be a subcomputation of i in a strict language.

Formally, a computation name, $Name$, is a finite string of $\circ$ and $\bullet$ tokens:

$$Name \stackrel{\text{def}}{=} \{\circ, \bullet\}^*$$

The $Work$ required by an evaluation is the set of computations that it has performed:[3] [4]

$$Work \stackrel{\text{def}}{=} \mathcal{P}(Name)$$

5.2.4 Formalising Cost Graphs and Cost Views

In the classical graph model, a *cost graph* is a 6-tuple $(N_{\mathbf{v}}, N_{\mathbf{d}}, E_{\mathbf{v}}, E_{\mathbf{d}}, E_{\mathbf{e}}, \ell)$, where:

$N_{\mathbf{v}} \subset Name$	is a set of value nodes
$N_{\mathbf{d}} \subset Name$	is a set of dependent nodes ($N_d \cap N_v = \emptyset$)
$E_{\mathbf{v}} \in (N_{\mathbf{d}} \rightarrow Name)$	is the value edges
$E_{\mathbf{d}} \subset (N_{\mathbf{d}} \times Name)$	is a set of demand edges
$E_{\mathbf{e}} \subset (N_{\mathbf{d}} \times Name)$	is a set of eager edges
$\ell \in (N_{\mathbf{v}} \rightarrow CValue \setminus \{\bot\})$	gives values to value nodes

[3] Names uniquely identify computation nodes; indeed it is convenient to think of the set of nodes as actually being a subset of $Name$.

[4] $\mathcal{P}(Name)$ is the *power set* of $Name$. This is the set of all subsets of $Name$.

The $CValue$ domain referred to here is not the $Value$ type defined in Chapter 4, but a new domain, which we define in Section 5.3.1.

Note that a cost graph can be an open graph: a node may have an edge that links to a name that is not a node in the graph. Open graphs arise as a result of non-termination, as we describe in Section 5.4.3.

For the purposes of our semantics, it is convenient to instead represent a cost graph as a partial function that maps a node onto its value, its immediately reachable nodes, or $\bot$. We refer to this representation as a *cost view* (CV), defined to be a domain[5] that is a solution to the following domain equation ($CValue$ is defined mutually recursively in Section 5.3.1):

$$CV \;=\; Name \;\to\; CValue \;\oplus\; (Name \times \mathcal{P}(Name) \times \mathcal{P}(Name))_\bot$$

where the right summand is interpreted as a flat domain.

A cost view γ will map a computation name i to $\bot$ if i is not a computation in the cost graph. If i is a value node then $\gamma(i)$ will be its value (discussed in Section 5.3.1). If i is a dependent node then $\gamma(i)$ will be (i', d, e) where i' is the destination of the value edge from i, and d and e are sets containing the destinations of the demand edges and eager edges from i respectively. We say γ *defines* i if $\gamma(i) \neq \bot$. We say that γ is a *complete view* if it defines every node in the cost graph for the program being analysed.

There is a bijection between these two representations. Given a cost graph in the classical representation, we can produce the corresponding cost view as follows:

$$\gamma(i) = \begin{cases} \bot & \text{if } i \notin (N_{\mathbf{v}} \cup N_{\mathbf{d}}) \\ \ell(i) & \text{if } i \in N_{\mathbf{v}} \\ (E_{\mathbf{v}}(i), \{i' \mid (i,i') \in E_{\mathbf{d}}\}, \{i' \mid (i,i') \in E_{\mathbf{e}}\}) & \text{if } i \in N_{\mathbf{d}} \end{cases}$$

The inverse translation is also simple:

$$\begin{array}{rcrcl} N_{\mathbf{v}} & = & \{i & \mid & \gamma(i) = v \;\wedge\; v \in CValue \setminus \{\bot\}\} \\ N_{\mathbf{d}} & = & \{i & \mid & \gamma(i) = (i', d, e)\} \\ E_{\mathbf{v}} & = & \{(i \mapsto i') & \mid & \gamma(i) = (i', d, e)\} \\ E_{\mathbf{d}} & = & \{(i, i') & \mid & \gamma(i) = (i'', d, e) \;\wedge\; i' \in d\} \\ E_{\mathbf{e}} & = & \{(i, i') & \mid & \gamma(i) = (i'', d, e) \;\wedge\; i' \in e\} \\ \ell & = & \{(i \mapsto v) & \mid & \gamma(i) = v \;\wedge\; v \in CValue \setminus \{\bot\}\} \end{array}$$

We define $\lfloor i \rfloor_\gamma$ to be the value produced by the computation i, in the cost view γ. $\lfloor - \rfloor$ follows value edges until it finds a value. $\lfloor - \rfloor$ is defined to be a solution to the following

[5] As before, we use the term *domain* to mean a set equipped with complete partial order (having limits of all ω chains) and having a least element $\bot$.

equation:

$$\lfloor - \rfloor : Name \to CG \to CValue$$

$$\lfloor i \rfloor_\gamma \;=\; \begin{cases} v & \text{if } (i \mapsto v) \in \gamma \\ \lfloor i' \rfloor_\gamma & \text{if } (i \mapsto (i', d, e)) \in \gamma \\ \bot & \text{if } (i \mapsto \bot) \in \gamma \end{cases}$$

If the chain of value dependencies from i is infinite, then $\lfloor i \rfloor_\gamma$ will be $\bot$.

5.2.5 The Real Cost of an Evaluation Strategy

An *evaluation strategy* defines a hybrid of lazy and eager evaluation. We define a $Strategy$, Ψ, to be the set of computations that should be evaluated eagerly.

$$Strategy \stackrel{\text{def}}{=} \mathcal{P}(Name)$$

If $i \in \Psi$ then i depends on the computations reachable by its eager edges, as well as those reachable by its value or demand edges, otherwise i only depends on those nodes reachable by its value or demand edges. Lazy Evaluation corresponds to the empty set $\emptyset$, while Eager Evaluation corresponds to the set of all computation names $Name$.

Given a complete cost graph γ and a strategy Ψ, the function $\mathcal{W}\{\!\{-\}\!\}$ defines the work required to produce values for the computations in the set D:

$$\begin{array}{rcl} \mathcal{W}\{\!\{-\}\!\} & : & \mathcal{P}(Name) \to CV \to Strategy \to Work \\ \mathcal{W}\{\!\{D\}\!\}^\Psi_\gamma & = & D \;\cup\; \mathcal{W}\{\!\{ \bigcup_{j \in D} realDeps(\gamma, \Psi, j) \}\!\}^\Psi_\gamma \end{array}$$

$$realDeps(\gamma, \Psi, j) \;=\; \begin{cases} \{i\} \cup d \cup e & \text{if } (j \mapsto (i, d, e)) \in \gamma \;\wedge\; j \in \Psi \\ \{i\} \cup d & \text{if } (j \mapsto (i, d, e)) \in \gamma \;\wedge\; j \notin \Psi \\ \emptyset & \text{otherwise} \end{cases}$$

If evaluation would not terminate, then the resulting work set will be infinite.

We define a function $programWork$ that gives the work required to evaluate the program defined by the cost view γ, using the strategy Ψ:

$$programWork(\gamma, \Psi) \;=\; \mathcal{W}\{\!\{\{\epsilon\}\}\!\}^\Psi_\gamma$$

In Section 5.6.5 we prove that this function correctly describes the work that would be done by an operational semantics when evaluating a program.

By comparing the work sets for different evaluation strategies, we can see how well they

perform. We explore this concept in more detail in Chapter 6, where we use it to motivate the design of an online profiler.

5.3 Producing a Cost View for a Program

In this Section we present a denotational semantics that produces a complete cost view for the a program. This cost view contains all computations that can possibly take place during the evaluation of the program.

The language we work with is that presented in Section 4.2.6.

5.3.1 Cost View Producers and CValues

We find it convenient to work with the notion of a *cost view producer* (CVP). The domain CVP is defined as follows:

$$CVP \quad = \quad Name \to CV \to CV$$

A cost view producer is a continuous function that will produce a new cost view γ' if it is given:

- a computation name i to use as the source for all names in γ'
- a complete cost graph γ that defines every computation that can possibly take place in the program.

By defining the type CVP it becomes possible for the types of our semantic functions to resemble those of the semantics from Chapter 4, with CVP taking on the role that $Value$ had previously.

A $CValue$, v, is either an integer or a function. The domain $CValue$ is defined to be a solution to the following domain equation, taken mutually recursively with the definition of CV:

$$CValue \quad = \quad \mathbb{Z}_\perp \quad \oplus \quad (Name \to CVP)$$

If v is a function value, then we can create a cost view producer by passing it the name of the computation that produced its argument; this cost view producer will produce a cost view for the evaluation of the function's body.

5.3.2 The Meaning of a Program

We define a semantic function $\mathcal{P}[\![-]\!]$ that produces a complete cost view for a program by taking the least fixed point of its cost view producer, for the root name ϵ. We obtain a cost view

producer for the program by applying the semantic function $\mathcal{E}[\![-]\!]$ (defined in Section 5.3.3) to its top level expression:

$$\begin{array}{lcl} \mathcal{P}[\![-]\!] & : & \mathit{Expression} \rightarrow \mathit{CV} \\ \mathcal{P}[\![E]\!] & \stackrel{\text{def}}{=} & \mu\gamma.\mathcal{E}[\![E]\!]_{\emptyset}\ \epsilon\ \gamma \end{array}$$

This fixed point is not strictly necessary. A computation can only depend on computations that would have been performed before it under Eager Evaluation. It is thus possible to give an alternative semantics which does not use a fixed point, instead passing every semantic function a cost view defining the computations that would be performed previously under Eager Evaluation. Although this semantics works, we have found it to be significantly more cumbersome than the semantics we present here. We know that a least fixed point must exist because $\mathcal{E}[\![-]\!]$ is continuous with respect to γ.

5.3.3 Meanings of Expressions

The semantic function $\mathcal{E}[\![-]\!]$ gives cost view producer meanings to expressions:

$$\mathcal{E}[\![-]\!] : \mathit{Expression} \rightarrow \mathit{Locals} \rightarrow \mathit{CVP}$$

$\mathcal{E}[\![E]\!]_{\beta}\ i\ \gamma$ denotes the cost view for the expression E where:

- β is a partial function that maps local variables to computation names.

$$\mathit{Locals} \stackrel{\text{def}}{=} \mathit{Var} \rightharpoonup \mathit{Name}$$

- i is the root name to used for nodes in the resulting view.
- γ is a complete cost view for the program (see Section 5.3.2).

The result of $\mathcal{E}[\![E]\!]_{\beta}\ i\ \gamma$ is a cost view γ' with the following properties:

- γ' defines a computation node with the name i.
- The value of i corresponds to the meaning of the expression under a conventional lazy semantics [Lau93, Abr90, Ong88].[6]
- All nodes defined in γ' have names descended from i (We say that γ' is *rooted at* i).
- Every node defined in γ' represents a computation that would take place if E were evaluated in a strict language. The computations performed to evaluate a thunk will be defined in the cost view for the thunk's definition, and not in the cost view for the thunk's user.

[6] If the value is an integer, it will be the same integer. Relating functions is somewhat harder.

- Some nodes may have dependencies on nodes that are not defined in γ', but which are defined in γ. Such *dangling dependencies* can be resolved by combining γ with a view that defines the node depended on.

- If a computation would not be performed during lazy evaluation of E then its node will not be reachable from i via demand or value edges.

- γ' may be infinite; this will be the case if eager evaluation of E would not terminate.

5.4 Rules for Evaluation

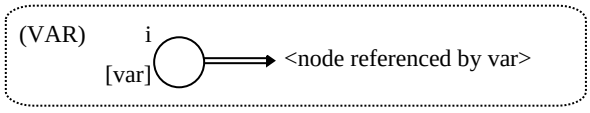
In this section we give the rules that define $\mathcal{E}[\![-]\!]$. These rules are summarised in Fig.5.15.

The following lemmas can be proved by induction over the structure of the rules defining $\mathcal{E}[\![-]\!]$:

Lemma 5.4.1 (Continuity)
$\mathcal{E}[\![-]\!]$ is continuous with respect to γ. It is this property that allows us to take a fixed point when we give meanings to programs in Section 5.3.2.

Lemma 5.4.2 (Local Names)
All names defined in $\mathcal{E}[\![E]\!]_\beta\ i$ will be descended from i. It is this property that causes the various lubs used in our definitions to be well defined.

5.4.1 Variables

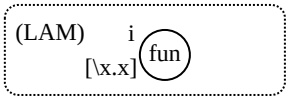


$$\mathcal{E}[\![x]\!]_\beta\ i\ \gamma \quad = \quad (i \mapsto (\beta\ x, \emptyset, \emptyset))$$

The cost view for a variable x defines a single node. This is a dependent node that represents the computation that looks up the value of x. The value edge links to the node that x is bound to in the environment β.

Note that the dependent node will depend on a node that is not defined (maps to $\perp$) in the new cost view. This reflects the fact that the computation that x references would already be evaluated in a strict language, and so would not form part of x's evaluation.

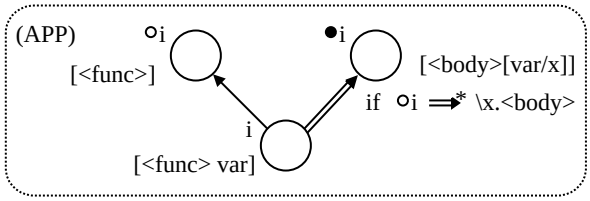
5.4.2 Lambda Expressions



$$\mathcal{E}[\![\lambda x.E]\!]_\beta\ i\ \gamma\ \ =\ \ (i \mapsto \lambda j\,.\,\mathcal{E}[\![E]\!]_{\beta[x \mapsto j]})$$

The cost view for a lambda expression also defines a single node. This node holds a function value as described in Section 5.3.1. When the function is given an argument name j it will extend the local environment to map x to j and partially apply the semantic function $\mathcal{E}[\![-]\!]$ to give a cost view producer for its body. We see an example of this in Section 5.4.6.

5.4.3 Function Application



$$\mathcal{E}[\![E\ x]\!]_\beta\ i\ \gamma\ \ =\ \ (i \mapsto (\bullet i, \{\circ i\}, \emptyset))\ \sqcup\ \mathcal{E}[\![E]\!]_\beta\ \circ i\ \gamma\ \sqcup\ \ (\lfloor \circ i \rfloor_\gamma\ (\beta\ x)\ \bullet i\ \gamma)$$

This rule is perhaps the most complex. To build the cost view for the application of an expression E to an argument x, we combine the cost views for evaluation of E and for application of E's function value to x, adding a new node which connects them.

We will describe the three parts of this view individually:

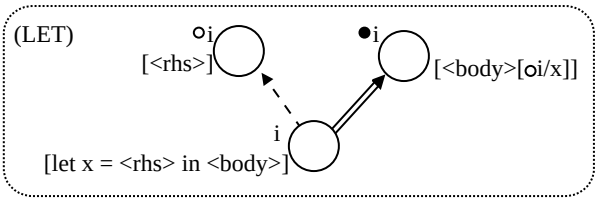
- $\mathcal{E}[\![E]\!]_\beta\ \circ i\ \gamma$ is the cost view for the evaluation of the function expression E. This evaluation is given the name $\circ i$.
- $(\lfloor \circ i \rfloor_\gamma\ (\beta\ x)\ \bullet i\ \gamma)$ is the cost view for application of the function value to the argument '$\beta\ x$'. This evaluation is given the name $\bullet i$.
- $(i \mapsto (\bullet i, \{\circ i\}, \emptyset))$ defines the computation that produces a value for '$E\ x$'. The value of '$E\ x$' is that of the evaluated function body ($\bullet i$), however the application also depends on the evaluation of the function ($\circ i$). Note the lack of a direct dependency on the function argument—as one would expect for a non-strict language.

The least upper bound is known to be well defined because all the names defined in $\mathcal{E}[\![E]\!]_\beta\ \circ i\ \gamma$ will be descended from $\circ i$, all the names defined in $(\lfloor \circ i \rfloor_\gamma\ (\beta\ x)\ \bullet i\ \gamma)$ will be descended from $\bullet i$, and $(i \mapsto (\bullet i, \{\circ i\}, \emptyset))$ only contains a binding for i. There thus be no conflicting definitions in the three cost views being combined.

The choice of which sub-evaluation to name with $\circ$ and which with $\bullet$ is arbitrary. All that matters is that both sub-evaluations are given names that descend from i and that neither name is a descendant of the other.

What happens if $\lfloor \circ i \rfloor_\gamma = \bot$? In this case no function application computations can take place and $(\lfloor \circ i \rfloor_\gamma \ (\beta \ x) \ \bullet i \ \gamma)$ will produce the empty cost view, $\bot$. i will thus have a value dependency on a node that does not exist and the corresponding cost graph will be open.

5.4.4 Let expressions



$$\mathcal{E}[\![\mathbf{let}\ x = E\ \mathbf{in}\ E']\!]_\beta\ i\ \gamma \ = \ (i \mapsto (\bullet i, \emptyset, \{\circ i\})) \ \sqcup \ \mathcal{E}[\![E]\!]_\beta \ \circ i \ \gamma \ \sqcup \ \mathcal{E}[\![E']\!]_{\beta[x \mapsto \circ i]} \ \bullet i \ \gamma$$

The meaning of a $\mathbf{let}$ expression is similar to the meaning of a function application. To build the cost view for a $\mathbf{let}$ expression, we combine the cost views for the body and right hand side of the $\mathbf{let}$, together with a new node connecting them. We will describe the three parts of this view individually:

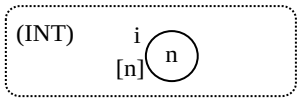
- $\mathcal{E}[\![E]\!]_\beta \ \circ i \ \gamma$ is the cost view for the evaluation of the right hand side of the $\mathbf{let}$. This evaluation is named $\circ i$.

- $\mathcal{E}[\![E']\!]_{\beta[x \mapsto \circ i]} \ \bullet i \ \gamma$ is the view for the evaluation of the $\mathbf{let}$ body, and is named $\bullet i$. The local environment β is extended to map x to the name of the right hand side. The view for the body may thus contain dependencies on $\circ i$.

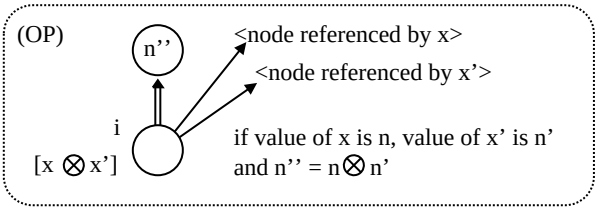
- $(i \mapsto (\bullet i, \emptyset, \{\circ i\}))$ defines the computation that evaluates the $\mathbf{let}$. The value of the $\mathbf{let}$ is that of the its body ($\bullet i$). Note that the evaluation of a $\mathbf{let}$ links to its right hand side with an eager edge rather than a demand edge, reflecting the fact that Eager Evaluation will force the right hand side to be evaluated, but Lazy Evaluation will not.

Once again, we can observe all computations defined in $\mathcal{E}[\![E]\!]_\beta \ \circ i \ \gamma$ will descend from $\circ i$ while all computations defined in $\mathcal{E}[\![E']\!]_{\beta[x \mapsto \circ i]} \ \bullet i \ \gamma$ will descend from $\bullet i$ and $(i \mapsto (\bullet i, \emptyset, \{\circ i\}))$ only gives a definition for i. We can thus see that the least upper bound will always be well defined.

5.4.5 Integers and Primitive Operations



$$\mathcal{E}[\![n]\!]_\beta\ i\ \gamma \;=\; (i \mapsto n)$$



$$\mathcal{E}[\![x \otimes x']\!]_\beta\ i\ \gamma \;=\; (i \mapsto (\bullet i, \{\beta\ x, \beta\ x'\}, \emptyset)) \sqcup (\mathcal{O}[\![\otimes]\!]\ \bullet i\ \lfloor \beta\ x \rfloor_\gamma\ \lfloor \beta\ x' \rfloor_\gamma)$$

The meaning of an integer constant is simply a value node containing that integer. A primitive operation depends on its arguments, and takes its value from the node produced by the $\mathcal{O}[\![-]\!]$ function. $\mathcal{O}[\![-]\!]$ gives meanings to primitive operators as follows:

$$\mathcal{O}[\![\otimes]\!]\ i\ v\ v' \;=\; \begin{cases} (i \mapsto (v\ \tilde{\otimes}\ v')) & \text{if } v \neq \bot\ \wedge\ v' \neq \bot \\ () & \text{otherwise} \end{cases}$$

If evaluation of one of the integer arguments does not terminate then the computation that produces the result cannot take place. The cost graph will thus be open, indicating that the primop application depends on a computation that cannot take place under any evaluation strategy.

5.4.6 An Example

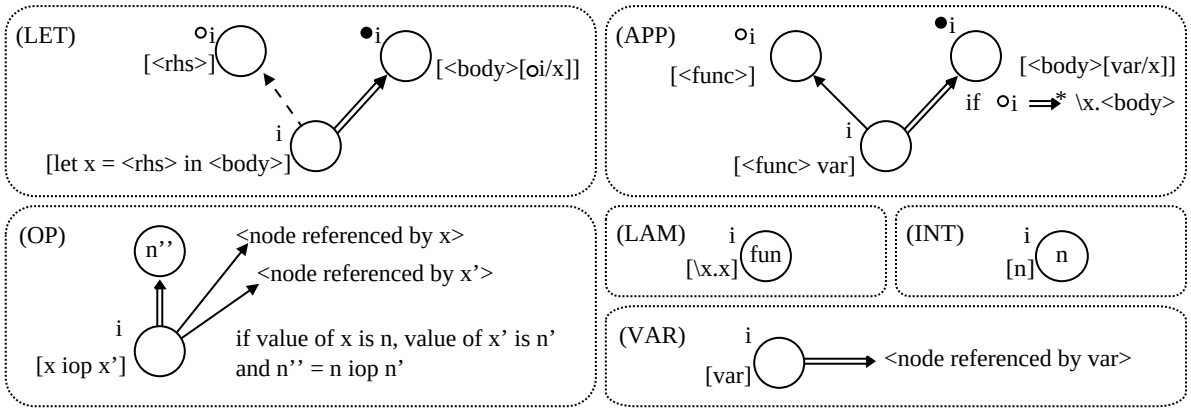
Suppose $\beta = (y \mapsto i)$ and $E = \lambda x.y\ x$, then

$$\mathcal{E}[\![E]\!]_\beta\ i'\ \gamma \;=\; (i' \mapsto \lambda j\ k\ \gamma'.(k \mapsto (\bullet k, \{\circ k\}, \emptyset), \circ k \mapsto (i, \emptyset, \emptyset)) \sqcup \mathcal{F}(\circ k)^{\bullet k}_{j}\ \ \gamma')$$

This example demonstrates the denotations of functions, application, and variables. The cost view defines only one node, which creates the function. Note that the function does not itself have a dependency on i, because evaluating the function does not require evaluation of i; however, any cost view produced by application of the function will contain a dependency on i, because applying the function does require evaluation of i.

5.5 An Operational Semantics

As we claimed in our introduction, cost graphs are independent of evaluation strategy, but relate closely to the costs incurred by a real implementation. We demonstrate this by means of an



$$
\begin{array}{rcl}
\mathcal{E}[\![x]\!]_\beta\ i\ \gamma & = & (i \mapsto (\beta\ x, \emptyset, \emptyset)) \\
\mathcal{E}[\![\lambda x.E]\!]_\beta\ i\ \gamma & = & (i \mapsto \lambda j\,.\,\mathcal{E}[\![E]\!]_{\beta[x \mapsto j]}) \\
\mathcal{E}[\![E\ x]\!]_\beta\ i\ \gamma & = & (i \mapsto (\bullet i, \{\circ i\}, \emptyset))\ \sqcup\ \mathcal{E}[\![E]\!]_\beta\ \circ i\ \gamma\ \sqcup\ (\lfloor \circ i \rfloor_\gamma\ (\beta\ x)\ \bullet i\ \gamma) \\
\mathcal{E}[\![\mathbf{let}\ x = E\ \mathbf{in}\ E']\!]_\beta\ i\ \gamma & = & (i \mapsto (\bullet i, \emptyset, \{\circ i\}))\ \sqcup\ \mathcal{E}[\![E]\!]_\beta\ \circ i\ \gamma\ \sqcup\ \mathcal{E}[\![E']\!]_{\beta[x \mapsto \circ i]}\ \bullet i\ \gamma \\
\mathcal{E}[\![n]\!]_\beta\ i\ \gamma & = & (i \mapsto n) \\
\mathcal{E}[\![x \otimes x']\!]_\beta\ i\ \gamma & = & (i \mapsto (\bullet i, \{\beta\ x, \beta\ x'\}, \emptyset)) \sqcup (\mathcal{O}[\![\otimes]\!]\ \bullet i\ \lfloor \beta\ x \rfloor_\gamma\ \lfloor \beta\ x' \rfloor_\gamma)
\end{array}
$$

Figure 5.15: The semantics of expressions, summarised graphically and formally

operational semantics. This operational semantics is parameterised by an evaluation strategy Ψ allowing it to model any blend of lazy and eager evaluation.

For each unit of computation done by the operational semantics, a closure is added to a *computation trace*. In Section 5.6.4 we prove that the closures in the computation trace map directly onto a subset of the work predicted by the $programWork$ function defined in Section 5.2.5.

5.5.1 An Operational Semantics for Cost

We define a transition relation '$\longrightarrow$' between program states, parameterised by an evaluation strategy 'Ψ'. Figure 5.16 describes the form of states while Figure 5.17 gives the rules defining '$\longrightarrow$'.

The semantics we present is very similar to the low-level semantics that we presented in Section 4.2. The key differences are the following:

- We evaluate expressions *in place*, placing the result in a specified location. The command '$E \rhd i$' instructs the virtual machine to evaluate E and to place the result in a closure with name 'i'. One notable consequence of this is that we do not need "update frames"—thunks are simply replaced by their values.

- Our states contain a *computation trace* T rather than a heap Γ. A computation trace is very similar to a heap, but contains mappings from computation names to closures, rather than from heap identifiers to closures. A computation trace contains a closure for every

State	S	::=	$T; c; s$	program state with computation trace, command and stack
Command	c	::=	$E \rhd i$	evaluate E to produce a closure with name i
		|	$\circledcirc i$	demand the value of i
		|	$\triangledown i$	return i to the top of the stack
Trace	T	::=	$\{i_j \mapsto K_j\}_0^n$	computation trace mapping names to closures
Closure	K	::=	$(\!\|V\|\!)$	value closure
		|	$\langle i, d, e \rangle$	indirection to i, demand edges d, eager edges e.
		|	E	thunk—an unevaluated expression
		|	$i \angle l$	suspended stack frame, returning i to l. From abortion.
Stack	s	::=	$[\,]$	empty stack
		|	$E \rhd i : s$	Eager return - evaluate E at i, then return to s
		|	$(l, i) : s$	Local frame - do l at i, then return to s
LocalFrame	l	::=	$@\ i$	Apply returned function to argument i.
		|	$n \otimes$	Perform operation $\otimes$ on n and returned value.
		|	$\otimes i$	Perform operation $\otimes$ on returned value and i.

Figure 5.16: Syntactic forms for states

computation that has taken place, and thus will contain more closures than would be necessary in a heap.

The evaluation strategy is dictated by Ψ. When evaluating a **let** expression, the virtual machine may choose to apply either rule *(LAZY)* or rule *(SPEC1)*, depending on whether the current computation name i is in the set Ψ.

Like the denotational semantics, the operational semantics uses strings of $\bullet$ and $\circ$ tokens to name its computations and thus also its closures. This makes it easier to relate the operational semantics to the denotational semantics. These names could be replaced by arbitrary identifiers without affecting the operational behaviour.

We can extend this semantics with a set of additional rules (given in Fig.5.18) that define abortion transitions. These transitions behave like the abortion semantics of Section 4.2.3, adapted to take account of cost.

5.5.2 How Commands Relate to the Cost Graph

Evaluation, as defined by the operational semantics of Figure 5.17, is a depth-first exploration of the cost graph for the program being evaluated. The evaluation strategy Ψ determines which edges are followed by this exploration.

The Program State

The computation trace T records all computations that have been visited/performed so far. Value closures represent value computations that have been performed and dependent closures

$$
\begin{array}{lrcl}
\textit{(VAL)} & T; V \rhd i; s & \longrightarrow_\Psi & T[i \mapsto V]; \triangledown i; s \\
\textit{(VAR)} & T; i' \rhd i; s & \longrightarrow_\Psi & T[i \mapsto \langle i', \emptyset, \emptyset\rangle]; \circledcirc i'; s \\
\textit{(DEM1)} & T[i \mapsto (\!|V|\!)]; \circledcirc i; s & \longrightarrow_\Psi & T[i \mapsto (\!|V|\!)]; \triangledown i; s \\
\textit{(DEM2)} & T[i \mapsto \langle i', d, e\rangle]; \circledcirc i; s & \longrightarrow_\Psi & T[i \mapsto \langle i', d, e\rangle]; \circledcirc i'; s \\
\textit{(DEM3)} & T[i \mapsto E]; \circledcirc i; s & \longrightarrow_\Psi & T; E \rhd i; s \\
\textit{(APP1)} & T; E\ i' \rhd i; s & \longrightarrow_\Psi & T[i \mapsto \langle \bullet i, \{\circ i\}, \emptyset\rangle]; E \rhd \circ i; ((@\ i', \bullet i) : s) \\
\textit{(APP2)} & T[i' \mapsto (\!|\lambda x.E|\!)]; \triangledown i'; ((@\ i'', i) : s) & \longrightarrow_\Psi & T[i' \mapsto (\!|\lambda x.E|\!)]; E[i''/x] \rhd i; s; \\
\textit{(OP1)} & T; j \otimes k \rhd i; s & \longrightarrow_\Psi & T[i \mapsto \langle \bullet i, \{j, k\}, \emptyset\rangle]; \circledcirc j; ((\otimes k, \bullet i) : s) \\
\textit{(OP2)} & T[j \mapsto (\!|n|\!)]; \triangledown j; ((\otimes k, i) : s) & \longrightarrow_\Psi & T[j \mapsto (\!|n|\!)]; \circledcirc k; ((n \otimes, i) : s) \\
\textit{(OP3)} & T[k \mapsto (\!|n'|\!)]; \triangledown k; ((n \otimes, i) : s) & \longrightarrow_\Psi & T[k \mapsto (\!|n'|\!)]; n \mathbin{\tilde{\otimes}} n' \rhd i; s \\
\textit{(LAZY)} & T; (\mathbf{let}\ x = E\ \mathbf{in}\ E') \rhd i; s & \longrightarrow_\Psi & T[i \mapsto \langle \bullet i, \emptyset, \{\circ i\}\rangle, \circ i \mapsto E]; \\
 & & & E'[\circ i/x] \rhd \bullet i; s \\
 & & & \text{if } i \notin \Psi \\
\textit{(SPEC1)} & T; (\mathbf{let}\ x = E\ \mathbf{in}\ E') \rhd i; s & \longrightarrow_\Psi & T[i \mapsto \langle \bullet i, \emptyset, \{\circ i\}\rangle]; E \rhd \circ i; \\
 & & & ((E'[\circ i/x] \rhd \bullet i) : s) \\
 & & & \text{if } i \in \Psi \\
\textit{(SPEC2)} & T; \triangledown i'; (E \rhd i : s) & \longrightarrow_\Psi & T; E \rhd i; s
\end{array}
$$

Figure 5.17: Operational semantics for costed evaluation

represent dependent computations that have been performed. Thunks represent computations that have been put off until later, and suspensions represent areas of the cost graph that have been partially explored.

The current command c says what the program is doing currently, and the stack s records parent nodes whose edges have not yet been completely explored. In the illustrations given in this section, solid circles represent computations that have been performed and dashed circles represent computations that have not yet been performed. As a program evaluates, it converts dashed circles to solid circles.

Evaluation Commands

Evaluation commands explore new nodes, and do the described work. The evaluation command $E \rhd i$ does the work corresponding to the computation i and records that it has done this by adding a closure to T. The evaluation command will then proceed to explore any children that the node has. If there are several children, then it will explore the leftmost child, and add a stack frame to s to remind it to explore the other children later. This is illustrated by Figure 5.19, in which rule *(APP1)* explores its function evaluation child.

$$
\begin{array}{rrcl}
\textit{(!EXP)} & T; E \rhd i; s & \longrightarrow_{\Psi} & T[i \mapsto E]; \circledcirc i; s \\
\textit{(!SPEC)} & T; \circledcirc i; E \rhd j : s & \longrightarrow_{\Psi} & T; E \rhd j; s \\
\textit{(!ABORT)} & T; \circledcirc i; (l, j) : s & \longrightarrow_{\Psi} & T[j \mapsto i \angle l]; \circledcirc j; s \\
\textit{(!RESUME)} & T[j \mapsto i \angle l]; \circledcirc j; s & \longrightarrow_{\Psi} & T; \circledcirc i; (l, j) : s
\end{array}
$$

Figure 5.18: Operational semantics for costed abortion

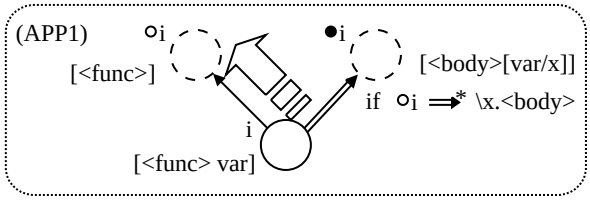


Figure 5.19: Evaluation commands walk over the cost graph, doing work

Return Commands

Return commands return to a partially explored node stored on the stack and explore its unexplored children. This is illustrated in Figure 5.20 in which rule *(APP2)* in which the program returns to a function application frame, and evaluates the body of the function.

Thunks

Thunks represent parts of the cost graph that the virtual machine has chosen to not explore until later. This is illustrated by Figure 5.21 in which rule *(LAZY)* creates a thunk to represent the fact that it has not explored the area of the cost graph representing the right hand side of the $\mathbf{let}$.

Demand Commands

Demand commands attempt to find a value for a closure. They follow value edges until they find a value or a previously unexplored part of the cost graph (a thunk). If a previously unexplored area is found, then they used an evaluation command to explore it. This is illustrated by Figure 5.22.

5.5.3 Work Done by a Program

We consider the work done by a program to be the evaluation transitions it has performed; return transitions and demand transitions are considered to be free. As we saw in Section 5.5.2, every evaluation transition performed is recorded in the computation trace T as either a value closure or a dependent closure. We can thus define the function $workDone$ which maps a state of the

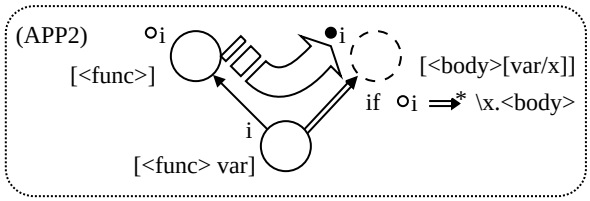


Figure 5.20: Return commands return to a previously unexplored edge

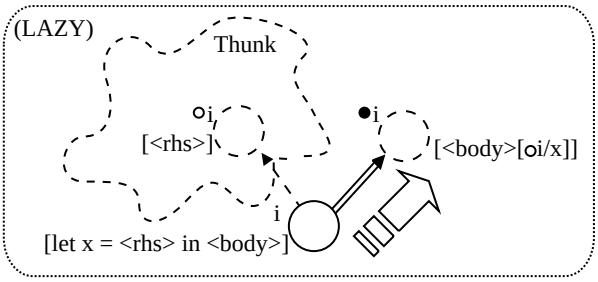


Figure 5.21: Thunks represent areas of the cost graph that the virtual machine chose to not explore until later

operational semantics to the set of computations that it has performed so far:

$$\begin{array}{rcl} workDone & : & Trace \rightarrow Work \\ workDone(T) & = & \{i \mid \quad (i \mapsto (\!|V|\!)) \in T \quad \vee \quad (i \mapsto \langle i', d, e\rangle) \in T\} \end{array}$$

In Section 5.6.5 we prove that the work done so far, as described by $workDone$, will be a subset of that which can take place during a complete program evaluation, as predicted by $programWork$ (Section 5.2.5).

5.6 Denotational Meanings of Operational States

We can give a denotational meaning to states of the operational semantics. The meaning of a state is the complete cost view for the program that the operational semantics is evaluating. This cost view will define a superset of the computations described in the computation trace.

In Section 5.6.4, we prove that this meaning is preserved by $\longrightarrow$ transitions. This demonstrates that our denotational semantics accurately models the cost graph explored by our operational semantics.

In Section 5.6.5 we will use this result to prove that the work done by a program executed using strategy Ψ will always be a subset of the work described by $programWork$ for that strategy. This demonstrates that our denotational semantics correctly predicts the cost of evaluating a program using a given strategy, as defined by the operational semantics.

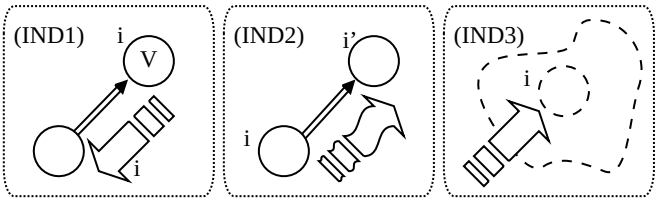


Figure 5.22: Demand commands follow value edges until they find either a value, or a previously unexplored part of the cost graph

5.6.1 Costed Meaning of a Runtime Value:

The operational semantics uses substitutions to replace variables with names. We handle this by treating computation names like local variables, and extending β to map all names onto themselves:

$$\mathcal{E}[\![i']\!]_\beta\ i\ \gamma = (i \mapsto (i', \emptyset, \emptyset))$$

5.6.2 Costed Meaning of a Stack:

We define a semantic function $\mathcal{S}[\![-]\!]$ that gives a meaning to a stack, given the complete cost view γ and the name i of the computation that produced the return value:

$$\mathcal{S}[\![-]\!] \ : \ \mathit{Stack} \to \mathit{CV} \to \mathit{Name} \to \mathit{CV}$$

$$\begin{aligned}
\mathcal{S}[\![(E \rhd j) : s]\!]_\gamma\ i &= \mathcal{E}[\![E]\!]_\emptyset\ j\ \gamma \sqcup \mathcal{S}[\![s]\!]_\gamma\ j \\
\mathcal{S}[\![(@\ k, j) : s]\!]_\gamma\ i &= \lfloor i \rfloor_\gamma\ k\ j\ \gamma \sqcup \mathcal{S}[\![s]\!]_\gamma\ j \\
\mathcal{S}[\![(n\otimes, j) : s]\!]_\gamma\ i &= (\mathcal{O}[\![\otimes]\!]\ j\ n\ \lfloor i \rfloor_\gamma) \sqcup \mathcal{S}[\![s]\!]_\gamma\ j \\
\mathcal{S}[\![(\otimes k, j) : s]\!]_\gamma\ i &= (\mathcal{O}[\![\otimes]\!]\ j\ \lfloor i \rfloor_\gamma\ \lfloor k \rfloor_\gamma) \sqcup \mathcal{S}[\![s]\!]_\gamma\ j \\
\mathcal{S}[\![[\,]]\!]_\gamma\ i &= ()
\end{aligned}$$

The cost view for a stack is the least upper bound of the cost views for all evaluations defined on the stack. This least upper bound is guaranteed to exist for any stack that can be produced during Optimistic Evaluation.

5.6.3 Costed Meaning of a Computation Trace:

We define a semantic function $\mathcal{T}[\![-]\!]$ that gives a meaning to a computation trace:

$$\mathcal{T}[\![-]\!] \ : \ \mathit{Trace} \to \mathit{CV} \to \mathit{CV}$$

$$\mathcal{T}[\![i_1 \mapsto K_1, \ldots, i_n \mapsto K_n]\!]_\gamma \ = \ \mathcal{E}[\![K_1]\!]_\emptyset\ i_1\ \gamma \sqcup \ \ldots \ \sqcup \mathcal{E}[\![K_n]\!]_\emptyset\ i_n\ \gamma$$

The cost view for a computation trace is the least upper bound of the cost views for all of the closures in the trace. The semantic function $\mathcal{E}[\![-]\!]$ is extended to give meanings to indirections,

value closures, and stack suspension closures as follows:

$$\begin{aligned}
\mathcal{E}[\![(j,u)]\!]_\beta\ i\ \gamma &= (i \mapsto (j,u)) \\
\mathcal{E}[\![(\!|\lambda x.E|\!)]\!]_\beta\ i\ \gamma &= \mathcal{E}[\![\lambda x.E]\!]_\beta\ i\ \gamma \\
\mathcal{E}[\![j \angle f]\!]_\beta\ i\ \gamma &= \mathcal{S}[\![(f,i) : []]\!]_\gamma\ j
\end{aligned}$$

Thunks are expressions, and so are dealt with by the rules given in Section 5.3.

5.6.4 Costed Meaning of a State:

We define a semantic function $\mathcal{M}[\![-]\!]$ that gives meanings to program states:

$$\mathcal{M}[\![-]\!] \quad : \quad State \to CV$$

$$\begin{aligned}
\mathcal{M}[\![T; E \rhd i; s]\!] &= \mu\gamma.\mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ i\ \gamma \\
\mathcal{M}[\![T; \circledcirc i, s]\!] &= \mu\gamma.\mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \\
\mathcal{M}[\![T; \triangledown i, s]\!] &= \mu\gamma.\mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i)
\end{aligned}$$

The complete cost view for a state is the least upper bound of the cost views for the computation trace, the stack, and the current expression. This least upper bound is guaranteed to exist for any state that can be produced during Optimistic Evaluation.

Theorem 5.6.1 (Soundness)
The meaning of a state as defined by $\mathcal{M}[\![-]\!]$, is the complete cost view for the program that the virtual machine is evaluating.

$$\emptyset; E; [\,] \longrightarrow^* T; c; s \quad \Rightarrow \quad \mathcal{M}[\![T; c; s]\!] = \mathcal{P}[\![E]\!]$$

We can easily observe that $\mathcal{M}[\![\emptyset; E; [\,]]\!] = \mathcal{P}[\![E]\!]$, so this amounts to proving that the meaning of a state is preserved by the transition rules for '$\longrightarrow$':

$$T; c; s \longrightarrow T'; c'; s' \quad \Rightarrow \quad \mathcal{M}[\![T; c; s]\!] = \mathcal{M}[\![T'; c'; s']\!]$$

A proof of this is given in Appendix B.

5.6.5 Work that Will be Done by a State

We can define a function $pendingWork$ that maps a program state to the work that the program has yet to do, but which it will do before it finishes. $pendingWork$ takes as its arguments a

program state, and the strategy that is being used to evaluate the program:

$$\begin{array}{rcl} pendingWork & : & State \rightarrow Strategy \rightarrow Work \\ pendingWork(T; c; s, \Psi) & = & \mathcal{W}\{\!\{\mathcal{C}\{\!\{c\}\!\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\mathcal{M}[\![T;c;s]\!]} \end{array}$$

This definition makes use of the functions $\mathcal{C}\{\!\{-\}\!\}$ and $\mathcal{S}\{\!\{-\}\!\}$ that find the *pending computations* of a command and a stack respectively. The pending computations are those computations that the program is planning to produce values for in the future. $\mathcal{C}\{\!\{-\}\!\}$ is defined as follows:

$$\begin{array}{rcl} \mathcal{C}\{\!\{E \rhd i\}\!\} & = & \{i\} \\ \mathcal{C}\{\!\{\circledcirc i\}\!\} & = & \{i\} \\ \mathcal{C}\{\!\{\triangledown i\}\!\} & = & \emptyset \end{array}$$

For convenience, we restrict ourselves to a subset of our language in which there are no primitive integer operations.[7] $\mathcal{S}\{\!\{-\}\!\}$ is thus defined as follows:

$$\begin{array}{rcl} \mathcal{S}\{\!\{E \rhd i : s\}\!\} & = & \{i\} \cup \mathcal{S}\{\!\{s\}\!\} \\ \mathcal{S}\{\!\{(@k, i) : s\}\!\} & = & \{i\} \cup \mathcal{S}\{\!\{s\}\!\} \\ \mathcal{S}\{\!\{[]\}\!\} & = & \emptyset \end{array}$$

Theorem 5.6.2 (Correct Work Done)
If the definitions given in this Chapter are correct, then we would like it to be the case that, for any state that can arise in the operational semantics, the work done so far, unioned with the pending work is a subset of the work predicted by $programWork$:

$$\begin{array}{l} \emptyset; E \rhd \epsilon; [\,] \;\longrightarrow^*\; T; c; s \;\Rightarrow \\ \quad workDone(T) \;\cup\; pendingWork(T; c; s, \Psi) \;\subseteq\; programWork(\mathcal{M}[\![T; c; s]\!], \Psi) \end{array}$$

In Appendix C we prove that this theorem does indeed hold. A corollary of this theorem is that the work done by a completed state is a subset of the work predicted by the cost graph:

$$\begin{array}{l} \emptyset; E \rhd \epsilon; [\,] \;\longrightarrow^*\; T; \triangledown i; [\,] \;\Rightarrow \\ \quad workDone(T) \;\subseteq\; programWork(\mathcal{M}[\![T; c; s]\!], \Psi) \end{array}$$

The relation is a subset relation rather than equality because abortion can cause a program to not do work that it was planning to do. We would like to be able to prove that this relation will be an equality if no abortion transitions are applied. Unfortunately, we have not yet been able to produce a proof of this property for the *(DEM2)* rule.[8]

[7] Primitive integer operations complicate the proof for *(!ABORT)* and *(RESUME)* in Appendix C.
[8] We think we *almost* have a proof, but it is not yet in a good enough state for us to publish it.

CHAPTER 6

Deriving an Online Profiler

In this chapter, we explain how the cost model described in Chapter 5 can be used to justify and verify the design of an online profiler for Optimistic Evaluation.

- We start, in Section 6.1, by categorising the work done by a program. We give definitions of *wasted work* and *saved work*, allowing us to formally state how much of the work done by a program execution was unnecessary, and how much work was done *for free*. From these concepts, we are able to develop the concept of *goodness*—a measure of the performance difference between the current evaluation strategy and Lazy Evaluation.

- We continue, in Section 6.2, by formalising the concept of *blame* given in Section 3.3.4.

- In Section 6.3, we explain how the concept of blame can be used as the basis for an online profiler which bounds the worst case performance of Optimistic Evaluation relative to Lazy Evaluation.

- Finally, in Section 6.4, we show how we can reduce the overhead of online profiling by only profiling a small proportion of ventures.

This chapter aims to act as a bridge between the formal cost model of Chapter 5 and the low-level implementation of Chapter 9. This chapter is thus considerably less mathematically rigorous than Chapter 5, but considerably more mathematically rigorous than Chapter 9.

6.1 Categorising Work

Not all work is equal.

Some of the work done by a program may be *wasted*—meaning that it would not have been done by Lazy Evaluation. Optimistic Evaluation will waste work whenever it speculatively evaluates an expression that would not be evaluated under Lazy Evaluation.

Similarly, some of the work done by a program may be *saved*—meaning that Optimistic Evaluation has managed to perform the work without incurring the costs that Lazy Evaluation would have incurred. When Optimistic Evaluation speculates the right hand side of a **let**, it saves the work done to create a thunk. If work has been done, and has not been saved, then we say the work has been *done-at-a-cost*.

The *goodness* of a program state is the difference between the amount of work that was wasted and the amount of work that was saved. Equivalently, it is the difference between the amount of time that Optimistic Evaluation has taken, and the amount of time that would be taken by Lazy Evaluation to do the same amount of useful work. If the goodness is positive, then the current evaluation strategy is outperforming Lazy Evaluation, while negative goodness indicates that the current evaluation strategy is doing worse than Lazy Evaluation.

The aims of the online profiler can be expressed in terms of goodness:

1. It imposes a lower bound on the worst case goodness of a program state, thus bounding the worst case performance of Optimistic Evaluation relative to Lazy Evaluation.

2. Given the first constraint, it attempts to maximise average goodness and thus maximise average performance.

6.1.1 Wasted Work

The *wasted work* of a program state is the work that has been done so far, but which would not be performed at any point by Lazy Evaluation of the same program. We can formally define the wasted work of a program state by taking the set of all computations performed by the program execution so far, and subtracting all computations that would be performed by Lazy Evaluation of the program:

$$\begin{aligned} wastedWork &: State \rightarrow Work \\ wastedWork(T; c; s) &= workDone(T) \setminus lazyWork(T; c; s) \end{aligned}$$

We illustrate this definition graphically in Figure 6.1.

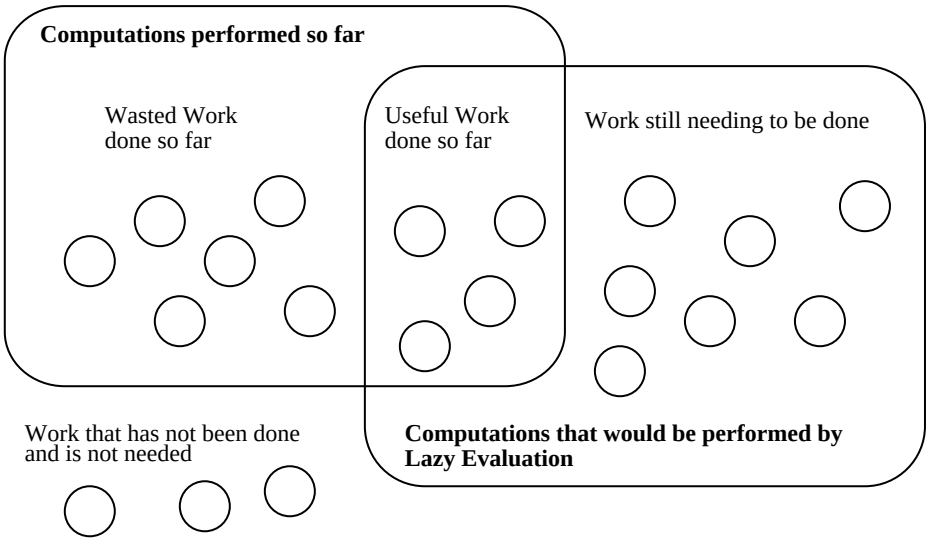


Figure 6.1: A computation is wasted if it would not have been performed by Lazy Evaluation

We define $lazyWork$, the work done by Lazy Evaluation, using the $programWork$ function from Section 5.2.5:

$$\begin{aligned} lazyWork &: State \rightarrow Work \\ lazyWork(T; c; s) &= programWork(\mathcal{M}[\![T; c; s]\!], \emptyset) \end{aligned}$$

6.1.2 Saved Work

If a program is evaluated to completion, then the work done by Optimistic Evaluation will always be a superset of the work done by Lazy Evaluation. However this does not mean that Optimistic Evaluation will always perform worse than Lazy Evaluation. Optimistic Evaluation may have been able to do some computations *for free* that would have taken time under Lazy Evaluation. We refer to the set of computations that have been performed for free as the *saved work*.

In our simplified model, saved work arises when a **let** expression is speculated (using rule *(SPEC1)*). By speculating a **let** expression, Optimistic Evaluation avoids the cost of building a thunk and so we consider it to have saved the computation for that **let** evaluation. With reference to the definitions of Section 5.5.3, we can define the saved work of a program to be the intersection of the work done so far with the set Ψ of all computations that will be speculated if encountered.

$$\begin{aligned} savedWork &: State \rightarrow Strategy \rightarrow Work \\ savedWork(T; c; s)\ \Psi &= workDone(T)\ \cap\ \Psi \end{aligned}$$

Operationally, we can obtain the set of saved work by adding a saved work field F to states. F contains the names of all computations that have been saved so far. The F field is ignored by

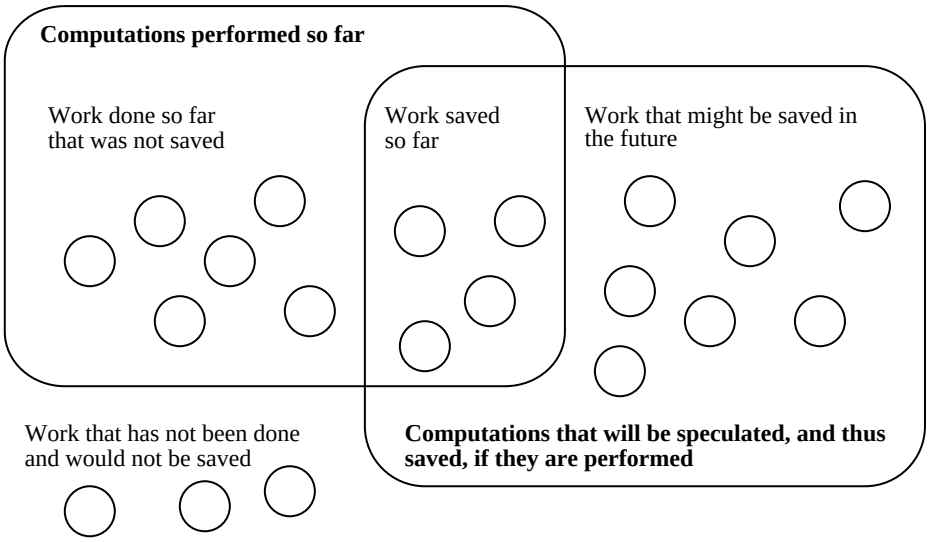


Figure 6.2: A computation is saved if has been speculated

all rules except *(SPEC1)*, which adds its computation name to the set F:

$$(SPEC1) \quad T; (\mathbf{let}\ x = E\ \mathbf{in}\ E') \rhd i; s; F \quad \longrightarrow_{\Psi} \\ T[i \mapsto \langle \bullet i, \emptyset \rangle]; E \rhd \circ i; (E'[\circ i/x] \rhd \bullet i) : s; F \cup \{i\} \\ \text{if } i \in \Psi$$

The saved work of such an extended state is simply its F field. We can observe that this is equivalent to the previous definition.

6.1.3 Goodness

The *goodness* of a program state is a measure of how well Optimistic Evaluation is performing relative to Lazy Evaluation. It is defined to be the difference between the amount of work that the program has done-at-a-cost[1]and the amount of work that would be done-at-a-cost by Lazy Evaluation to do the same amount of useful work. This is the amount of work that has been saved, minus the amount of work that has been wasted:

$$\begin{aligned} goodness &: State \to Strategy \to \mathbb{Z} \\ goodness(T; c; s; \Psi) &= \left| savedWork(T; c; s)\ \Psi \right| - \left| wastedWork(T; c; s) \right| \end{aligned}$$

To understand why this definition makes sense, it is helpful to consider the goodness of an individual computation. From the definitions given in the previous sections, we can see that every computation performed by the program is either wasted or not wasted, and either saved or not saved (Figure 6.3):

[1]Recall that a computation is considered to be *done-at-a-cost* if it has been done and its cost was not saved.

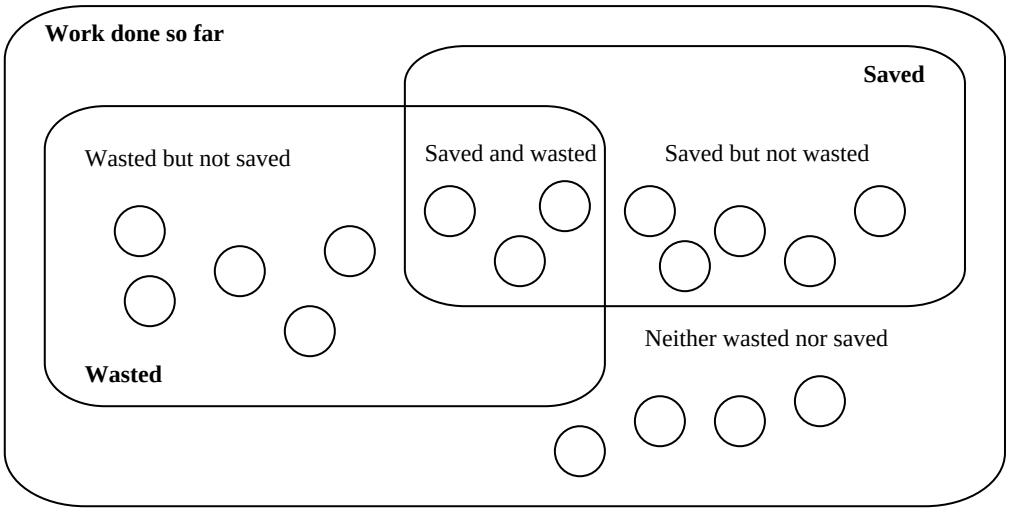


Figure 6.3: Each computation done so far can be wasted, saved, both, or neither

Wasted but not saved: This computation was not needed, and took time to perform. The goodness of the computation is thus -1.

Saved but not wasted: This computation was done for free, but would have taken time under Lazy Evaluation. The goodness of the computation is thus 1.

Wasted and saved: This computation was not needed, but took no time to perform. The goodness of the computation is thus 0.

Neither wasted nor saved: This computation took one unit of time, and would have taken one unit of time under Lazy Evaluation. The goodness of the computation is thus 0.

We can see that saved and wasted computations cancel each other out, leading to the definition of goodness given above.

6.1.4 Using Goodness

By calculating goodness at runtime, an online profiler can keep track of how well Optimistic Evaluation is doing. If the goodness is negative, then Optimistic Evaluation is wasting more work that it is saving, and so the profiler should decrease the level of speculation in an attempt to reduce the amount of waste taking place. If the goodness is positive, then speculation is saving more work than it is wasting, and so the profiler can increase the level of speculation in an attempt to save more work. If the goodness falls below a user-specified cutoff point then evaluation will be made entirely lazy, ensuring that the goodness cannot decrease any further. A possible relationship between goodness and speculation level is illustrated by Figure 6.4.

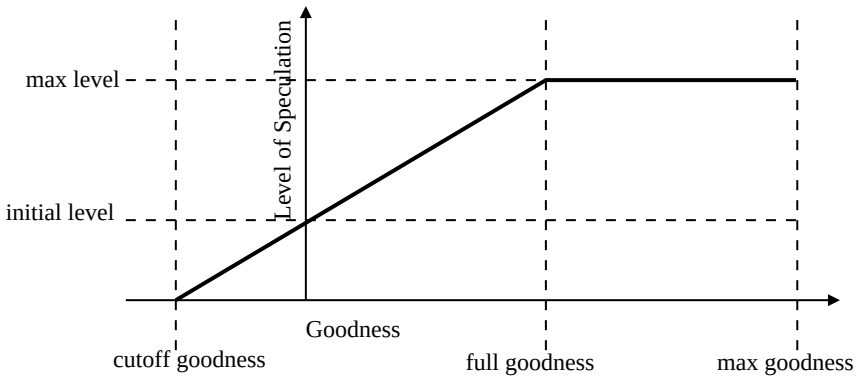


Figure 6.4: The level of speculation depends on the current estimate of goodness, ensuring that goodness should never fall below a pre-determined cutoff point.

6.1.5 Underestimating Goodness

It is not necessary for a profiler to know the exact goodness of a program state. All that is required is that the profiler is able to produce a safe underestimate of goodness. If the goodness is underestimated, then the profiler will be overly cautious and evaluate the program more lazily than necessary. In the worst case, Optimistic Evaluation will decide that speculation is entirely counterproductive and will revert to Lazy Evaluation. The online profiler is only required to ensure that the program does not run significantly slower than it would under Lazy Evaluation. It is not required to ensure that performance is within a particular bound of the optimal evaluation strategy[2]. Thus, while an overestimate of goodness would be unsafe, an underestimate is acceptable.

Although it is acceptable for the profiler to underestimate goodness, it is desirable for it to underestimate goodness by as little as possible, so as to avoid being unnecessarily cautious.

6.2 Blame

If we discover that some work was wasted, it is useful to know *why* the wasted work was done. In this section, we present a semantics in which every wasted computation is *blamed* on a speculation—thus formalising the concept of blame introduced in Section 3.3.3. We go on, in Section 6.2.5 to show that blame can be used as the basis for an online profiler.

6.2.1 An Informal Overview of Blame

Every computation performed by a program is blamed on either the root venture, or exactly one speculation. If a computation is blamed on the root venture, then that means that the

[2]By which we mean the optimal blend of lazy and strict evaluation. The term ‘Optimal Evaluation’ is more commonly used to refer to something else—as we discuss in Section 13.9.2.

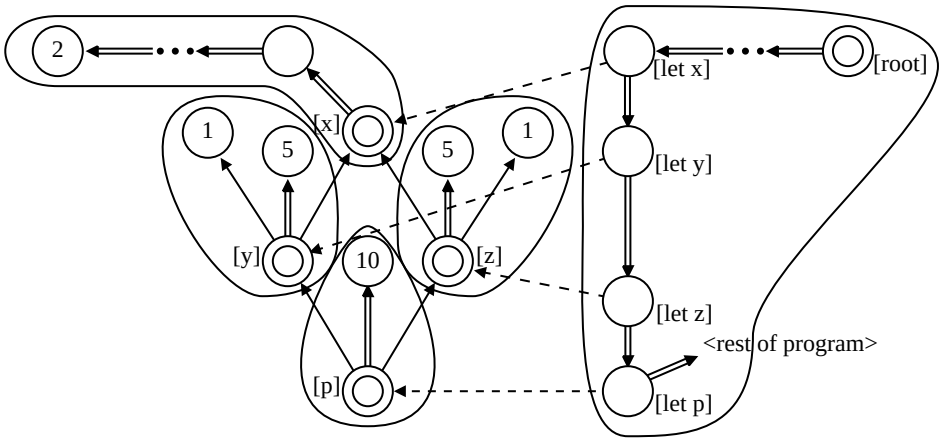


Figure 6.5: All computations are part of the local work for exactly one venture

computation was definitely not wasted. If a computation is blamed on a speculation, then this means that the computation may have been wasted, and the profiler has chosen to blame the work on that speculation. The sum of all work blamed on speculations is guaranteed to be an overestimate of the wasted work of the program state.

Our profiler passes blame between ventures as the program runs. If a venture x is found to be needed by a venture y then any blame that had been attributed to x will be passed to y. It is thus the case that a speculation can only be blamed for work if it is not yet known to be needed.[3]

6.2.2 Ventures

In terms of a cost graph, a venture is a computation that produces a value for the right hand side of a $\mathbf{let}$. A venture is a speculation if was performed before it was known to be needed. The local work of a venture is the work that can be reached directly from that venture, without having to go through any other ventures. We say that a venture i demands a venture i' if any of the computations in the local work of i demand i'.

These concepts are illustrated in Figure 6.5, which extends the cost graph of Section 5.2. In this figure, each venture is marked with a circle and a loop is drawn around the local work for each venture.

6.2.3 Relating Blame to a Cost Graph

We can illustrate the allocation of blame to ventures using a *blame forest.* A *blame forest* consists of a number of disconnected *blame trees*, each of which is rooted at either a speculation or the root venture. If a speculation has no edges linking to it then its *blame* is the number of computations reachable from it; otherwise its blame is zero.

[3]This is the reason why a thunk venture cannot be blamed for work; a thunk venture is needed by the venture that demanded the value of the thunk.

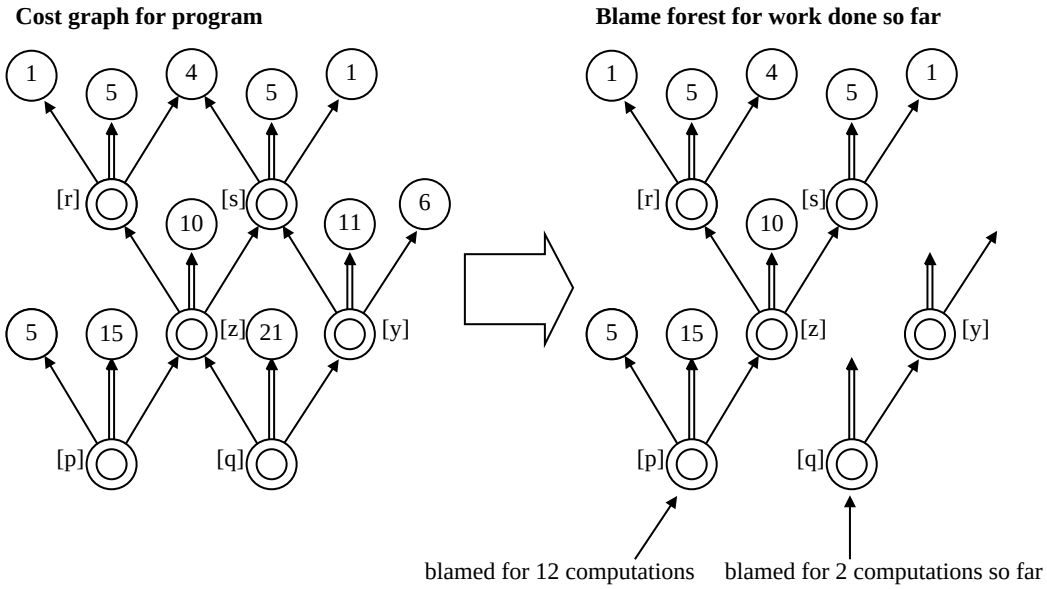


Figure 6.6: A blame forest contains a subset of the nodes and edges of the cost graph

The computations and edges in a blame forest are a subset of those for the cost graph of the program. The computations are those that have been done so far, while the edges are a subset of those in the cost graph, chosen such that no node has more than one node linking to it. As a program executes, we can maintain a blame forest describing the profiler's current allocation of blame to speculations. Whenever a computation is done, it is added to the blame forest, together with as many of its value and demand edges as can be added without causing any node to have more than one edge linking to it.

In Figure 6.6 we illustrate a typical blame forest, together with its corresponding cost graph. We can see that every node and edge in the blame forest is also present in the cost graph. In Figure 6.7 we illustrate the effect of adding a new computation node to a blame forest. In this example the new node depends on the root node, s, of a blame tree, causing that blame tree to be subsumed by the blame tree containing the new node. Operationally, the blame attributed to s is passed to p.

The work blamed on speculations is a superset of the $wastedWork$ of the program state. This follows from the fact that every computation done so far is blamed on either a speculation or the root venture, and the fact that a computation can only be blamed on the root venture if it is reachable from the root venture by value and demand links. If the profiler also has an accurate measure of saved work then it can use the blame forest to produce an underestimate of the goodness of the current state.

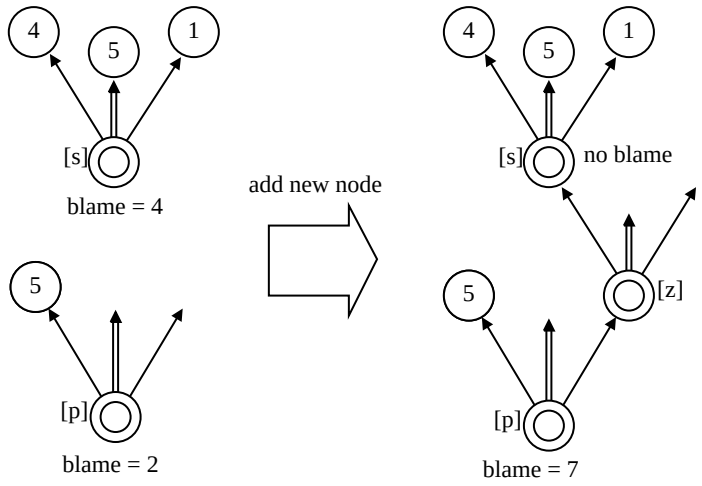


Figure 6.7: When a new node is added to the blame forest, it may cause one blame tree to be subsumed by another one

6.2.4 Per Let Goodness

Our main motivation for blaming computations on ventures is that it allows us to assign a goodness to each individual $\mathbf{let}$ in the program. The goodness of a $\mathbf{let}$ is the amount of work saved by the $\mathbf{let}$, minus the work blamed on ventures spawned by that $\mathbf{let}$. We define a *GoodnessMap*, Π, to be a finite, partial, mapping from $\mathbf{let}$ identifiers to integer goodness values.

The sum of all goodness in the goodness map is guaranteed to be an underestimate of the goodness of the program state.

6.2.5 An Operational Semantics for Blame

Figure 6.8 gives the rules for an operational semantics that keeps track of blame. This semantics is based on the low-level semantics of Section 4.2 and so maintains a heap Γ rather than a computation trace T. States are now of the following form:

$$\Gamma; c; s; B; \Pi$$

where:

- Γ is a heap, mapping heap identifiers to closures, as before. We extend the type of closures to include *costed indirections*. A costed indirection holds the result of a speculation that has work blamed on it. If a blame forest is drawn for the program state, then each blame tree corresponds to a costed indirection in the heap.

  Costed indirections are written as:

  $$B\langle\alpha\rangle^x$$

Evaluate a value constant:

(VAL) $\Gamma; V; s; B; \Pi \longrightarrow \Gamma[\alpha \mapsto V]; \triangledown\alpha; s; B+1; \Pi$ where α is fresh

Demand the value of a closure:

(VAR) $\Gamma; \alpha; s; B; \Pi \longrightarrow \Gamma; \circledcirc\alpha; s; B+1; \Pi$

(DEM1) $\Gamma[\alpha \mapsto (\!|V|\!)]; \circledcirc\alpha; s; B; \Pi \longrightarrow \Gamma[\alpha \mapsto (\!|V|\!)]; \triangledown\alpha; s; B; \Pi$

(DEM2) $\Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha; s; B; \Pi \longrightarrow \Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha'; s; B; \Pi$

(DEM3) $\Gamma[\alpha \mapsto E]; \circledcirc\alpha; s; B; \Pi \longrightarrow \Gamma; E; (\#\alpha : s); B; \Pi$

(UPD) $\Gamma; \triangledown\alpha; (\#\alpha' : s); B; \Pi \longrightarrow \Gamma[\alpha' \mapsto \langle\alpha\rangle]; \triangledown\alpha; B; \Pi$

Function Application:

(APP1) $\Gamma; E\ \alpha; s; B; \Pi \longrightarrow \Gamma; E; (@\alpha : s); B+1; \Pi$

(APP2) $\Gamma[\alpha \mapsto (\!|\lambda x.E|\!)]; \triangledown\alpha; (@\alpha' : s); B; \Pi \longrightarrow \Gamma[\alpha \mapsto (\!|\lambda x.E|\!)]; E[\alpha'/x]; s; B; \Pi$

Lazy Evaluation of a `let`:

(LAZY) $\Gamma; (\mathbf{let}\ x = E\ \mathbf{in}\ E'); s; B; \Pi \longrightarrow \Gamma[\alpha \mapsto E]; E'[\alpha/x]; s; B+1; \Pi$ if $\mathit{goodToLim}(\Pi(x)) \leq \mathit{specDepth}(s)$ where α is fresh

Speculative Evaluation of a `let`:

(SPEC1) $\Gamma; (\mathbf{let}\ x = E\ \mathbf{in}\ E'); s; B; \Pi \longrightarrow \Gamma; E; ((\{x\}E', B+1) : s); 0; \Pi[x \mapsto \Pi(x)+1]$ if $\mathit{goodToLim}(\Pi(x)) > \mathit{specDepth}(s)$

(SPEC2) $\Gamma; \triangledown\alpha; ((\{x\}E, B') : s); B; \Pi \longrightarrow \Gamma[\alpha' \mapsto B\langle\alpha\rangle^x]; E[\alpha'/x]; s; B'; \Pi[x \mapsto \Pi(x) - B]$ where α' is fresh

Demand the value of a costed indirection:

(CST) $\Gamma[\alpha \mapsto B'\langle\alpha'\rangle^x]; \circledcirc\alpha; s; B; \Pi \longrightarrow \Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha'; s; B+B'; \Pi[x \mapsto \Pi(x)+B']$

Figure 6.8: Operational semantics with blame

where B is the amount of work blamed on the speculation, α is a heap reference for the result of the speculation, and x is the identifier for the $\mathbf{let}$ that spawned the speculation.

- s is a stack. Stack frames are similar to those used in the low-level semantics of Section 4.2; however speculative return frames are extended to contain the blame accumulated so far by the enclosing speculation. Stack frames are thus as follows:

Stack	s	::=	$[\,]$	Empty stack
		$\mid$	$(\{x\}E, B) : s$	Speculative return
		$\mid$	$\#\alpha : s$	Thunk update
		$\mid$	$@\alpha : s$	Lazy return

- B an integer count of the amount of work that has been blamed on the current speculation so far.

- Π is a goodness map, recording the goodness that is currently assigned to each $\mathbf{let}$ in the program.

  The semantics ensures that $\Pi(x)$ is always the total amount of work saved by ventures spawned by x, minus the total amount of work blamed on speculations spawned by x.

If the current speculation demands the result of a costed indirection in the heap (rule *(CST)*), then the costed indirection is replaced by a simple indirection and its blame is transferred to the current speculation. Referring to the blame forest, this is equivalent to adding an edge linking the current speculation to the blame tree represented by the costed indirection, as illustrated by Figure 6.7.

Whenever the cost semantics of Section 5.5 would add a new node to the computation trace, the semantics of Figure 6.8 will increment the current blame B, blaming the current speculation for the work.

Whenever a speculation starts, rule *(SPEC1)* adds one unit of goodness to the goodness count for the spawning $\mathbf{let}$. This reflects the fact that the venture saved one computation by avoiding building a thunk. As discussed in Section 3.3.5, rule *(SPEC1)* blames the enclosing venture for the work needed to build a thunk, even though a thunk has not been built.

Whenever a speculation completes, rule *(SPEC2)* subtracts B units of goodness from the goodness count for x. This records the fact that B computations have been blamed on a venture spawned by x. If the blame for these computations is later transferred elsewhere, then *(CST)* will add B to the goodness count for x; thus ensuring that the goodness of a $\mathbf{let}$ accurately reflects the blame currently attributed to ventures spawned from that let.

[3] By which we mean the blame tree that was current before the frame was pushed.

6.2.6 Directing Evaluation with Goodness

Rather than using a speculation configuration Σ or an evaluation strategy Ψ to chose when to speculate a $\mathbf{let}$ expression, the semantics of Figure 6.8 instead uses the goodness map Π. We assume the existence of a *goodness weighting function*, $goodToLim$, that maps goodness counts to speculation depth limits. This is equivalent to defining the speculation configuration Σ as follows:

$$\Sigma(x) = goodToLim(\Pi(x))$$

$goodToLim$ can be any function, provided that there is some minimum goodness $MINGOODNESS$ such that $goodToLim$ will be zero for any goodness below $MINGOODNESS$, and provided that there is no goodness for which $goodToLim$ will be greater than $MAXDEPTH$. That is:

$$\forall z. \quad (z \leq MINGOODNESS \quad \Rightarrow \quad goodToLim(z) = 0)$$
$$\forall z. \quad goodToLim(z) \leq MAXDEPTH$$

6.3 Bounding Worst Case Performance

The blame semantics of Section 6.2 ensures that no new speculations will be started for a $\mathbf{let}$ if the goodness of that $\mathbf{let}$ is less than $MINGOODNESS$. However this is not sufficient to impose a lower bound on the goodness that can be attributed to a $\mathbf{let}$: a venture may start while the goodness of a $\mathbf{let}$ is high, and then do a large amount of wasted work, causing a large amount of goodness to be subtracted from the $\mathbf{let}$.

If we are to place a lower bound on the goodness that can be assigned to a $\mathbf{let}$ then we must not only place restrictions on when new speculations can be created, but also place restrictions on the amount of work that can be blamed on the active speculations.

In this section we explain how such a limit can be imposed, and how this allows us to guarantee a bound on the worst case performance of Optimistic Evaluation.

6.3.1 Bounded Speculation with Blame

In Section 4.2.4 we gave a semantics for *bounded speculation*, which ensures that no more than $MAXTIME$ steps of speculative evaluation can be performed before abortion takes place. We can refine this semantics so that it instead places a limit on the amount of blame that can be assigned to active speculations. The rules for this refined semantics are given in Figure 6.9.

The rule *(RUN)* performs an evaluation step only if that evaluation step will not cause more than $MAXBLAME$ units of work to be blamed on active speculations. This rule makes use of

$$\textit{(RUN)}\quad \frac{\Gamma; c; s; B; \Pi \quad \longrightarrow \quad \Gamma'; c'; s'; B'; \Pi'}{\Gamma; c; s; B; \Pi \quad \curvearrowright \quad \Gamma'; c'; s'; B'; \Pi'} \quad \begin{array}{l}\text{if } B' + \mathit{activeBlame}(s') \leq \mathit{MAXBLAME} \\ \text{or } \mathit{specDepth}(s') = 0\end{array}$$

$$\textit{(ABORT)}\quad \frac{\Gamma; c; s; B; \Pi \quad \rightsquigarrow \quad \Gamma'; c'; s'; B'; \Pi'}{\Gamma; c; s; B; \Pi \quad \curvearrowright \quad \Gamma'; c'; s'; B'; \Pi'} \quad \text{if rule } \textit{(RUN)} \text{ could not be applied}$$

Figure 6.9: Operational semantics of blame bounded speculation

a function $\mathit{activeBlame}$ that sums the blame for all active ventures recorded on the stack:

$$\begin{aligned}
\mathit{activeBlame}((\{x\}E, B) : s) &= \begin{cases} 0 & \text{if } \mathit{specDepth}(s) = 0 \\ B + \mathit{activeBlame}(s) & \text{otherwise} \end{cases} \\
\mathit{activeBlame}(\#\alpha : s) &= \mathit{activeBlame}(s) \\
\mathit{activeBlame}(l : s) &= \mathit{activeBlame}(s) \\
\mathit{activeBlame}([]) &= 0
\end{aligned}$$

Recall that the blame field B in a speculative return frame contains the blame accumulated so far for the enclosing venture. We thus do not count the blame attached to the outermost speculative return frame as this will be the blame accumulated so far by the root venture—which is not a speculation.

If rule *(RUN)* cannot be applied, then rule *(ABORT)* will apply abortion transitions until the *(RUN)* rule can be applied again.

It is important that we limit blame rather than evaluation steps or local work. If we limited evaluation steps, then it would be possible for a speculation to accumulate a lot of blame by demanding the results of previous ventures. When the venture completed, this blame would be subtracted from the goodness of the spawning $\mathbf{let}$. It is also important that the side condition is on the blame after the transition rather than the blame before the transition. It is possible for a venture to accumulate a large amount of blame in one step using the *(CST)* rule, thus testing the blame before a transition would allow a venture to accumulate significantly more than $\mathit{MAXBLAME}$ units of blame.

6.3.2 Abortion

The bounded speculation semantics presented in Section 6.3.1 makes use of the abortion relation given in Figure 6.10. The rules in Figure 6.10 are largely the same as those given in Section 4.2.3, but have been extended to keep track of blame.

abort a speculation:

$$\textit{(!EXP)} \quad \Gamma; E; s; B; \Pi \rightsquigarrow \Gamma[\alpha \mapsto E]; \circledcirc\alpha; s; B; \Pi$$

$$\textit{(!RET)} \quad \Gamma; \triangledown\alpha; s; B; \Pi \rightsquigarrow \Gamma; \circledcirc\alpha; s; B; \Pi$$

$$\textit{(!SPEC)} \quad \Gamma; \circledcirc\alpha; (\{x\}E, B') : s; B; \Pi \rightsquigarrow \Gamma[\alpha' \mapsto B\langle\alpha\rangle^x]; E[\alpha''/x]; s; B'; \Pi[x \mapsto \Pi(x) - B - B_{\text{abort}}] \text{ where } \alpha' \text{ is fresh}$$

$$\textit{(!UPD)} \quad \Gamma; \circledcirc\alpha; \#\alpha' : s; B; \Pi \rightsquigarrow \Gamma[\alpha' \mapsto (\alpha)]; \circledcirc\alpha; s; B; \Pi$$

$$\textit{(!ABORT)} \quad \Gamma; \circledcirc\alpha; l : s; B; \Pi \rightsquigarrow \Gamma[\alpha' \mapsto \alpha\angle l]; \circledcirc\alpha'; s; B; \Pi \text{ where } \alpha \text{ is fresh}$$

resume a suspended evaluation:

$$\textit{(RESUME)} \quad \Gamma[\alpha \mapsto \alpha'\angle l]; \circledcirc\alpha; s; B; \Pi \longrightarrow \Gamma; \circledcirc\alpha'; l; \#\alpha : s; B; \Pi$$

Figure 6.10: Operational semantics : abortion with blame

The only rule to have changed significantly is *(!SPEC)*, which blames any work accumulated so far on the **let** that spawned that venture. The abortion system also blames the **let** for B_{abort} units of work, representing the work done to perform the abortion itself.

6.3.3 Worst Case Performance

We can tell from rule *(SPEC)* in the semantics of Section 6.2.5, and the restrictions on the definition of $badToLim$ given in Section 6.3.1, that a new speculation can only be created for a **let** if the goodness for that **let** is less than $MINGOODNESS$.

The minimum goodness that can be attributed to a **let** is thus $MINGOODNESS$ minus the maximum amount of goodness that can be subtracted from that **let** by speculations that were started before the goodness of the **let** reached $MINGOODNESS$.

We saw in Section 6.3.1 that the maximum amount of blame that can be attributed to active speculations is $MAXBLAME$. If every one of these ventures is aborted, then the cost of abortion would be $MAXDEPTH \times B_{\text{abort}}$.[4] It is thus the case that the minimum goodness that can be attributed to any **let** is $MINGOODNESS - MAXBLAME - (MAXDEPTH \times B_{\text{abort}})$.

There are a finite number of **let** expressions in a program. It is thus the case that a bound on the goodness attributable to any one **let** will also give a bound on the total goodness attributed

[4]Recall that $MAXDEPTH$ is the maximum number of speculations that can be active, as defi ned by Section 6.2.6.

to all **let**s. In Section 6.2.3 we showed that this will be an underestimate of the goodness of the program state.

In Section 6.1.3 we argued that the goodness of a program state is an accurate measure of the performance difference between Optimistic Evaluation and Lazy Evaluation. We thus have a bound on the worst case performance of Optimistic Evaluation, relative to Lazy Evaluation.

This is exactly what the profiler is intended to do!

6.3.4 Variants on the Worst Case Bound

The user has quite a lot of control over the worst case performance bound. If $MINGOODNESS$ and $MAXSPEC$ are fixed constants then the minimum goodness will be lower if the number of **let** expressions in the program is increased. This anomaly can be fixed by arranging for $MINGOODNESS$ and $MAXSPEC$ to be inversely proportional to the number of **let** expressions in the program.

$MINGOODNESS$ and $MAXSPEC$ do not have to be constants either. Indeed, it can be useful for $MINGOODNESS$ to be expressed as a proportion of the total runtime of the program, causing the worst case performance to be a fixed percentage slower than Lazy Evaluation, rather than taking a fixed amount of extra time.

6.4 Burst Profiling

It is not practical to profile a program all of the time; the overhead would be too great. In this section we formalise the concept of *burst profiling*, as introduced in Section 3.4. Burst Profiling profiles a program for only a small proportion of its runtime, but still allows goodness to be accurately estimated.

6.4.1 Periods

Burst profiling divides the runtime of a program up into a series of distinct periods. Each period starts at a *boundary point* and lasts until the next boundary point. Any speculations that start during a period are considered to belong to that period. This is illustrated in Figure 6.11 in which speculations are rectangles, and the shading of a speculation indicates the period that it belongs to.

A random selection of periods is profiled. When a period is profiled, all speculations that start during that period will be profiled (Figure 6.12). If every period has a probability p of being profiled, then it follows that every speculation also has a probability p of being profiled. Given that the goodness of the program is the sum of the goodness of the all speculations, it

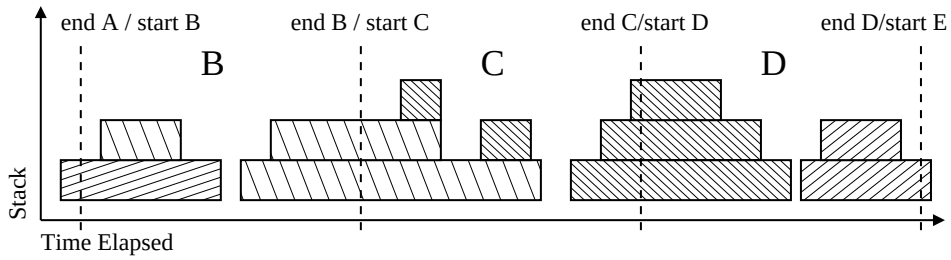


Figure 6.11: Every speculations belongs to exactly one profi le period

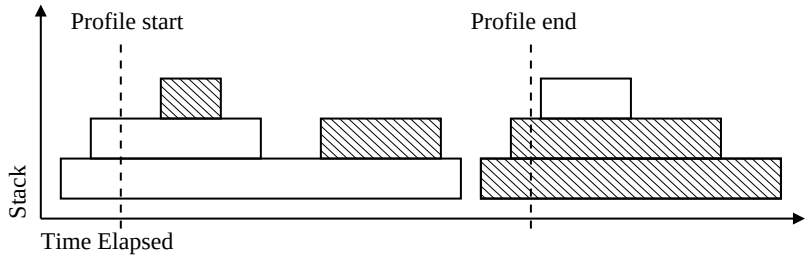


Figure 6.12: Only those speculations that start during a profi led period will be profi led.

follows that the measured goodness should, on average, be p times the goodness that would be measured if profiling was always enabled.

The burst profiler aims to calculate the following:

- An overestimate of the number of computations that took place during profiled periods and were wasted (as defined by Section 6.1.1)
- An exact count of the number of computations that took place during profiled periods and were saved (as defined by Section 6.1.2)

From these, it aims to underestimate the goodness of the profiled part of the program's execution. By dividing this goodness by the number of computations profiled, it is possible to calculate the *relative goodness* of the program. If we assume that profiled computations behave, on average, in the same way as unprofiled computations, then the relative goodness of the profiled computations should be a good estimate of the relative goodness of the program execution. The profiler can thus estimate the goodness of the program execution by multiplying the relative goodness by the total number of computations performed.

6.4.2 Operational Semantics

Figure 6.13 gives the rules for burst profiling. This semantics is a combination of the rules from the blame profiling semantics of Section 6.2 and the unprofiled semantics of Section 4.2. The evaluation relation $\longrightarrow$ is parameterised by a profiling switch p. If p is 'on' then the profiling rules are used, while, if p is 'off' then the unprofiled rules are used.

Evaluate a value constant:

(VAL) $\Gamma; V; s; B; \Pi \longrightarrow_p \Gamma[\alpha \mapsto V]; \triangledown\alpha; s; B+1; \Pi$ where α is fresh

Demand the value of a closure:

(VAR) $\Gamma; \alpha; s; B; \Pi \longrightarrow_p \Gamma; \circledcirc\alpha; s; B+1; \Pi$

(DEM1) $\Gamma[\alpha \mapsto (\!|V|\!)]; \circledcirc\alpha; s; B; \Pi \longrightarrow_p \Gamma[\alpha \mapsto (\!|V|\!)]; \triangledown\alpha; s; B; \Pi$

(DEM2) $\Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha; s; B; \Pi \longrightarrow_p \Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha'; s; B; \Pi$

(DEM3) $\Gamma[\alpha \mapsto E]; \circledcirc\alpha; s; B; \Pi \longrightarrow_p \Gamma; E; \#\alpha : s; B; \Pi$

(UPD) $\Gamma; \triangledown\alpha; \#\alpha' : s; B; \Pi \longrightarrow_p \Gamma[\alpha' \mapsto \langle\alpha\rangle]; \triangledown\alpha; B; \Pi$

Function Application:

(APP1) $\Gamma; E\ \alpha; s; B; \Pi \longrightarrow_p \Gamma; E; @\ \alpha : s; B+1; \Pi$

(APP2) $\Gamma[\alpha \mapsto (\!|\lambda x.E|\!)]; \triangledown\alpha; @\ \alpha' : s; B; \Pi \longrightarrow_p \Gamma[\alpha \mapsto (\!|\lambda x.E|\!)]; E[\alpha'/x]; s; B; \Pi$

Lazy Evaluation of a `let`:

(LAZY) $\Gamma; (\mathbf{let}\ x = E\ \mathbf{in}\ E'); s; B; \Pi \longrightarrow_p \Gamma[\alpha \mapsto E]; E'[\alpha/x]; s; B+1; \Pi$ if $\mathit{goodToLim}(\Pi(x)) \leq \mathit{specDepth}(s)$ where α is fresh

Unprofiled Speculative Evaluation of a `let`:

(SPEC1N) $\Gamma; (\mathbf{let}\ x = E\ \mathbf{in}\ E'); s; B; \Pi \longrightarrow_{\text{off}} \Gamma; E; (\{x\}E' : s); B+1; \Pi$ if $\mathit{goodToLim}(\Pi(x)) > \mathit{specDepth}(s)$

(SPEC2N) $\Gamma; \triangledown\alpha; (\{x\}E : s); B; \Pi \longrightarrow_p \Gamma; E[\alpha/x]; s; B; \Pi$

Profiled Speculative Evaluation of a `let`:

(SPEC1P) $\Gamma; (\mathbf{let}\ x = E\ \mathbf{in}\ E'); s; B; \Pi \longrightarrow_{\text{on}} \Gamma; E; ((\{x\}E', B+1) : s); 0; \Pi[x \mapsto \Pi(x)+1]$ if $\mathit{goodToLim}(\Pi(x)) > \mathit{specDepth}(s)$

(SPEC2P) $\Gamma; \triangledown\alpha; ((\{x\}E, B') : s); B; \Pi \longrightarrow_p \Gamma[\alpha' \mapsto B\langle x\rangle^{x'}]; E[\alpha'/x]; s; B'; \Pi[x \mapsto \Pi(x) - B]$ where α' is fresh

Demand the value of a costed indirection:

(CSTN) $\Gamma[\alpha \mapsto B'\langle\alpha'\rangle^x]; \circledcirc\alpha; s; B; \Pi \longrightarrow_p \Gamma[\alpha \mapsto B'\langle\alpha'\rangle^x]; \circledcirc\alpha'; s; B; \Pi$ if $\neg\mathit{profiled}(s)$

(CSTP) $\Gamma[\alpha \mapsto B'\langle\alpha'\rangle^x]; \circledcirc\alpha; s; B; \Pi \longrightarrow_p \Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha'; s; B+B'; \Pi[x \mapsto \Pi(x)+B']$ if $\mathit{profiled}(s)$

Figure 6.13: Operational semantics for Burst Profi ling

The stack s can contain two types of speculative return frame, one for a speculation that is being profiled and one for a speculation that is not being profiled. These are the speculative return frames from Sections 6.2 and 4.2 respectively:

$$\begin{array}{llll} \text{Stack} & s & ::= & [\,] & \text{empty stack} \\ & & | & (\{x\}E, B) : s & \text{speculative return (profiled)} \\ & & | & \{x\}E : s & \text{speculative return (unprofiled)} \\ & & | & \#\alpha : s & \text{Thunk update} \\ & & | & @\alpha : s & \text{lazy return} \end{array}$$

The rules *(CSTN)* and *(CSTP)* make use of a function $profiled$ that looks at the stack and determines whether the current speculation is being profiled:

$$profiled : Stack \to \mathbb{B}$$

$$\begin{array}{rcl} profiled([\,]) & = & \mathbf{true} \\ profiled(\{x\}E : s) & = & \mathbf{false} \\ profiled((\{x\}E, B) : s) & = & \mathbf{true} \\ profiled(\#\alpha : s) & = & profiled(s) \\ profiled(@\alpha : s) & = & profiled(s) \end{array}$$

The rules behave as follows:

- Most of the rules behave just like they did in the blame profiled semantics of Section 6.2.

  As we explain in Section 9.1.1 the blame counter B is implemented using a counter which increments automatically as the program executes. The virtual machine thus increments the current blame counter B even when the current speculation is not being profiled.

- When a speculation starts, the virtual machine has a choice of two rules to apply. *(SPEC1P)* is used if the current period is profiled, otherwise *(SPEC1N)* is used. Rule *(SPEC1P)* behaves like *(SPEC1)* in the profiled semantics of Section 6.2, while *(SPEC1N)* behaves like *(SPEC1)* in the original unprofiled semantics of Section 4.2.

  If an unprofiled speculation, y, is nested inside a profiled speculation, x, then the x will be blamed for any local work done by the y. This can cause the profiler to overestimate wasted work, but that is allowed.

- When a speculation completes, the virtual machine again has a choice of two rules. If the speculation was profiled, then it ends with *(SPEC2P)*, otherwise it will end with *(SPEC2N)*. Again, these are the rules from Section 6.2 and Section 4.2.

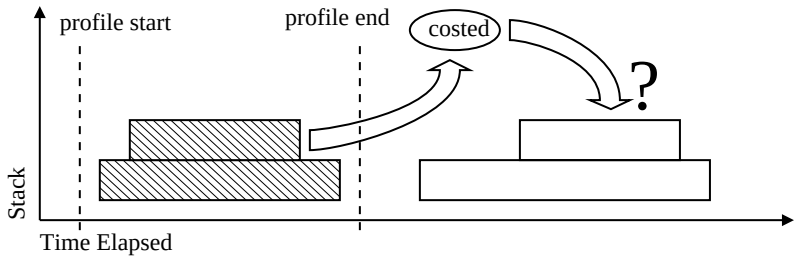


Figure 6.14: If an unprofi led speculation uses a costed indirection then there is nowhere for the cost to be passed to

$$\textit{(CSTN)} \quad \Gamma[\alpha \mapsto B'\langle\alpha'\rangle^x]; \circledcirc\alpha; s; B; \Pi \quad \longrightarrow_p \quad \Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha'; \mathit{chainProf}(B, s); B'; \Pi \quad \text{if } \neg \mathit{profiled}(s)$$

Figure 6.15: Chain profi ling uses this revised version of *(CSTN)*

- To demand a costed indirection, the evaluator applies *(CSTN)* or *(CSTP)*. *(CSTP)* is applied if the current speculation is profiled, otherwise *(CSTN)* is applied. These rules are discussed further in Section 6.4.3.

6.4.3 Profile Chaining

If some speculations are profiled and others are not, what should happen when the result of a profiled speculation is used by an unprofiled speculation. This situation is illustrated by Figure 6.14.

If a profiled speculation, x, demands the result of an unprofiled speculation, y, then we can transfer blame from y to x, as described in Section 6.2.3. However, if x is not profiled, there will be no blame counter for blame to be transferred into. In the semantics of Figure 6.13 we deal with this problem by ignoring any demands made by unprofiled speculations. While this strategy is entirely safe, it turns out (see results in Section 12.3.7) that there are performance advantages to adopting a strategy that we call *chaining*.

If a profiled speculation, x, demands the result of an unprofiled speculation, y then it seems unfair to continue to blame y for wasting its work when the work was needed by x. What we would like to be able to do is to pass this blame to x. Since we cannot pass work to an unprofiled speculation, we need to turn x into a profiled speculation. We can do this by replacing *(CSTN)* with the rule given in Figure 6.15.

This new rule uses a function $\mathit{chainProf}$ that converts the current speculation into a profiled speculation. $\mathit{chainProf}$ is defined as follows:

$$
\begin{array}{rcll}
\mathit{chainProf} & : & \mathbb{N} \to \mathit{Stack} \to \mathit{Stack} & \\
\mathit{chainProf}(B, [\,]) & = & \mathit{error}! & \text{no frame to profile} \\
\mathit{chainProf}(B, \{x\}E : s) & = & (\{x\}E, B) : s & \\
\mathit{chainProf}(B, (\{x\}E, B') : s) & = & \mathit{error}! & \text{already profiled} \\
\mathit{chainProf}(B, \#\alpha : s) & = & \#\alpha : \mathit{chainProf}(B, s) & \\
\mathit{chainProf}(B, @\alpha : s) & = & @\alpha : \mathit{chainProf}(B, s) &
\end{array}
$$

$\mathit{chainProf}$ walks down the stack until it finds the return frame for the current unprofiled speculation. We know that such a stack frame must exist because *(CSTN)* is only applied if $\mathit{profiled}(s)$ is false. When $\mathit{chainProf}$ finds this frame, it converts it into a profiled speculation frame. The current blame B is stored in this frame, causing the blame counter to be restored when the speculation completes.

Since the current speculation is now profiled, *(CSTN)* can now blame it for the work attached to x. Note that we take care to avoid crediting the newly profiled speculation with saving work. This is because the work saved by the speculation did not take place during a profiled period, and so should not be included in the saved work count described in Section 6.4.1.

We describe our implementation of profile chaining in more detail in Section 9.4.1. We discuss its performance impact in Section 12.3.7.

Part III

Implementation

CHAPTER 7

The GHC Execution Model

Optimistic Evaluation is implemented as an extension of the GHC compiler. Thus, in order to explain how Optimistic Evaluation is implemented, it is necessary to first explain how the GHC execution model works and how it relates to the semantic models we have presented. This chapter only gives a very brief overview of the workings of GHC; the real GHC implementation is significantly more complex than that described here. Those wishing to discover more about the workings of GHC are encouraged to read the many papers that have been written about it [PHH+93, PMR99, San95b, PP93, PL91, Pey92, Pey91]. A more detailed tutorial on the compilation of non-strict functional languages in general can be found in [PL92].

In this chapter, we describe the way that GHC works by comparing it to the semantics of Section 4.2, with the parts specific to speculative evaluation removed. This semantics is a very close match to the way that GHC actually works.

- In Section 7.1 we describe the structure of the GHC heap, and explain how it relates to the heap used by the formal semantics of Section 4.2.

- In Section 7.2 we describe the structure of the GHC stack, and explain how it corresponds to the stack used by the formal semantics of Section 4.2.

- In Section 7.3 we describe the code generated by GHC, and explain how it corresponds to the commands and evaluation rules used by the formal semantics of Section 4.2.

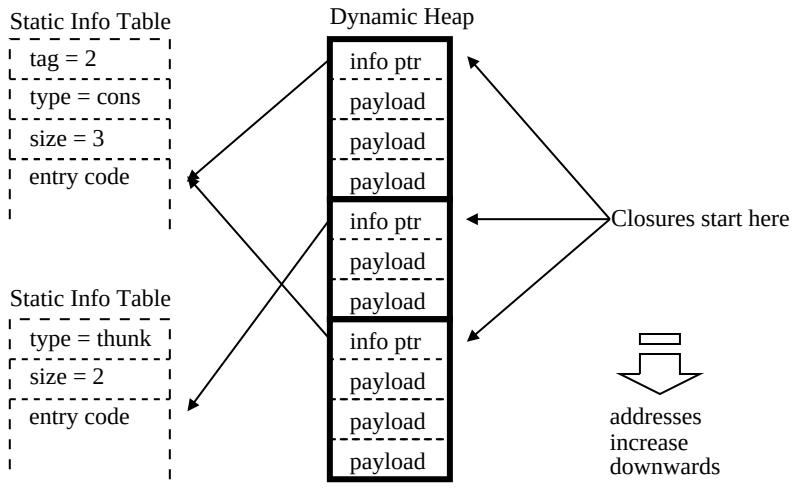


Figure 7.1: Structure of Heap Closures

7.1 The Heap

In the formal semantics of Section 4.2, the heap Γ is a function mapping heap references α to closures K. In GHC, the heap is a continuous block of memory.[1] A closure K is a sequence of words in the heap, and a heap reference α is a pointer to the start of the closure that it references.

Figure 7.1 illustrates a typical GHC heap. In this figure, objects in the dynamic heap (closures) are drawn with bold outlines while static objects are drawn with dotted outlines. All closures in GHC have the same basic form in the heap, irrespective of whether they represent values, thunks or something else; the first word of the closure is the *info pointer*, and the other words are known as the *payload*. The info pointer serves two purposes:

- It points to the end of the *info table*. The info table is a static structure that contains useful information about the closure.
- It points to the beginning of the *entry code*. This is the code that should be executed in order to force the closure to evaluate to a value. If the closure is already a value then this code will be a stub that returns immediately to the caller.

The info table contains a number of fields, including the following:

- **type:** What kind of thing the closure is. E.g. a **function**, **thunk** or **cons***tructed* value.
- **size:** How many words there are in the closure payload.
- **layout:** Information for the garbage collector (we will omit this in most of our diagrams)

It the following subsections, we explain how each closure type described in Section 4.2 is represented in the GHC heap.

[1] Actually, it is divided into sub-blocks, but such details are unimportant.

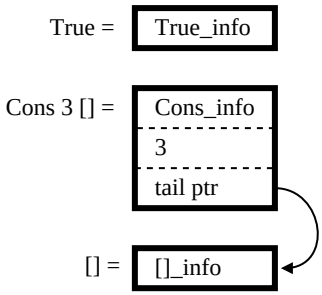


Figure 7.2: Heap Representation of Constructed Values

7.1.1 Constructed Values

$$(\!|C\ \alpha_0 \ldots \alpha_n|\!)$$

Figure 7.2 illustrates the way that GHC represents a constructed value in the heap: the info table describes the static constructor C and the payload is arguments $\alpha_0 \ldots \alpha_n$. The entry code for a constructor will simply return to the caller; there is no need for any work to be done as the constructor is already fully evaluated.

In this figure, and in the other figures used in this chapter, we write a number (e.g. 3) as shorthand for a pointer to a closure representing that number.

7.1.2 Function Closures

$$(\!|\lambda x.E|\!)$$

In the formal semantics, a function closure is a value $\lambda x.E$ that has been formed by substituting heap references for the free variables of a statically defined function.[2]

Figure 7.3 illustrates the way that GHC represents a function closure in the heap: the info table describes the static function, and the closure payload contains the heap references that have been substituted for the free variables of that function. A function closure is a value[3] , and so the entry code will do nothing; however the info table also contains a pointer to the body code for the static function, which can be used to apply the function to arguments.[4]

Function calling conventions are discussed in Section 7.3.2.

[2]The operational semantics would have been a closer match for the real implementation if it had used environments rather than substitutions. We chose not to do this because it made the semantics generally more complicated.

[3]i.e. a weak head normal form

[4]This is a simplification. The real implementation is more complex than this; however the difference is unimportant.

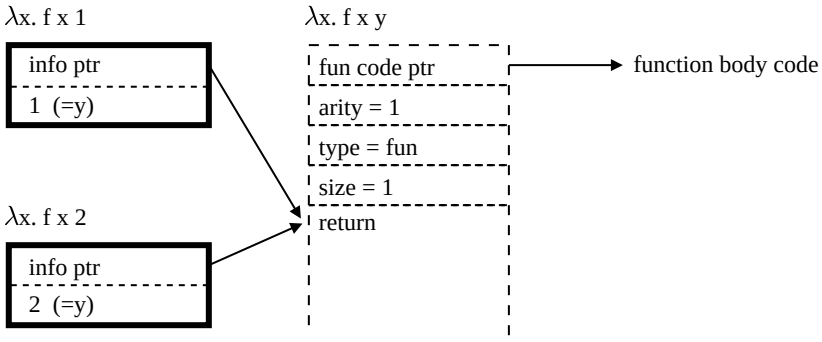


Figure 7.3: Heap Representation of Functions

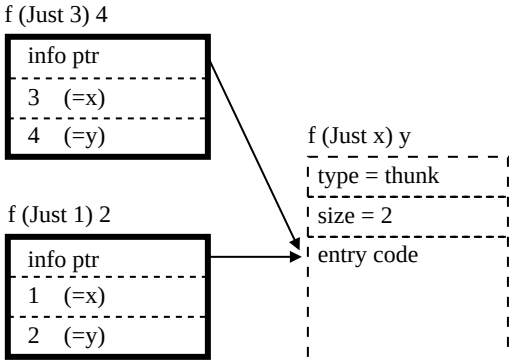


Figure 7.4: Heap Representation of Thunks

7.1.3 Thunks

$$E$$

In the formal semantics, a thunk is an expression E that has been formed by substituting heap references for the free variables of a static expression.

Figure 7.4 illustrates the way that GHC represents thunks in the heap. The info table describes the static expression, while the payload contains the heap references that have been substituted for its free variables. The entry code for a thunk will read the bindings for its free variables from the closure payload and then evaluate the expression, as we describe in Section 7.4.2.

7.1.4 Indirections

$$\langle \alpha \rangle$$

Indirections take the form illustrated in Figure 7.5. The first word points to the standard info table for indirections and the second word is α (the *indirectee* pointer). When a thunk is evaluated to a value, its closure will be overwritten with an indirection to that value.

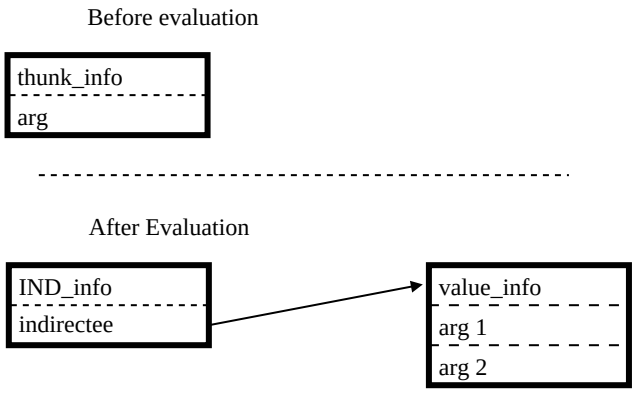


Figure 7.5: Overwriting a thunk with an indirection

7.1.5 Blackholes

GHC has a special kind of closure that it calls a *blackhole*. A blackhole represents a thunk that is currently being evaluated. We see in Section 7.4.2 that a thunk is overwritten with a blackhole when it is entered and see in Section 7.4.2 that this blackhole is overwritten with an indirection to a value once the thunk has been fully evaluated.

The heap representation of a blackhole is very simple. The info pointer points to the standard black hole info table and there is no payload.

7.2 The Stack

In the formal semantics, the stack can contain frames of the following forms:

$\{P_i\}_0^n$	Case Match
$@\alpha$	Application
$\#\alpha$	Thunk update
$\otimes\alpha$	Primop, awaiting first argument
$n\otimes$	Primop, awaiting second argument

Figure 7.6 illustrates the way that GHC implements its stack. The GHC stack is very similar in structure to the GHC heap; it is a continuous block of memory containing a sequence of stack frames. Each stack frame corresponds directly to a stack frame in the semantics. The structure of a stack frame is very similar to that of a heap closure; the first word points to an info table describing the stack frame, and the rest of the words carry the data associated with the stack frame. The stack pointer $\mathbf{Sp}$ points to the start of the topmost stack frame. Local variables may be stored in the stack space above $\mathbf{Sp}$.[5]

[5]In the real implementation, $\mathbf{Sp}$ points to the topmost occupied word in the stack and thus local variables are below $\mathbf{Sp}$. However we find that it is easier to describe the implementation if we instead say that $\mathbf{Sp}$ points to the topmost stack frame.

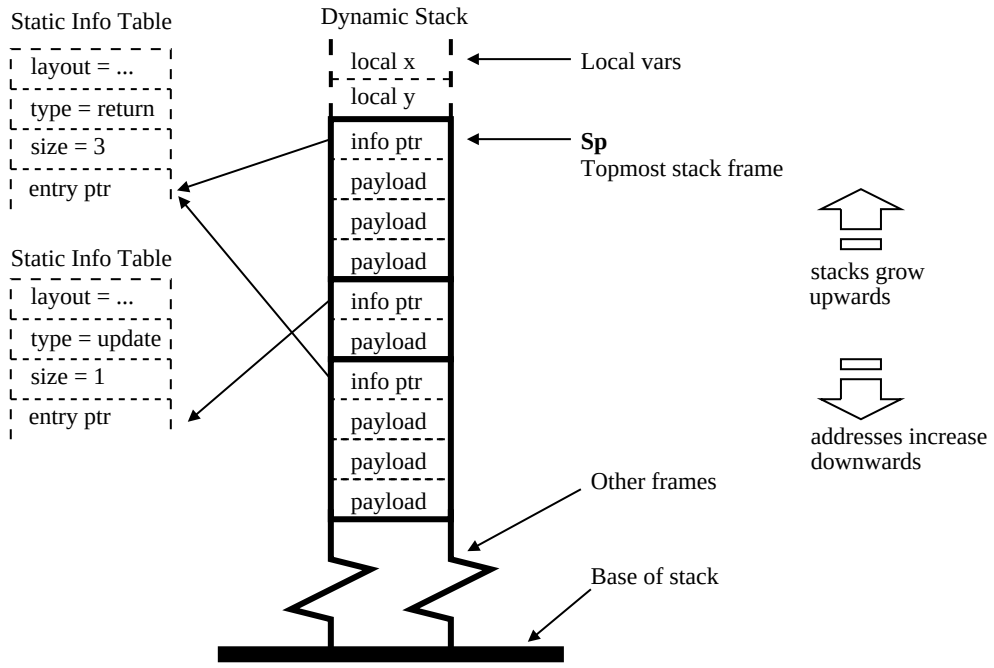


Figure 7.6: Structure of Stack Frames

In the following sections, we explain how each stack frame described in Section 4.2 is represented in the GHC stack.

7.2.1 Case Return Frames

$$\{P_i\}_0^n$$

For any given case return frame, all of the alternatives $\{P_i\}$ must correspond to constructors from the same data-type. GHC divides data-types into two categories. *Large data-types* are those that have more than a given number of constructors (typically more than 8), while *small data-types* are those that have less than that number of constructors. Small data-types are typically the most common, due largely to the fact that lists and booleans are both small data-types.

GHC has two return conventions for constructors, known as *direct returns* and *vectored returns*. Direct returns are used for large data-types, while vectored returns are used for small data-types.

Direct Returns

Direct returns are illustrated in Figure 7.7. Each case alternative is compiled into a separate block of code. The entry code for the case return frame will examine the constructor tag of the value returned and jump to the block of code for whichever case alternative is appropriate.

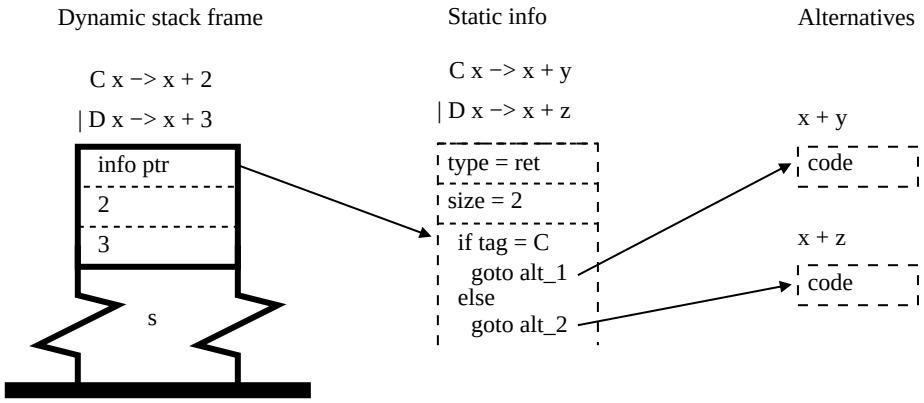


Figure 7.7: Direct Returns

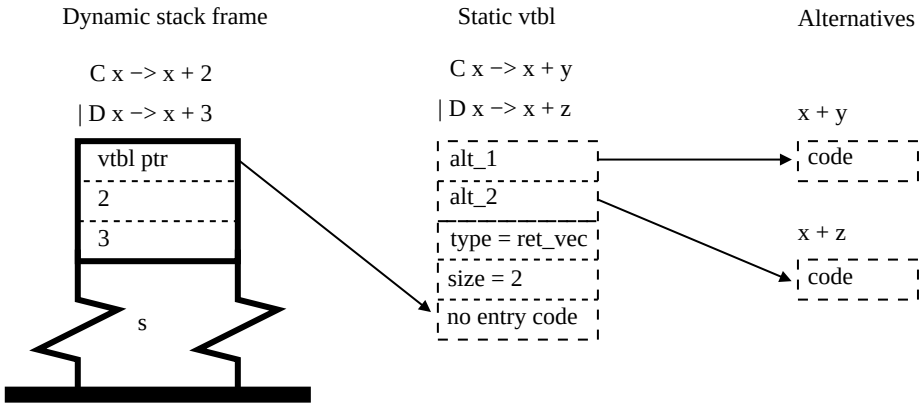


Figure 7.8: Vectored Returns

Vectored Returns

Vectored returns are illustrated in Figure 7.8. Rather than having a single entry point, the info table has a vector of pointers to the alternatives. If the return code knows the constructor of the value it is returning, then it can retrieve the relevant return address from this vector and return do it directly—avoiding the extra test and branched required for a direct return.

For reasons of simplicity, the rest of this chapter assumes that all returns are direct returns.

7.2.2 Application Frames

$$@_\alpha$$

Application frames are illustrated by Figure 7.9. GHC allows an application frame to contain more than one argument. A set of standard info tables are provided, each holding a different number of arguments. An application frame will have the info table appropriate for the number of argument being passed, and will have the passed arguments as its payload. We discuss the entry code for an application frame in Section 7.4.3.

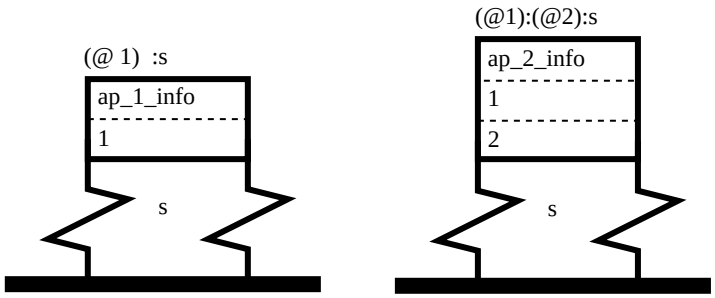


Figure 7.9: Application Frame

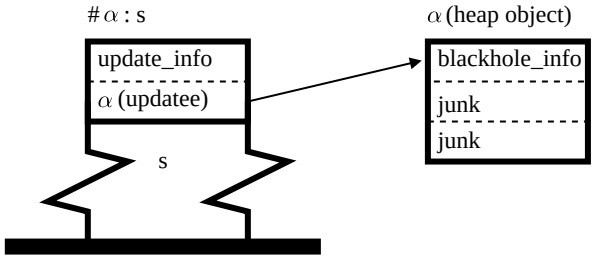


Figure 7.10: Update Frame

7.2.3 Update Frames

$$\#\alpha$$

Figure 7.10 illustrates the way that GHC implements update frames. The first word of the stack frame points to a standard info table shared by all update frames, and the second word holds α (the *updatee* pointer).

7.2.4 Primitive Operation Frames

These frames are implemented in exactly the same way as direct return **case** frames.

7.3 Evaluating Expressions

In the formal semantics, the state contains a command c which tells the virtual machine what it should be doing. In the GHC implementation, the command is represented by the *instruction pointer*, which points to a block of code that will carry out that command:

- For an evaluation command E, the instruction pointer will point to a block of code that will evaluate E, given the local bindings on the stack.
- For a return command $\triangledown\alpha$, the instruction pointer will point to some code that will return α to the topmost stack frame.

module	::=	{*defn*}	module with definitions
defn	::=	**code** *label*: {absc}	code block within a module
	|	**data** *label*: {*literal*}	data block
absc	::=	*lval* **:=** *rval*	assignment
	|	**branch** *rval label*	conditional branch
	|	**jump** *rval*	unconditional jump
rval	::=	*literal*	explicit constant
	|	*label*	code label
	|	*primop* $\{rval_i\}_0^n$	primitive operation applied to args
	|	*lval*	
lval	::=	*reg*	a register
	|	*global*	data label
	|	*lval*[*offset*]	indirect with offset
reg	::=	**Sp**	the stack pointer
	|	**Hp**	the heap pointer
	|	**Node**	the node pointer

Figure 7.11: Abstract C

- For a demand command $\circledcirc\alpha$, the instruction pointer will point to a some code that will find the value of α.

If, in the semantics, the following transition would take place:

$$\Gamma; c; s \quad \longrightarrow \quad \Gamma'; c'; s'$$

Then, for any GHC runtime state that corresponds to (Γ, c, s), executing the code pointed to by the instruction pointer will cause the GHC runtime state to transform into one that corresponds to $(\Gamma'; c'; s')$.

7.3.1 Abstract C

GHC compiles each static expression in the source program into a block of code that will evaluate instances of that expression. For example $x + y$ will compile into a block of code that will evaluate expressions of the form $x + y$ given that the stack contains the heap references that x and y are bound to.

Rather than compiling directly to native machine code, GHC compiles static expressions to a low level intermediate language called *abstract C*. A simplified form of abstract C is pre-

sented in Figure 7.11. Most of this should be fairly self-explanatory. The $\mathbf{Node}$ register is a distinguished register used in various calling and return conventions. The $\mathbf{Sp}$ register always points to the topmost stack frame , and the $\mathbf{Hp}$ register always points to the next free word in the heap.[6]

7.3.2 Calling Conventions

Several standard calling conventions are used, which we outline below:

- When we jump to the entry code for a closure, $\mathbf{Node}$ points to the closure.
- When we jump to the body code for a function, $\mathbf{Node}$ points to the function closure and all function arguments are provided on the stack, above $\mathbf{Sp}$.

  If the function has no free variables then $\mathbf{Node}$ need not be set.
- When we return to a stack frame, $\mathbf{Node}$ points to the value being returned, and $\mathbf{Sp}$ points to the stack frame being returned to.

7.4 Implementing the Evaluation Rules

Every expression is compiled into code that will evaluate that expression according to the Lazy Evaluation subset of the rules given in Section 4.2. In the following subsections, we explain how each of these rules is implemented in GHC.

7.4.1 Evaluate a Value Constant

A value V compiles to code that builds a representation of that value in the heap, and returns it to the topmost return frame. For example, the lambda expression $\lambda xy.x + y + p + q$ might compile to the following code (rule *(VAL)*):

[6] We see in Section 10.1.2 that we need to regularly check that the $\mathbf{Hp}$ and $\mathbf{Sp}$ register do not point past the ends of the heap or stack respectively. We ignore such details in this chapter.

Code for $\lambda xy.x + y + p + q$:

$\mathbf{Hp}[0] := 2574\_\text{info}$	this is function 2574
$\mathbf{Hp}[1] := p$	save free variables
$\mathbf{Hp}[2] := q$	
$\mathbf{Node} := \mathbf{Hp}$	
$\mathbf{Hp} := \mathbf{Hp} + 3$	
$\mathbf{jump}\ \mathbf{Sp}[0]$	return it

The compiler will also generate an info table describing the function:

Info table for $\lambda xy.x + y + p + q$:

$\mathbf{data}$:	
2574_body	address of body code
2	arity (two arguments)
FUN	type (a function)
2	size (two free vars)
$\mathbf{code}\ 2574\_\text{info}$:	
$\mathbf{jump}\ \mathbf{Sp}[0]$	already a value
$\mathbf{code}\ 2574\_\text{body}$:	
code to evaluate the function body	

Similarly, the constructed expression $C\ x\ y$ might compile to the following code:

Code for $C\ x\ y$:

$\mathbf{Hp}[0] := \text{C\_info}$	create constructed value in the heap
$\mathbf{Hp}[1] := x$	
$\mathbf{Hp}[2] := y$	
$\mathbf{Node} := \mathbf{Hp}$	
$\mathbf{Hp} := \mathbf{Hp} + 3$	
$\mathbf{jump}\ \mathbf{Sp}[0]$	return it

The info table for C (C_info) will have been created when the declaration for the constructor was compiled.

7.4.2 Demand the Value of a Closure

A variable x compiles to code that demands the value of the closure referenced by that variable. It does this by jumping to the entry code for the closure (rule *(VAR)*):

Code to evaluate x:

$\mathbf{Node} := x$	load x into $\mathbf{Node}$
$\mathbf{jump}\ \mathbf{Node}[0]$	enter the closure

The entry code for a closure will behave like one of the rules *(DEM1)*, *(DEM2)*, *(DEM3)*, depending on the type of closure.

If the closure is a value, then the entry code will return a pointer to itself (rule *(DEM1)*):

Entry code for a value $(\!|V|\!)$ (direct return):

$\mathbf{jump}\ \mathbf{Sp}[0]$	return $\mathbf{Node}$ (which points to us)

If the closure is an indirection, then the entry code will enter the indirectee (rule *(DEM2)*):

Entry code for an indirection $\langle x \rangle$:

$\mathbf{Node} := \mathbf{Node}[1]$	load x into $\mathbf{Node}$
$\mathbf{jump}\ \mathbf{Node}[0]$	enter x

If the closure is a thunk, then the entry code will push an update frame, save any bindings from the thunk, overwrite the thunk with a black hole, and then evaluate the thunk body (rule *(DEM3)*):

Entry code for a thunk E:

$\mathbf{Sp} := \mathbf{Sp} - 2$	push an update frame onto the stack
$\mathbf{Sp}[0] := \mathrm{update\_info}$	
$\mathbf{Sp}[1] := \mathbf{Node}$	
$\mathbf{Sp}[-2] := \mathbf{Node}[1]$	save local bindings from the thunk
$\mathbf{Sp}[-1] := \mathbf{Node}[2]$	
...	
$\mathbf{Node}[0] := \mathrm{blackhole\_info}$	overwrite the thunk with a blackhole
code to evaluate E	

In the semantics, we represent blackholing by removing the binding from the heap.

When control eventually returns to the update frame, it will overwrite its updatee with the value returned to it (rule *(UPD)*):[7]

Return code for an update frame (update_info):

$\mathbf{Sp}[1][0] := \mathrm{ind\_info}$	overwrite updatee with an indirection
$\mathbf{Sp}[1][1] := \mathbf{Node}$	
$\mathbf{jump}\ \mathbf{Sp}[0]$	return

[7] The real GHC implementation is rather complex here as an update frame can be returned to with either a direct return or a vectored return.

7.4.3 Function Application

A function application $E\ x$ compiles to code that will evaluate E to a function, and then apply this function to x. The generated code will push a function application frame, and then execute the code to evaluate E (rule *(APP1)*):[8]

Code to evaluate $E\ x$:

$\mathbf{Sp} := \mathbf{Sp} - 2$	push an application frame
$\mathbf{Sp}[0] := \mathrm{ap\_1\_info}$	
$\mathbf{Sp}[1] := x$	
code for E	

When a function closure is returned to the application frame, the return code will check that the function expects the number of arguments provided (in this case one). If it does, then the return code will jump to the body code for the function. This corresponds to rule *(APP2)*:

Return code for an application frame with one argument (ap_1_info):

$\mathbf{branchif}\ (\mathbf{Node}[0][\mathrm{arity}] \neq 1)\ \mathrm{arity\_stuff}$	check the arity
$\mathbf{Sp} := \mathbf{Sp} + 2$	pop the frame
$\mathbf{jump}\ \mathbf{Node}[0]$	call the function

Once the info pointer for the application frame has been popped, the arguments will be in their correct positions ready for a function call; there is no need for the arguments to be copied.

We do not discuss what the code at arity_stuff does. Interested readers are encouraged to refer to Marlow and Peyton Jones [MP03]. We also omit a lot of other details that make the real implementation more efficient, but also significantly more complex.

[8] This function application model is known as eval/apply [MP03].

7.4.4 Case Expression

A **case** expression $\mathbf{case}\ E\ \mathbf{of}\ \{P_i\}_0^n$ compiles to code that will evaluate the *scrutinee* E and then match the result against the alternatives in P. It does this by pushing a **case** return frame and then executing the code for E (rule *(CASE1)*):[9]

Code for $\mathbf{case}\ E\ \mathbf{of}\ \{P_i\}_0^n$:

$\mathbf{Sp} := \mathbf{Sp} - (1 + \text{env\_size\_1234})$	push a **case** return frame
$\mathbf{Sp}[0] := \text{expr1234\_ret\_info}$	
$\mathbf{Sp}[1] := ?; \mathbf{Sp}[2] := ? \ldots$	(vars live in $\{P_i\}_0^n$)
code to evaluate E	

When a reference to a constructed value is returned to the **case** return frame, it will select one of its alternatives and jump to the code for that alternative (rule *(CASE2)*). The alternative is selected by referring to the constructor tag in the closure's info table. For a direct return, the code will resemble the following:[10]

Return code for a **case** frame (expr1234_ret, direct return):

$\mathbf{Sp} := \mathbf{Sp} + (1 + \text{env\_size\_1234})$	pop the **case** frame
$\mathbf{branchif}\ (\mathbf{Node}[0][tag] = 1)\ 1234\_\text{alt}\_1$	jump to the correct alternative
$\mathbf{branchif}\ (\mathbf{Node}[0][tag] = 2)\ 1234\_\text{alt}\_2$	
...	

In this case the constructor alternatives will have been compiled to the blocks 1234_alt_1, 1234_alt_2, ...

7.4.5 Exceptions

If an exception is raised, a special RTS routine is called. This routine walks down the stack, tearing frames off as it goes. The behaviour of this routine is very similar to that described by rules *(EXN1),(EXN2),(EXN3),* and *(EXN4).*

[9] In this example, we write ? to represent the location of a free variable.

[10] Recall that the first word of a closure ($\mathbf{Node}[0]$) points to the info table.

7.4.6 Lazy Evaluation of a Let

A $\mathbf{let}$ expression $\mathbf{let}\ x = E\ \mathbf{in}\ E'$ compiles to code that will build a *thunk* for E in the heap, and then execute the code for E':

Code for $\mathbf{let}\ x = E\ \mathbf{in}\ E'$:	
$\mathbf{Hp}[0] := 4573\_\mathrm{info}$	Create a thunk in the heap
$\mathbf{Hp}[1] := ?$	(free variables)
$\mathbf{Hp}[2] := ?$	
...	
$\mathbf{Sp}[-1] := \mathbf{Hp}$	save address as a local var
$\mathbf{Hp} := \mathbf{Hp} + (1 + \mathrm{env\_size\_4573})$	move $\mathbf{Hp}$ to next free word
code for E'	

We will also generate an info table and entry code for the thunk. The entry code will behave as described is Section 7.4.2, and the info table will be as described in Section 7.1.3.

GHC has various special cases for compiling $\mathbf{let}$ expressions. In particular, if the right hand side of a $\mathbf{let}$ is already a value then GHC will build a representation of this value in the heap rather than building a thunk that will evaluate it. The code generated will thus resemble that given in Section 7.4.1, followed by the code for E'.

7.4.7 Summary

Code blocks are generated for the following things:

- Alternatives of a case
- Right hand side of a $\mathbf{let}$ expression (thunk)
- Function body

Heap is allocated in the following places:

- Creating a value (rule *(VAL)*)
- Creating a thunk for a $\mathbf{let}$ (rule *(LAZY)*)

7.5 Other Details

7.5.1 Garbage Collection

The GHC heap is garbage collected. At periodic intervals, the garbage collector sweeps through the heap and throws away any closures that are no longer reachable. In order to determine what is reachable, the garbage collector needs to know which fields in a closure's payload are pointers, rather than literal values. This information is encoded in a special *layout* field in the closure info table.

7.5.2 Lazy Blackholing

As with any garbage collected system, one must take care to ensure that a closure is considered to be garbage if it is not reachable by program execution. In particular, it is important that pointers held in a thunk closure are not considered to be reachable while the thunk is being evaluated. This is the purposes of GHC's blackholing [Jon92].

The simplest approach to blackholing is *eager blackholing*. This is the strategy described in Section 7.4.2. With eager blackholing, a thunk blackholes itself as soon as it is entered.

The standard GHC implementation takes an alternative approach, known as *lazy blackholing* [MP98]. Lazy blackholing does not blackhole a thunk when it is entered. Instead, it waits until garbage collection time, and then blackholes every thunk that is pointed to by an update frame on the stack. The motivation for this is that, by only blackholing thunks at garbage collection time, it reduces the number of blackholing operations that must be done.

Unfortunately, as we explain in Section 8.4, lazy blackholing turns out to interact very badly with Optimistic Evaluation. We thus disable it in our version of GHC.

CHAPTER 8

Switchable Let Expressions

Now that we have explained how the GHC execution model works, we can explain how Optimistic Evaluation extends this:

- We begin, in Section 8.1 by describing the way that we implement switchable **let** expressions.

- In Section 8.2, we describe *Flat Speculation*: an alternative implementation technique that we have also implemented.

- In Section 8.3, we describe *Semi-Tagging*: an implementation technique that improves the performance of **case** expressions. While the idea of semi-tagging has been proposed before [PMR99], it is far more effective under Optimistic Evaluation than under Lazy Evaluation.

- Finally, in Section 8.4, we explain why lazy blackholing interacts badly with Optimistic Evaluation.

Online profiling and abortion are discussed in Chapters 9 and 10 respectively.

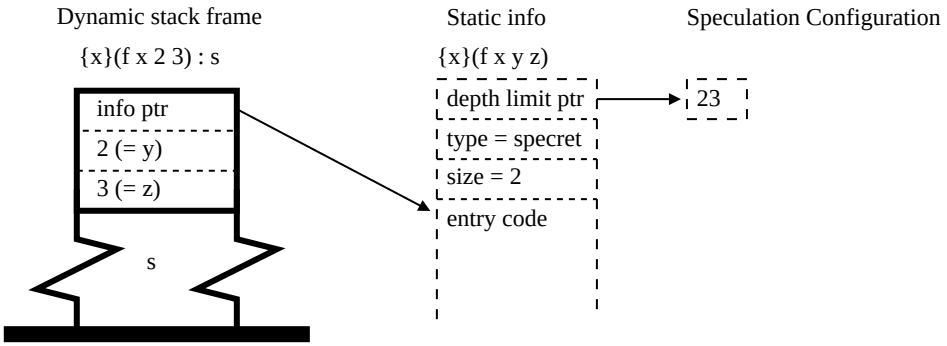


Figure 8.1: Speculative return frame

8.1 Speculative Evaluation of a Let

Section 7.4 described the implementation of GHC by comparing it to the a restricted form of the semantics of Section 4.2. That subset corresponds to Lazy Evaluation, and differs from the full semantics in the following ways:

- It omits the rules *(SPEC1)* and *(SPEC2)* that implement speculative evaluation.
- It omits the speculation configuration parameter Σ from the evaluation relation.
- It omits the side condition on rule *(LAZY)* that causes the rule to only be applied if the speculation configuration says it should.
- It does not include speculative return frames $\{x\}E$.
- It does not make use of the speculation depth function $specDepth$.

In the subsections that follow, we describe the way in which we have added each of these features to GHC.

8.1.1 Speculation Return Frames

In Section 7.2 we described how GHC represents **case** return frames, application frames, and update frames. Rules *(SPEC1)* and *(SPEC2)* make use of an additional kind of return frame, which we call a *speculative return frame*, written:

$$\{x\}E$$

The representation of a speculative return frame is very simple (see Figure 8.1); the info table contains the code for E, and a pointer to a *depth limit* variable (see Section 8.1.3). The payload of the stack frame contains bindings for any free vars in E.

8.1.2 The $specDepth$ Function

Rules *(LAZY)* and *(SPEC2)* make use of a function $specDepth$ that counts the number of active speculations (defined in Section 4.2.2). While it would be possible to implement $specDepth$ as a function that walks down the stack and counts the number of speculative return frames found, this would not be efficient. Instead, we maintain a global register, $\mathbf{SpecDepth}$, that we increment and decrement whenever we push or pop speculation frames.

Speculative return frames are not just there to provide something to return to. The abortion and profiling systems both rely on the fact that there will always be a speculation frame on the stack if speculation is taking place. It is thus important that we avoid optimisations that might prevent a speculation frame being be pushed when speculation starts. Consider the following example:

$$
\begin{array}{l}
\mathbf{let} \\
\quad x = \mathbf{case}\ f\ y\ \mathbf{of} \\
\qquad\qquad \mathit{True} \rightarrow \mathit{True} \\
\qquad\qquad \mathit{False} \rightarrow g\ y \\
\mathbf{in} \\
\quad E
\end{array}
$$

A naive compiler might think that it could wait until the call to g before pushing a return frame for the $\mathbf{let}$. Unfortunately, this would cause the call to f to take place with no speculative return frame on the stack. If the call to f did not terminate then the abortion system would not realise that it was part of a speculation and so would not abort it.

8.1.3 The Speculation Configuration

In the formal semantics, the speculation configuration Σ is a function that maps $\mathbf{let}$ identifiers to speculation *depth limits*. In the implementation, the depth limit for each $\mathbf{let}$ is represented as a static global variable. The info table for a speculative return includes a pointer to the depth limit for the $\mathbf{let}$.

One might wonder why the depth limit is not itself in the info table. This is because the info table has to live in code space, in order to be placed directly before the entry code, while the depth limit needs to be placed in data space so that it can be written to by the online profiler without causing the instruction cache to be flushed.[1]

[1] Many operating systems require the use of a system call such as `mprotect` to make code writable.

$$\textit{(LAZYX)} \quad \Gamma; \mathbf{let}\ x = E\ \mathbf{in}\ E'; s \quad \longrightarrow_\Sigma \quad \Gamma[\alpha \mapsto E]; \triangledown\alpha; (\{x\}E' : s) \quad \text{if } \Sigma(x) \leq \mathit{specDepth}(s) \text{ and } \alpha \text{ is fresh}$$

Figure 8.2: An alternative lazy rule for $\mathbf{let}$ expressions

8.1.4 Rules *(LAZY)*, *(SPEC1)* and *(SPEC2)*

Rule *(SPEC1)* evaluates the right hand side of a $\mathbf{let}$ speculatively, while rule *(LAZY)* builds a thunk for it. The virtual machine choses which rule to apply based on the current speculation depth and speculation configuration.

For convenience, our implementation of Lazy Evaluation behaves as if *(LAZY)* were replaced with the rule *(LAZYX)* from Figure 8.2. *(LAZYX)* pushes a speculative return frame but immediately returns to it.[2] While the end result is the same, this approach turns out to make implementation easier. In particular, it makes it easier for the lazy and speculative code to share one version of the code for the $\mathbf{let}$ body, E'. One can observe that *(LAZYX)* followed by *(SPEC2)* is equivalent to *(LAZY)*.

The first thing a $\mathbf{let}$ does is push a return frame, incrementing $\mathbf{SpecDepth}$ so as to keep it consistent with the number of speculation frames on the stack. The $\mathbf{let}$ then compares $\mathbf{SpecDepth}$ with its depth limit to see whether it should evaluate speculatively or lazily (rules *(SPEC1)* and *(LAZYX)*):

Code to evaluate $\mathbf{let}\ x = E\ \mathbf{in}\ E'$ (expression id is 3562):	
$\mathbf{Sp} := \mathbf{Sp} - (1 + \mathrm{env\_size})$	push a speculative return frame
$\mathbf{Sp}[0] := 3562\_\mathrm{ret\_info}$	
$\mathbf{Sp}[1] := ?;\ \mathbf{Sp}[2] := ?; \ldots$	(save vars live in E')
$\mathbf{SpecDepth} := \mathbf{SpecDepth} + 1$	increment $\mathbf{SpecDepth}$
$\mathbf{branchif}\ (\mathbf{SpecDepth} > 3562\_\mathrm{limit})\ 3562\_\mathrm{lazy}$	branch to lazy code
$\mathbf{jump}\ 3562\_\mathrm{spec}$	or speculative code

The lazy code builds a thunk for the right hand side of the $\mathbf{let}$ and then immediately returns to the speculative return frame (rule *(LAZYX1)*):

[2] Note that unlike every rule presented previously, this rule can return a reference to something other than a value. This has knock-on effects throughout the runtime system, as one can no longer assume that a returned reference always points to a value.

Code for the lazy branch of a **let** (3562_lazy:	
$\mathbf{Hp}[0] := 3562\_thunk\_info$	build a thunk in the heap
$\mathbf{Hp}[1] :=?;\ \mathbf{Hp}[2] :=?; \ldots$	(save vars live in E)
$\mathbf{Node} := \mathbf{Hp}$	
$\mathbf{Hp} := \mathbf{Hp} + (1 + \text{env\_size})$	
$\mathbf{jump}\ \mathbf{Sp}[0]$	return the thunk[3]

The speculative code (3562_spec) will be the code to evaluate E (rule *(SPEC1)*).

When control returns to the speculative return frame, it will pop itself off the stack, decrement $\mathbf{SpecDepth}$, and evaluate the body of the **let** (rule *(SPEC2)*):

Return code for a speculative return frame '$\{x\}E$':	
$\mathbf{Sp} := \mathbf{Sp} + (1 + \text{env\_size})$	pop the frame
$\mathbf{SpecDepth} := \mathbf{SpecDepth} - 1$	decrement $\mathbf{SpecDepth}$
code to evaluate E'	(x is in $\mathbf{Node}$)

In the real implementation, there are additional complexities involving heap/stack checking and direct/vectored returns; however these are not particularly interesting and so we do not discuss them here.

8.1.5 Not All Let Expressions are Speculated

As with normal **let** expressions (Section 7.4.6), if the right hand side of a **let** is already a value, we simply build a representation of that value in the heap and continue. There is no need to have separate lazy and speculative versions in this case.

Our current implementation also avoids speculating recursive **let** expressions, because, if the right hand side of a **let** is speculated, then there is no thunk to which the binder can be bound during evaluation of the right hand side. In such cases, the code generated will be the same as for Lazy Evaluation.

[3] This will actually be a direct jump to 3562_ret_info.

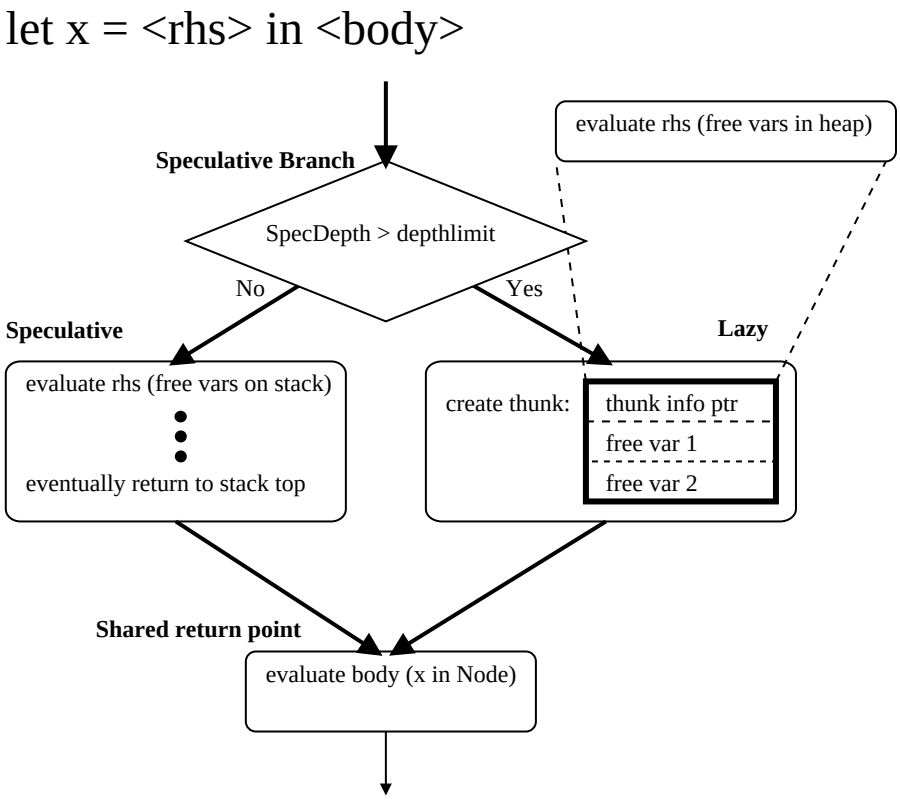


Figure 8.3: Flow diagram for evaluation of a **let** expression

8.1.6 Avoiding Code Explosion

The entire process of evaluating a **let** is summarised by Figure 8.3. Note that the code to evaluate the right hand side E is duplicated, but the code to evaluate the body E' is shared.

The reason for the duplication of the right hand side is to allow the compiler to specialise the two blocks of code relative to the environment in which they are used. In particular the lazy code will expect to find its free variables in the payload of a closure pointed to by **Node**, while the speculative code will expect to find its free variables on the stack and in registers.

It is important that the size of a duplicated expression is kept small, otherwise there can be a potentially exponential increase in code size. Consider for example the following program:

$$\mathbf{let}\ x = (\mathbf{let}\ y = (\mathbf{let}\ z = E_1\ \mathbf{in}\ E_2)\ \mathbf{in}\ E_3)\ \mathbf{in}\ E_4$$

If we were to duplicate every **let** right hand side, then we would compile E_1 8 times. If we had another layer of nesting, then the inner expression would be compiled 16 times (Figure 8.4). While such programs are not often written by programmers, they can often appear as the result of inlining, or other compiler optimisations. Even without nested **let** expressions, duplicating the right hand side of every **let** can almost double the code size if there are a lot of **let** expressions with large right hand sides. This is clearly undesirable.

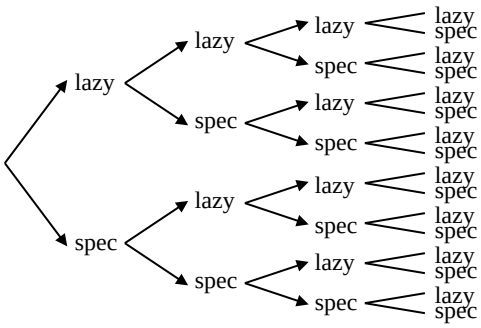


Figure 8.4: Uncontrolled duplication of code leads to an explosion in code size

In practice, it is only worth duplicating an expression if it is very small indeed (e.g. an integer addition). For larger expressions, we use an expression lifting transformation to convert the right hand side of the **let** into function call. For example, if E is a large expression and has free variables y and z then we will make the following transformation:

$$\begin{gathered}\mathbf{let}\ x = E\ \mathbf{in}\ E' \\ \Downarrow \\ \mathbf{let}\ x = f\ y\ z\ \mathbf{in}\ E' \\ \text{where } f\ y\ z = E\end{gathered}$$

We generate a new function f whose body is E and whose arguments are the free variables of E. We transform the **let** expression so that its right hand side is a call to this function (and thus a small expression). This transformation can be considered to be a special case of lambda lifting [Joh85] for functions with no arguments.

In the extreme case, we can apply this transformation to the right hand side of every **let** expression, resulting in the evaluation scheme described in Section 8.2.

8.2 Flat Speculation

If the transformation described in Section 8.1.6 is applied to every **let** expression, then all **let** expressions will be of the following form:

$$\mathbf{let}\ x = f\ y_1\ \ldots\ y_n\ \mathbf{in}\ E'$$

This allows us to implement switchable **let** expressions very differently, using a technique that we call *Flat Speculation*. We have implemented both Flat Speculation and the technique described in Section 8.1; we compare their performance in Section 12.7.1.

8.2.1 Evaluating a Let Expression Speculatively

Under Flat Speculation, the code generated by the compiler for a **let** expression assumes that the **let** will always be evaluated speculatively. A **let** expression thus compiles to the following code:

Code to evaluate **let** $x = f\ y_1\ \ldots\ y_n$ **in** E' (expression id is 3562):	
$\mathbf{Sp} := \mathbf{Sp}\ -(1 + \text{env\_size})$	push a speculative return frame
$\mathbf{Sp}[0] := 3562\_\text{ret\_info}$	
$\mathbf{Sp}[1] := ?; \mathbf{Sp}[2] := ?; \ldots$	(save vars live in E')
$\mathbf{SpecDepth} := \mathbf{SpecDepth} + 1$	increment $\mathbf{SpecDepth}$
$\mathbf{Sp}[-1] := y_1;\ \mathbf{Sp}[-2] := y_2; \ldots$	put arguments on the stack
$\mathbf{jump}\ \text{f\_body}$	call f

The info table for the speculative return frame will contain three extra fields:[4]

- **rhsfun:** The function that this **let** calls (in this case f)
- **argcount:** The number of arguments the function required
- **jmpaddr:** The address of the **jump** instruction in memory.

8.2.2 Evaluating a Let Expression Lazily

If the profiler decides to switch the **let** to being evaluated lazily, then it will overwrite the call to f with a call to the standard function app_lazy_N, where N is the number of arguments passed to f. This code modification is extremely simple; all the profiler has to do is find the address of f in the jump instruction and overwrite it with the address of the appropriate `ap_lazy` function. While this write to code will probably cause an instruction cache flush and thus considerable cost, these writes are extremely rare, and so the amortised cost is negligible.

The `ap_lazy` function finds the info table for the speculative return frame on the top of the stack and uses the $rhsfun$ field to build a special *function application thunk* using a standard info table `ap_N_info`, where N is the number of function arguments:

[4] It actually has some others as well, which we do not discuss here.

Code for `ap_lazy_2`:

$\mathbf{Hp}[0] := \mathrm{ap\_2\_info}$	create a function application thunk
$\mathbf{Hp}[1] := \mathbf{Sp}[0][\mathit{rhsfun}]$	(the function f)
$\mathbf{Hp}[2] := \mathbf{Sp}[-2]$	(arguments)
$\mathbf{Hp}[3] := \mathbf{Sp}[-1]$	
$\mathbf{Node} := \mathbf{Hp}$	
$\mathbf{Hp} := \mathbf{Hp} + 4$	
$\mathbf{jump}\ \mathbf{Sp}[0]$	return this thunk

The entry point for a function application thunk is very simple. It simply pushes an update frame, copies its arguments onto the stack, blackholes the thunk, and then jumps to the function entry point:

Code for `ap_2_ret`:

$\mathbf{Sp} := \mathbf{Sp} - 2$	push an update frame
$\mathbf{Sp}[0] := \mathrm{update\_info}$	
$\mathbf{Sp}[1] := \mathbf{Node}$	
$\mathbf{Sp}[-2] := \mathbf{Node}[2]$	put the arguments onto the stack
$\mathbf{Sp}[-1] := \mathbf{Node}[3]$	
$\mathbf{Node}[0] := \mathrm{blackhole\_info}$	blackhole the thunk
$\mathbf{jump}\ \mathbf{Node}[1]$	jump to f

8.2.3 Chunky Entry Points

If the profiler wants the **let** to be speculated only up to a certain depth,[5] then it will overwrite the call to f with a call to the standard function `app_chunky_N`, where N is the number of function arguments. The `ap_chunky` function tests the current speculation depth against the depth limit for the **let** (referenced from the info table) and then decides whether to call f or `ap_lazy_N`:

[5] This is the usual case for the system described in Section 8.1.

Code for ap_chunky_2:

$\mathbf{branchif}\ (\mathbf{SpecDepth}\ > \mathbf{Sp}[0][limit][0])\ \mathrm{ap\_lazy\_2}$	evaluate lazily
$\mathbf{jump}\ \mathbf{Sp}[0][rhsfun]$	evaluate speculatively

In flat speculation, the depth limit for a **let** is a member of the following set:[6]

$$\{0, 1, \ldots, MAXDEPTH\} \cup \{\infty\}$$

If the depth limit is ∞ then the **let** will always be speculated and the code for the **let** will jump directly to f. If the depth limit is 0 then the **let** will never be speculated and the code for the **let** will jump to ap_lazy_N. For all other depth limits, the code for the **let** will jump to ap_chunky_N.

8.3 Semi-Tagging

A typical program will evaluate a large number of expressions of the following form:

$$\mathbf{case}\ x\ \mathbf{of}\ \{P_i\}_0^n$$

Not only are such expressions frequently written by the programmer, but they are also generated as a result of desugaring other expressions within the compiler. For example, record field selection and addition of boxed integers both reduce to case expressions that scrutinise a variable.

8.3.1 Scrutinising Variables in Normal GHC

In Section 7.4.4 we saw that GHC normally implements such expressions by pushing a return frame and then entering the closure referenced by x.

Normal GHC code for $\mathbf{case}\ x\ \mathbf{of}\ \{P_i\}_0^n$:

$\mathbf{Sp} := \mathbf{Sp} - (1 + \mathrm{env\_size})$	push a **case** return frame
$\mathbf{Sp}[0] := 1234\_\mathrm{ret\_info}$	
$\mathbf{Sp}[1] :=?;\ \mathbf{Sp}[2] :=?;\ \ldots$	
$\mathbf{Node} := x$	enter closure for x
$\mathbf{jump}\ \mathbf{Node}[0]$	

[6] Infinity is represented internally as $MAXDEPTH + 1$.

If x refers to a value, then the entry point for x will immediately return to the **case** return frame. In such cases the whole process of pushing an return frame and entering x will have been pointless.

8.3.2 Scrutinising Variables with Semi-Tagging

If x will usually be a value, then it may be more efficient to test whether x is a value before entering it. This leads to the following code:

Value-testing code for **case** x **of** $\{P_i\}_0^n$:

$\mathbf{Sp} := \mathbf{Sp} - (1 + \text{env\_size})$	push a **case** return frame
$\mathbf{Sp}[0] := 1234\_\text{ret\_info}$	
$\mathbf{Sp}[1] :=?;\ \mathbf{Sp}[2] :=?;\ \ldots$	
$\mathbf{Node} := x$	
branchif $(\text{ISVALUE}(x))\ 1234\_\text{ret\_info}$	return immediately if a value
jump $\mathbf{Node}[0]$	otherwise enter x

For a constructed value this code will perform two branches: one to test whether it is evaluated, and then one in `1234_ret_info` to see what constructor it is. We can combine these two together by treating a thunk as if it was a constructed value with the tag `UNEVAL`.

Semi-Tagged code for **case** x **of** $\{P_i\}_0^n$:

$\mathbf{Node} := x$	
branchif $(\mathbf{Node}[0][\text{tag}] = \mathit{UNEVAL})\ 1234\_\text{normcase}$	normal **case** evaluation
branchif $(\mathbf{Node}[0][\text{tag}] = 1)\ 1234\_\text{alt\_1}$	direct **case** alternative
branchif $(\mathbf{Node}[0][\text{tag}] = 2)\ 1234\_\text{alt\_2}$	

If the value is not evaluated, then we jump to `1234_normcase` which contains the code from Section 8.3.1. This technique will cause a moderate increase in code size as the compiler must generate tag-branching code both inline, and in the return frame. The real implementation also has to take care to ensure that both blocks of branching code place local variables in the same locations.

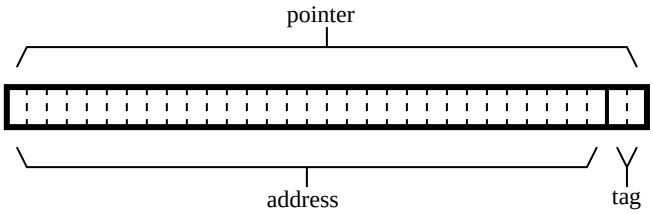


Figure 8.5: Semi-tagging uses the least signifi cant bits of the address to store tag information

If x is not a value, then semi-tagging will be slower than GHC's normal implementation; however, if x is a value, then semi-tagging will be faster. Optimistic Evaluation increases the likelihood that x will be evaluated and so makes semi-tagging a more appealing idea. Indeed, we give results in Section 12.5 that demonstrate that, although semi-tagging is not worthwhile under Lazy Evaluation, it is worthwhile under Optimistic Evaluation.

8.3.3 Storing the Tag in the Address

The simplest way to implement semi-tagging is to store the constructor tag in the info table of every constructor and every thunk. However this still requires an indirect read from the closure info table in order to find the tag. A slightly more efficient approach is to encode the tag in the address of the closure (Figure 8.5). Closure info tables are always aligned to word boundaries and so, on a 32 bit architecture, the bottom two bits of the address will thus always be zero. If the closure data-type has three or fewer constructors then these bits can be used to encode the constructor:

00 unevaluated or unknown
01 a value with constructor 1
10 a value with constructor 2
11 a value with constructor 3 or greater

If the data-type has more than three constructors then we must look at the tag in the info table in order to distinguish between constructors with identifiers greater than 3.

These tags impose very little additional overhead. There is no need to clear the tags in a closure pointer before reading from it or entering it. If one is reading a field from a constructed value then one will know what constructor it is and so be able to adjust the offset to take account of that. Similarly, one will only enter a closure if one has already tested that its tag is `UNEVAL`; however the `UNEVAL` tag is 00, and so the address will be untagged.

All that is required is that we maintain the following invariants:

- When a value is returned, $\mathbf{Node}$ is correctly tagged
- When a closure is entered, $\mathbf{Node}$ is untagged
- All pointers are either untagged or correctly tagged

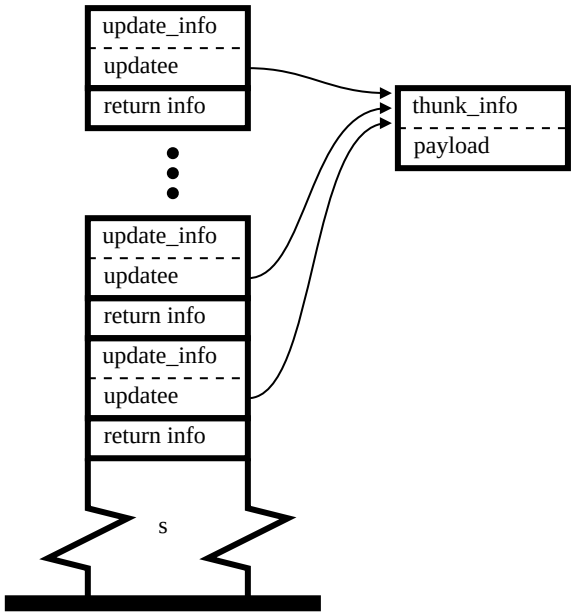


Figure 8.6: Lazy blackholing can cause several update frames to reference the same closure

8.4 Problems with Lazy Blackholing

In Section 7.5.2 we described the concept of lazy blackholing, and remarked that it interacted poorly with Optimistic Evaluation. In this Section we explain why.

8.4.1 Finite Depth Looping

Consider the following program:[7]

$$
\begin{array}{l}
\textbf{let } x = \\
\quad \textbf{let } y = \textbf{case } x \textbf{ of} \\
\qquad \mathit{Just}\ z \rightarrow \mathit{Cons}\ 1\ z \\
\quad \textbf{in } \mathit{Just}\ y \\
\textbf{in } \ldots
\end{array}
$$

What happens if y is speculated, but x is not speculated? If blackholing is lazy, then the thunk for x will repeatedly enter itself, pushing a series of update frames onto the stack for the same closure. This is illustrated in Figure 8.6.

If this looping continues indefinitely, then the program will eventually run out of stack, causing the runtime system to spot the long-running looping computation and abort it. In this case, the duplicate update frames will never be entered and so nothing particularly bad happens.

[7]Note that the **let** for x is recursive, and thus cannot be speculated (Section 8.1.5); however the **let** for y is not recursive, and so can be speculated.

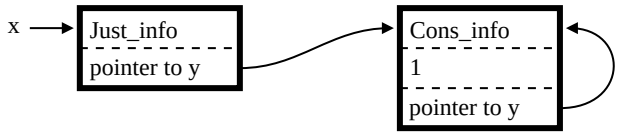


Figure 8.7: This is what we would expect x to evaluate to

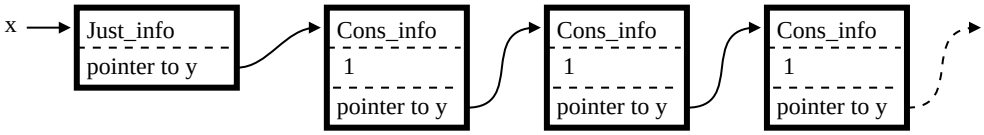


Figure 8.8: Lazy blackholing can cause loss of sharing

However, if chunky evaluation is used then y will only be speculated to a finite depth. At some point, the inner evaluation of x will decide not to speculate y and will instead return the value $Just\ y$. The program will return through several update frames, each of which attempts to update the same value.

This can cause several problems:

- Various parts of the GHC runtime (particularly the generational garbage collector [SP93]) can no longer assume that a thunk will only be overwritten with an indirection once.
- There can be a loss of sharing (Section 8.4.2).
- Values may be overwritten with indirections to themselves (Section 8.4.3).

8.4.2 Loss of Sharing

If the program was evaluated lazy, then we would expect x to eventually evaluate to the structure illustrated in Figure 8.7, however, if we are speculating y and are blackholing lazily, then we may end up with the structure illustrated in Figure 8.8. Rather than having one closure for y we have several.

Every time x is entered, it will create a new speculation for y. This speculation will enter x again. When an evaluation of x returns, the speculation for y will evaluate to a cons containing the previous closure for y, rather than itself.

8.4.3 Indirection Loops

Consider the situation illustrated in Figure 8.9. The frame at the top of the stack is an update frame, however another update frame has already overwritten the updatee with an indirection. What would happen if a garbage collection were to take place at this point?

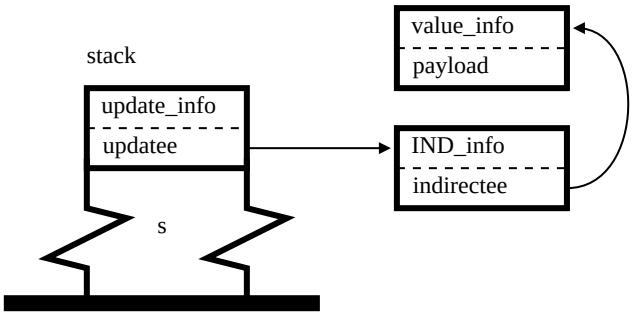


Figure 8.9: If garbage collection strikes here, we could be in trouble.

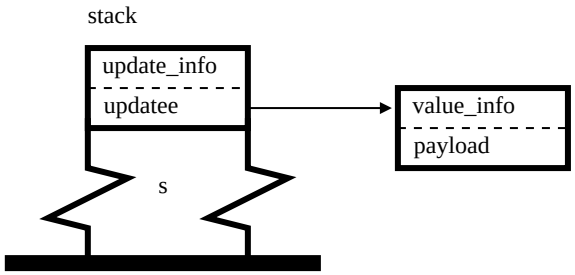


Figure 8.10: Short-circuiting indirections causes problems

When garbage collecting, normal behaviour is to replace any pointer to an indirection with a pointer to the indirectee. If we do this with the state in Figure 8.9, then we end up with the state illustrated in Figure 8.10.

When control returns to the update frame it will overwrite the value with an indirection to another value. If the new value is the same as the old value then the update frame will overwrite it with an indirection to itself—causing chaos. If the old value is a static value, then it may be in read-only memory, in which case the program will fail with a segmentation fault.

These problems could probably be fixed. One plausible solution would be for the garbage collector to remove from the stack any update frame whose indirectee is not a thunk. However solving all of the problems with lazy blackholing adds considerable complexity to our implementation for very little performance benefit. We have thus made the decision to remove lazy blackholing from GHC, reverting instead to eager blackholing. The performance implications of this decision are analysed in Section 12.7.5.

CHAPTER 9

Online Profiling

In this chapter we describe our implementation of online profiling, explaining how the implementation described in Chapter 8 can be extended so that it implements the burst profiling semantics given in Section 6.4.2.

- We start, in Section 9.1, by describing the way in which the runtime state can be extended to support online profiling.
- In Section 9.2, we explain how we can extend the compiler to generate code that can profile itself.
- In Section 9.3, we discuss ways that heap residency can be controlled.
- Finally, in Section 9.4, we explore various technical details.

9.1 Runtime State for Profiling

The burst profiling semantics of Section 6.4.2 extends the runtime state in several ways:

- The runtime state contains a blame count B.
- The runtime state contains a goodness map Π.
- The heap can contain costed indirections $B\langle\alpha\rangle^x$.
- The stack can contain profiled speculation frames $(\{x\}E, B)$.

In the subsections that follow, we describe how we have added each of these features to GHC, causing it to implement the full blame profiling semantics of Section 6.4.2. The burst profiling semantics also introduces additional rules, which we describe in Section 9.2.

9.1.1 The Blame count B

In the formal semantics of Section 6.4.2 the program state contains an integer count B, representing the amount of work that should be blamed on the current venture.

Heap usage as a measure of Work

In Section 3.3.2 we said that work could be measured using any reasonable measure of execution cost. It turns out that the heap allocation is a convenient measure of work. This is for the following reasons:

- It allows much finer grain measurements than would be possible with any OS time counter. This is very important, given that a typical speculation is quite small (See Section 12.4.5).
- It can be calculated very cheaply, by simply looking at the $\mathbf{Hp}$ register.
- The heap usage of a speculation seems to be a reasonable approximation to its time usage (See Section 12.4.2).
- It makes it easier for the profiler to bound heap usage as well as wasted time (See Section 9.3).
- It is easy to arrange that all recursive function calls allocate heap, and thus that any long-running computation must allocate heap.

If the processor provides suitable instructions then work can instead be measured as an exact cycle-count. We discuss this in Section 9.4.4.

Representing Blame using Heap Usage

We represent the blame counter B as the difference between the heap pointer $\mathbf{Hp}$ and a base variable $\mathbf{BlameBase}$:

$$B = \mathbf{Hp} - \mathbf{BlameBase}$$

To add work to B, we subtract it from $\mathbf{BlameBase}$. To save B, we simply calculate B and store it as an integer. To set B to a particular value B', we set $\mathbf{BlameBase}$ to $\mathbf{Hp} - B'$. When heap is allocated, B will increase automatically, reflecting the fact that work has been done.

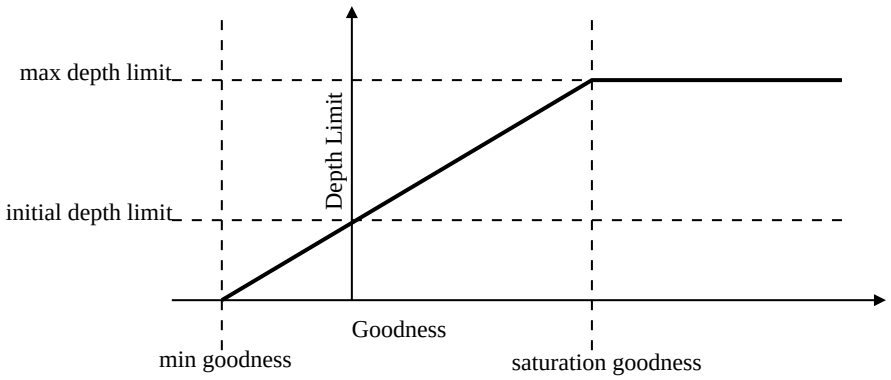


Figure 9.1: The speculation level for a let is a function of its recorded goodness

9.1.2 The Goodness Map Π

In the formal semantics, the goodness map Π is a function that maps each $\mathbf{let}$ identifier x to its integer goodness. In the implementation, the goodness map Π is implemented in the same way as the speculation configuration (Section 8.1.3). A static *goodness counter* variable is created for every $\mathbf{let}$, and the info table for a speculative return contains a pointer to this variable.

Rather than applying $goodToLim$ to the goodness every time a speculation depth limit is needed, the depth limit variables are updated only when the goodness might have changed. Figure 9.1 illustrates a typical $goodToLim$ function. This function is parameterised by four constants:

min goodness: How bad the goodness can get before the depth limit becomes zero and evaluation becomes entirely lazy.

saturation goodness: The level of goodness beyond which the speculation level stops increasing.

max depth limit: The maximum speculation depth limit that is allowed.

initial depth limit: The speculation limit corresponding to zero goodness. This is the speculation level that every $\mathbf{let}$ has when a program starts.[1]

The limit function does not have to be the shape illustrated in Figure 9.1. This is just one choice in a large design space. We have chosen to use this shape because it can be implemented efficiently. We discuss the effect of the tuning parameters in Section 12.3.5.

[1] Unless a persistent speculation configuration is used—see Section 9.4.3.

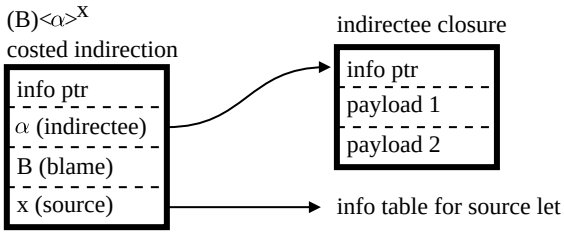


Figure 9.2: Heap representation of a costed indirection

9.1.3 Costed Indirections $B\langle\alpha\rangle^x$

In the formal semantics, a costed indirection describes the result of a previous venture:

$$B\langle\alpha\rangle^x$$

where B is the blame associated with the venture, α is a reference to the result produced by the venture, and x is the identifier for the $\mathbf{let}$ that spawned the venture.

Figure 9.2 illustrates the way that we represent a costed indirection in the heap. The first word points to a standard info table shared by all costed indirection closures, the second word contains α (a pointer), the third word contains B (an integer), and the fourth word contains a pointer to the info table for the $\mathbf{let}$ that spawned the venture (x). We see how such costed indirections are created in Section 9.2.2.

9.1.4 Profiled Speculation Frames $(\{x\}E, B)$

In the formal semantics, a profiled speculation frame is a stack frame of the following form:

$$(\{x\}E, B)$$

where x is the identifier for the $\mathbf{let}$ that spawned the venture, E is the body code for the $\mathbf{let}$, and B is the blame accumulated so far for the enclosing venture.

Figure 9.3 illustrates the way that profiled speculation frames are represented on the stack. The stack layout is the same as that of an unprofiled speculation frame, except that the info pointer points to the generic info table `profile_info` rather than to the info table for the $\mathbf{let}$. The info table for the $\mathbf{let}$ is stored on a separate *profile stack*, together with the blame B. While this representation is somewhat awkward, it allows the profiler to transform speculation frames between their profiled and unprofiled states without disturbing the rest of the stack. We see the usefulness of this in Sections 9.2.1 and 9.4.1.

Throughout the rest of this chapter, we will assume that the profile stack is a continuous block of memory with a register $\mathbf{ProfTop}$ pointing to the topmost frame. In practice it may be better to implement it as a linked list with cells allocated from the general heap.

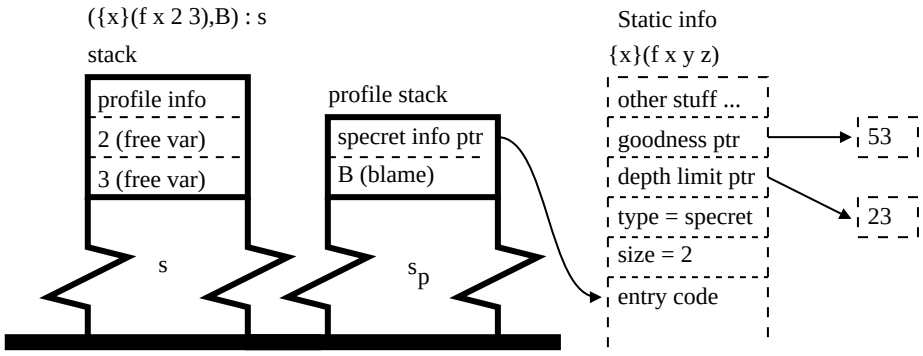


Figure 9.3: A profi led speculation frame contains its profi ling information on a special profi ling stack.

9.2 Implementing the Evaluation Rules

The evaluation rules for the profiling semantics of Section 6.4.2 are significantly more complex than those for the unprofiled semantics of of Section 4.2. In particular:

- Every rule that does work increments the blame count B
- The evaluation relation is parameterised by a profiling flag p.
- New rules are added to describe profiled speculation of a $\mathbf{let}$.
- New rules are added for demanding the value of a costed indirection.

In the subsections that follow, we describe how these features can be added to the implementation described in Chapter 8.

9.2.1 Starting a Profiled Speculation

The semantics of Section 6.4.2 has three rules that describe evaluation of a $\mathbf{let}$:

- *(LAZY)* describes lazy evaluation of a $\mathbf{let}$
- *(SPEC1N)* describes unprofiled speculation of a $\mathbf{let}$
- *(SPEC1P)* describes profiled speculation of a $\mathbf{let}$

As in Section 8.1.4, we replace rule *(LAZY)* with the rule *(LAZYX)* from Figure 9.4. This allows us to extend the code from Section 8.1.4 so that it performs a three way branch:[2]

[2]See Section 9.4.2 for a more effi cient way to implement this branch.

(LAZYX) $\Gamma; (\mathbf{let}\ x = E\ \mathbf{in}\ E'); s; B; \Pi \quad \longrightarrow_p \quad \Gamma[\alpha \mapsto E]; \triangledown\alpha; (\{x\}E' : s); B; \Pi$
if $goodToLim(\Pi(x)) \leq specDepth(s)$
and α is fresh

Figure 9.4: An alternative lazy rule for **let** expressions

Code to evaluate $\mathbf{let}\ x = E\ \mathbf{in}\ E'$:

$\mathbf{Sp} := \mathbf{Sp} - (1 + \text{env\_size})$	push a speculative return frame
$\mathbf{Sp}[0] := 3562\_\text{ret\_info}$	
$\mathbf{Sp}[1] := ?;\ \mathbf{Sp}[2] := ?; \ldots$	(save vars live in E')
$\mathbf{SpecDepth} := \mathbf{SpecDepth} + 1$	increment **SpecDepth**
$\mathbf{BlameBase} := \mathbf{BlameBase} - thunkcost$	add thunk cost to B
$\mathbf{branchif}\ (PROFILING)\ \text{maybe\_profile}$	three-way branch
$\mathbf{branchif}\ (\mathbf{SpecDepth} > 3562\_\text{limit})\ 3562\_\text{lazy}$	
$\mathbf{jump}\ 3562\_\text{spec}$	

If profiling is disabled, then the **let** will behave exactly as described in Section 8.1.4, except that it will add *thunkcost* to the current blame, to take account of the work the **let** does to allocate a thunk.[3] In the semantics, *thunkcost* is always 1; we explore the performance effect of different values for *thunkcost* in Section 12.3.3.

If profiling is enabled, then the **let** code will call the special runtime system function `maybe_profile`. If the **let** should be speculated, then `maybe_profile` jumps to `profile_start`, otherwise it jumps directly to the lazy code (in this case `3562_lazy`). It finds a pointer to this function in the info table for the **let**, which has conveniently been left on the top of the stack. We also take this opportunity to update the depth limit using the current goodness value:[4]

Code for `maybe_profile`:

$\mathbf{Sp}[0][limit][0] := goodToLim(\mathbf{Sp}[0][goodness][0])$	update depth limit
$\mathbf{branchif}\ (\mathbf{SpecDepth} \leq \mathbf{Sp}[0][limit][0])\ \text{profile\_start}$	profile this speculation
$\mathbf{jump}\ \mathbf{Sp}[0][lazycode]$	evaluate the **let** lazily

[3] Recall that we can add work to the current blame by subtracting it from **BlameBase** (Section 9.1.1).

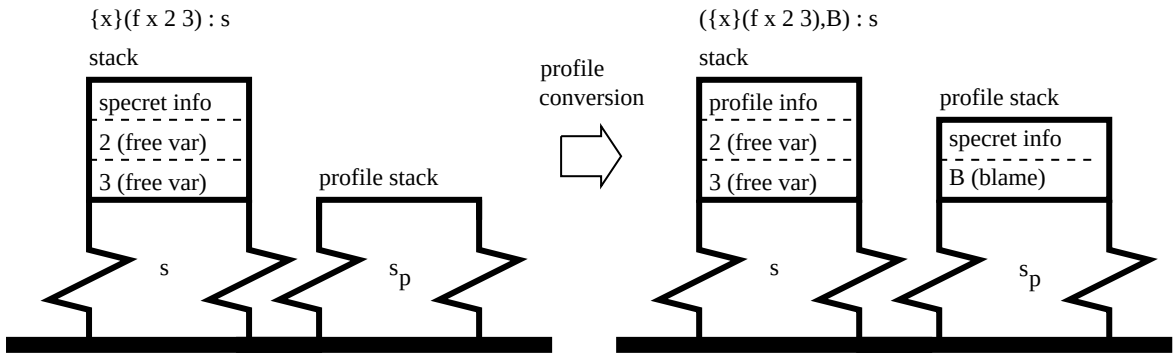


Figure 9.5: Speculation frames are converted into costed speculation frames

The function `profile_start` converts the end state of rule *(SPEC1N)* into the end state of rule *(SPEC1P)*, thus causing the speculation to be profiled. It does the following things:

- Adds $thunkcost$ to the local goodness. This represents the work saved by not performing a thunk.
- Converts the unprofiled speculation frame '$\{x\}E$' into the profiled speculation frame '$(\{x\}E, B)$'. This is illustrated in Figure 9.5.
- Sets the current blame B to zero.
- Jumps to the speculative code for the **let** (taken from the info table).

Code for `profile_start`:

$\mathbf{Sp}[0][goodness][0] :=$ $\mathbf{Sp}[0][goodness][0] + thunkcost$	add $thunkcost$ to goodness
$\mathbf{ProfTop} := \mathbf{ProfTop} - 2$	Convert to profiled frame
$\mathbf{ProfTop}[0] := \mathbf{Sp}[0]$	(specret info)
$\mathbf{ProfTop}[1] := \mathbf{Hp} - \mathbf{BlameBase}$	(B)
$\mathbf{Sp}[0] := \text{profile\_info}$	(overwrite stack info table)
$\mathbf{BlameBase} := \mathbf{Hp}$	set B to zero
$\mathbf{jump}\ \mathbf{ProfTop}[0][speccode]$	start speculating

[4]Recall that the *goodness* fi eld of an info table contains a pointer to a variable that contains the goodness, rather than the goodness itself (Section 9.1.2).

(SPEC2X) $\Gamma; \triangledown\alpha; ((\{x\}E, B') : s); \quad B; \Pi \longrightarrow_p \Gamma[\alpha' \mapsto B\langle\alpha\rangle^x]; \triangledown\alpha'; (\{x\}E : s); \quad B'; \Pi[x \mapsto \Pi(x) - B]$ where α' is fresh

Figure 9.6: An Alternative rule for completing a profi led speculation

9.2.2 Completing a Profiled Speculation

When a profiled speculation finishes, control will return to the generic `profile_info` entry point. This code will behave as described by rule *(SPEC2X)* in Figure 9.6. We can observe that rule *(SPEC2X)* followed by rule *(SPEC2N)* is equivalent to rule *(SPEC2P)*.

Rule *(SPEC2X)* wraps the returned reference in a costed indirection, restores the saved value of B, converts the profiled speculation frame back into an unprofiled speculation frame, updates the goodness counter, and then returns the newly created costed indirection to the unprofiled speculation frame:

Return code for `profile_info`:

$\mathbf{Hp}[0] := \text{blamedesc\_info}$	create a costed indirection $B\langle\alpha\rangle^x$
$\mathbf{Hp}[1] := \mathbf{Node}$	α (indirectee)
$\mathbf{Hp}[2] := \mathbf{Hp} - \mathbf{BlameBase}$	B (blame)
$\mathbf{Hp}[3] := \mathbf{ProfTop}[0]$	x (source)
$\mathbf{Node} := \mathbf{Hp}$	
$\mathbf{Hp} := \mathbf{Hp} + 4$	
$\mathbf{BlameBase} := \mathbf{Hp} - \mathbf{ProfTop}[1]$	restore blame to saved value
$\mathbf{Sp}[0] := \mathbf{ProfTop}[0]$	Convert to unprofiled frame
$\mathbf{ProfTop} := \mathbf{ProfTop} + 2$	
$\mathbf{Sp}[0][\mathit{goodness}][0] :=$ $\mathbf{Sp}[0][\mathit{goodness}][0] - \mathbf{Node}[2]$	subtract B from goodness
$\mathbf{jump}\ \mathbf{Sp}[0]$	return to unprofiled frame

9.2.3 Demanding a Costed Indirection

The entry code for a costed indirection behaves like rules *(CSTN)* and *(CSTP)*. To decide which of these rules to apply, the entry code must determine whether the current venture is being

profiled. It does this by comparing the number of frames on the profile stack with $\mathbf{SpecDepth}$. If we assume the existence of a profile stack base register $\mathbf{ProfBot}$ then we can obtain the number of frames on the stack by dividing the size of the stack by the size of a stack frame. The branch can be written as follows:

```
Entry code for a costed indirection:
    branchif ((ProfBot − ProfTop)/2 < SpecDepth)      not profiled
        costind_cstn
    jump costind_cstp                                 profiled
```

If the current venture is not being profiled, then the entry code jumps to `costind_cstn`, which behaves like rule *(CSTN)*. Note that this code is identical to that used for a normal indirection (Section 7.4.2):

```
Code for costind_cstn:
    Node := Node[1]        demand indirectee α
    jump Node[0]
```

If the current venture is being profiled, then the the entry code jumps to `costind_cstp`, which behaves like rule *(CSTP)*. Any blame in the costed indirection is transferred into the current venture. This blame is added to the goodness for the costed indirection's source, and the costed indirection is converted into a normal indirection. Once this is done, the costed indirection enters the indirectee α.

```
Code for costind_cstp:
    BlameBase := BlameBase − Node[2]        add B′ to current blame

    Node[3][goodness][0] :=                 add B′ to goodness
      Node[3][goodness][0] + Node[2]

    Node[0] := ind_info                     convert to a normal indirection

    Node := Node[1]                         enter indirectee
    jump Node[0]
```

In Section 9.4.1 we extend this implementation to support profile chaining.

9.3 Heap Profiling

The theory presented in Chapters 5 and 6 assumes that any difference in performance between Optimistic Evaluation and Lazy Evaluation is due to different amounts of work being done during evaluation. This neglects the effect of *heap residency*: the amount of reachable data that is present in the heap.

9.3.1 Why Heap Residency is Important

Garbage collection can take up a large proportion of a program's runtime (Section 12.6.4). It is thus important that we take account of any effects that Optimistic Evaluation might have on the cost of garbage collection. If the heap residency increases moderately, then the effects are not particularly dramatic; most modern garbage collectors reduce their collection frequency as the heap residency increases, ensuring that garbage collection takes up a fairly constant proportion of runtime [App87, Wil92]. However this approach breaks down if the heap size increases beyond the available physical memory. In this case, the program will slow down dramatically because the garbage collector must wait for the virtual memory system to fetch pages from disk.

9.3.2 Bounding Extra Heap Residency

We have implemented a crude mechanism that bounds the extent to which Optimistic Evaluation can increase the heap residency of a program. The garbage collector disables speculation completely if the heap residency rises above a user-defined limit, $MAXHEAP$. In the worst case, all heap allocation up to that point will have been due to speculation, and thus the maximum heap residency over the complete program run will be the residency that the program would have had under Lazy Evaluation, plus $MAXHEAP$.

The safety of this technique relies on the fact that Optimistic Evaluation cannot cause any further increases in residency once speculation has been disabled. If the right hand side of a **let** is evaluated speculatively, then every closure reachable from the result must either have been allocated during the speculation or have been reachable from the free variables of the expression evaluated. If the **let** had instead been evaluated lazily, then the thunk produced would have contained these same free variables. It thus follows that any extra closures made reachable by a speculation must have been allocated during that speculation.

Although this approach is effective, it is somewhat brutal. Speculation can be disabled even if it has not caused any increase in heap residency at all. It also incumbent on the user to set $MAXHEAP$ to an appropriate value, given the available memory on their machine. We believe that it should be possible to produce a better solution than this; however we leave the design of such a system for further work.

9.3.3 Blaming A Let for Extra Heap Residency

Rather than waiting for the mechanism described in Section 9.3.2 to disable speculation completely, it can be beneficial to punish **let** expressions that seem to be increasing heap residency.

One way to do this is to arrange that, whenever the garbage collector sees a costed indirection in the heap, it multiplies the attached blame by an appropriate scaling factor[5] and subtracts it from the goodness of the **let** that produced it. It should be noted that this technique is purely a heuristic and is not guaranteed to detect all additional heap residency. To see why, consider the following example:

$$\begin{array}{ll} \textbf{let } x = \text{allocate lots of heap} & \textbf{in} \\ \textbf{let } y = (\textbf{case } x \textbf{ of } P\ z \rightarrow \mathit{True}) & \textbf{in} \\ (x, 4) & \end{array}$$

When x is speculated, it allocates a large number of closures in the heap, all of which are reachable from its result. When y is speculated, it will demand x and take the blame for x's allocation. When the next garbage collection takes place, the garbage collector will not see any costed indirections: x's costed indirection will have been converted into a normal indirection, and y's costed indirection will not be reachable. The garbage collector will thus not be aware that speculation has increased the heap residency.

We can envisage several ways in which this heuristic could be improved. One way would be to arrange that, when a costed indirection is demanded, it is not converted into a normal indirection (Section 9.2.3), but is instead converted into a special form of indirection that records the fact that it was the result of a speculation. This would allow the garbage collector to calculate the amount of heap that was reachable only through the results of speculations (c.f. [RR96b]).

9.3.4 Heap Wasted by Laziness

If a **let** is evaluated lazily rather than speculatively then it may continue to hold onto free variables that would not have been referenced by the result of a speculative evaluation. It is very difficult to accurately measure the effect that this has [RR96a], however we have experimented with a rough approximation.

We extended the garbage collector to credit a **let** with a fixed amount of saved work every time a thunk for that **let** was garbage collected. The motivation for this idea was that, if there are a large number of thunks in the heap for a particular **let**, then it may mean that these thunks form a chain that would not have appeared under speculative evaluation (see Section 12.6). The profiler thus adds some saved work to the **let** that produced the thunks in an attempt to stop the chain growing longer. We discuss the performance of this technique in Section 12.6.5.

[5] Which we discuss in Section 12.6.5

$$\textit{(CSTN)}\quad \Gamma[\alpha \mapsto B'\langle\alpha'\rangle^x]; \circledcirc\alpha; s; \quad \longrightarrow_p \quad \Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha'; \mathit{chainProf}(B, s); \\ B; \Pi \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad B'; \Pi \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } \neg \mathit{profiled}(s)$$

Figure 9.7: Chain profi ling uses this revised version of *(CSTN)*

9.4 Further Details

In this section we explore various ways in which the implementation described in Sections 9.1 and 9.2 can be extended.

9.4.1 Profile Chaining

In Section 6.4.3 we presented the concept of *profile chaining*. Profile chaining replaces the *(CSTN)* rule with the rule given in Figure 9.7. Under this rule, if a costed indirection is used by an unprofiled speculation, it applies the function $\mathit{chainProf}$ to the stack, converting the innermost speculation frame into a profiled speculation frame.

The function $\mathit{chainProf}$ is implemented as a runtime system function that walks down the stack and converts the innermost speculation frame into a profiled speculation frame in the manner described in Section 9.2.1.

If profile chaining was implemented exactly as described in the semantics then it could cause a very large number of ventures to be profiled. Consider for example the following program:

$$\begin{array}{l} f\ x = \\ \quad \textbf{let}\ y = x + 1\ \textbf{in} \\ \quad \textbf{if}\ y > 1000\ \textbf{then}\ 4\ \textbf{else}\ f\ y \end{array}$$

If speculation of y is ever profiled, then chaining will cause every subsequent speculation of y to also be profiled. This will cause the program to run very slowly.

We avoid this problem by maintaining a limit on the number of chaining operations that any given unit of blame can pass through before chaining stops. A *chain count* field is added to every costed indirection and every profiled speculation frame. Every time a costed indirection is created, it will take on the chain count of the profiled speculation that created it. Every time $\mathit{chainProf}$ is applied, the chain count for the newly profiled speculation will be one greater than that of the costed indirection that it used. If the chain count rises above a defined *chain limit* then no further chaining operations will be applied and the costed indirection will revert to the behaviour described in Section 9.2.3.

We describe the performance effect of varying the chain limit in Section 12.3.7.

9.4.2 Encoding the Profile Flag in $\mathbf{SpecDepth}$

In Section 9.2.1 we used the following three way branch to decide how a **let** should be evaluated:

Code for a three way branch:

branchif $(PROFILING)$ maybe_profile — three-way branch
branchif $(\mathbf{SpecDepth} > 3562\_\text{limit})$ 3562_lazy
jump 3562_spec

In the common case of a **let** being speculated and unprofiled, this code will involve two branches. We can reduce this to one branch by encoding the profiling flag inside the speculation depth, giving a combined $\mathbf{SpecDepthProf}$ register:

$$\mathbf{SpecDepthProf} = \begin{cases} specDepth(s) & \text{if } p \text{ is off} \\ specDepth(s) + PROFILE\_ON & \text{if } p \text{ is on} \end{cases}$$

where $PROFILE\_ON$ is a large constant, greater than the maximum allowable speculation depth. Given this encoding, we can retrieve the speculation depth and profile flag as follows:

$$specDepth(s) = \mathbf{SpecDepthProf} \mod PROFILE\_ON$$

$$p = \begin{cases} \text{on} & \text{if } \mathbf{SpecDepthProf} > PROFILE\_ON \\ \text{off} & \text{otherwise} \end{cases}$$

This encoding makes it possible for us to combine the test for speculation and profiling as follows:

Branch code for a burst-profiled **let** expression:

branchif $(\mathbf{SpecDepth} \leq 3562\_\text{limit})$ 3562_spec
branchif $(\mathbf{SpecDepth} < PROFILE\_ON)$ 3562_lazy
jump maybe_profile

This code will detect the most common case, speculated and unprofiled, with only one branch.

9.4.3 Warming up the Profiler

Variable-Frequency Profiling

Rather than having a fixed profiling frequency, it can be beneficial for the profiler to vary its profiling frequency. Indeed our implementation varies its profiling frequency according to how confident it is about its speculation configuration. The profiler starts out with no confidence in its speculation configuration and so profiles every period. If a profiled period causes no significant changes to the speculation configuration then the profiler will increase its confidence and reduce its profiling frequency. If however a profiled period causes a significant change to the speculation configuration (e.g. it found lots of wasted work) or a speculation was aborted, then the profiler will fall back to profiling every period.

Warming up Gradually

Continuing this philosophy, no **let** is allowed to execute speculatively in an unprofiled period until it has been observed to behave well when being speculated during a profiled period. All **let**s start with a depth limit of zero. A **let** is not given a positive depth limit until the first time it is profiled. When the **let** is first profiled, the `maybe_profile` function will set the depth limit to the value corresponding to its goodness (See Section 9.2.1).

Persistent State

One way to reduce warm-up times is to make the speculation configuration for a program persistent. When a program completes, its speculation configuration is written into a file in the users home directory (e.g. `/home/rje33/.opteval/[progname]`). If the program is run again, then this speculation configuration will be reloaded and the warmup time can be avoided.

We implemented this persistent state scheme in a previous implementation of Optimistic Evaluation and found that it had relatively little effect on long-running programs, but significantly improved the performance of programs that run for a very short time.[6]

9.4.4 Representing Blame as a Cycle Count

Rather than measuring work by heap usage, another alternative is to measure the number of elapsed processor cycles. Some Intel processors provide an instruction called `rdtsc` [int97] that allows a program to discover the number of processor cycles that have elapsed since the processor was turned on.

[6] Unfortunately, it also upset several referees, who complained that, by using persistent state, our profiler could not really be considered to be an "online profiler". We thus decided to avoid giving benchmark results that used this feature, and did not include this feature in subsequent implementations.

We have produced a version of our profiler that measures work using `rdtsc` rather than with heap allocation. There are several difficulties with this approach. One particular problem is that it can be hard to distinguish between cycles that constitute the work done by a speculation, and cycles that took place in another process, if a context switch took place during the speculation. Another problem is that the number of cycles required for a speculation is extremely unpredictable, making it quite hard for to debug the implementation or to track changes to its performance. We discuss the performance of `rdtsc` profiling in Section 12.7.2.

CHAPTER 10

Abortion

In this chapter we describe our implementation of abortion:

- In Section 10.1 we describe the way in which the runtime system decides *when* a venture should be aborted.
- In Section 10.2 we describe *how* a venture is aborted.

10.1 When to Abort

It is important that our runtime system limits the amount of time that a speculation can run for before it is aborted. In this section we describe the way in which our implementation does this.

10.1.1 Sample Points

In Section 6.3.1 we presented a bounded speculation semantics that places a limit on the amount of blame that can be accumulated by active speculations. In Section 6.3.3 we argued that this semantics allows us to bound the worst case performance of a program.

The semantics of Section 6.3.1 assumes that all speculations are being profiled; however the periodic profiling scheme described in Chapter 9 will only profile a small proportion of speculations. We work around this problem by ensuring that all long-running speculations are profiled. At periodic sample points, the runtime system walks down the stack and converts all unprofiled speculation frames into profiled speculation frames. This uses the same mechanism as described in Section 9.2.1. All profiled speculation frames created in this way will start off being blamed for no work and will accumulate blame from that point onwards.

The runtime system also uses this opportunity to check that the sum of all blame attributed to active profiled speculations is less than $MAXBLAME$, using a literal implementation of the $activeBlame$ function from Section 6.3.1.

10.1.2 Heap/Stack Checks

One detail of GHC that we have ignored so far is heap and stack checks. As we described in Section 7.3.1, the GHC runtime maintains a $\mathbf{Hp}$ register that points to the next free word in the heap. Whenever the program wishes to create a closure in the heap, it allocates it a range of addresses starting at $\mathbf{Hp}$ and then adds the size of the closure to $\mathbf{Hp}$, so that $\mathbf{Hp}$ points to the next free word once again. Unfortunately, heap is not an infinite resource; it is thus important that a program check that there is heap available before it allocates a new closure. The same issues apply to the stack pointer $\mathbf{Sp}$; it is essential that the program checks that there is stack available before it attempts to push a new frame onto the stack.

In the code we gave in Chapters 7, 8, and 9 we omitted such heap and stack checks for reasons of simplicity. However, a real program will include heap and stack checks at the beginning of the entry code for every closure, return frame, or function. The stack check will compare $\mathbf{Hp}$ and $\mathbf{Sp}$ against the limit registers $\mathbf{HpLim}$ and $\mathbf{SpLim}$. If the heap and stack pointers are within safe limits, then the program will continue; however, if the heap and stack pointers are outside safe limits then the program will call into the runtime system so that more heap or stack can be allocated.

It is at these points that the runtime system ensures that all active speculations are profiled, and checks to see if abortion should take place.

10.1.3 Paused States

Before the program calls into the runtime system, it first puts its state into a standard form known as a *paused state*. A paused state is one that can be correctly resumed by entering the closure pointed to by the $\mathbf{Node}$ register (known as the *current closure*). In the formal semantics, a paused state is a state of the following form:

$$\Gamma; \circledcirc\alpha; s$$

where α is a reference to the *current closure*.

If a heap/stack check is at the beginning of the entry code for a closure, then the program is already in a paused state, and so the program can call into the runtime system directly.

Things are slightly more complicated if the heap/stack check is at the beginning of the entry code for a return frame. If Lazy Evaluation is implemented in the manner described in Section 8.1.4 then it is not safe to simply replace a return command $\triangledown\alpha$ with a closure

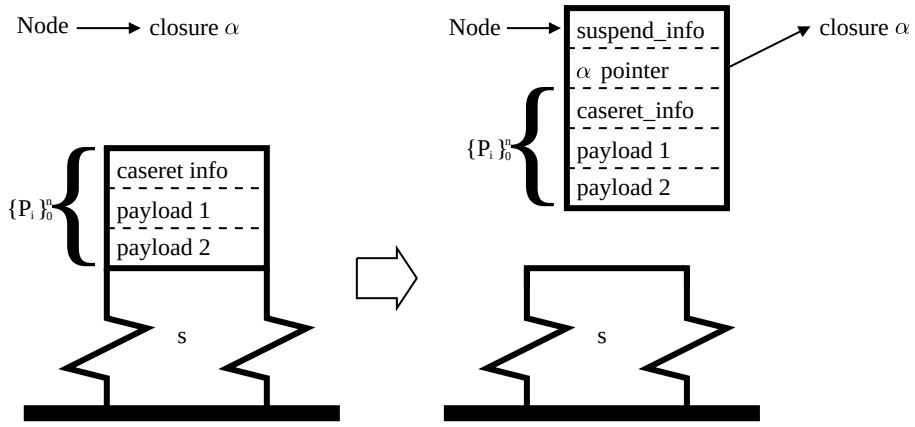


Figure 10.1: Moving a stack frame into a suspension closure

demanding command $\circledcirc\alpha$ as α may reference a thunk that should not be evaluated. We thus push a special frame onto the stack that will return α when it is returned to, and set $\mathbf{Node}$ to point to a generic value closure that will return immediately when entered.

If a heap/stack check is at the beginning of the entry code for a function then the program can pause itself by pushing a function application frame containing its arguments and setting $\mathbf{Node}$ to point to the function that was just entered.

10.2 How to Abort

In this section we describe how the runtime system actually goes about aborting a speculation. The implementation described here has much in common with the implementation that already existed in GHC for the implementation of asynchronous exceptions [MPMR01]. In the subsections that follow, we describe our implementation of abortion by comparing it to the abortion semantics of Section 6.3.2.

10.2.1 Pause a State: *(!EXP)* and *(!RET)*

As we described in Section 10.1.3, a program will pause itself before entering the runtime system. The abortion system thus does not need to implement an equivalent of *(!EXP)* or *(!RET)*.

10.2.2 Suspend a Stack Frame: *(!ABORT)*

If the topmost stack frame is not a speculation frame or an update frame, then the contents of the stack frame is transferred into a *suspension closure* in the heap. Figure 10.1 illustrates the way that a suspension closure $\alpha\angle l$ is implemented. The info table is the standard info table

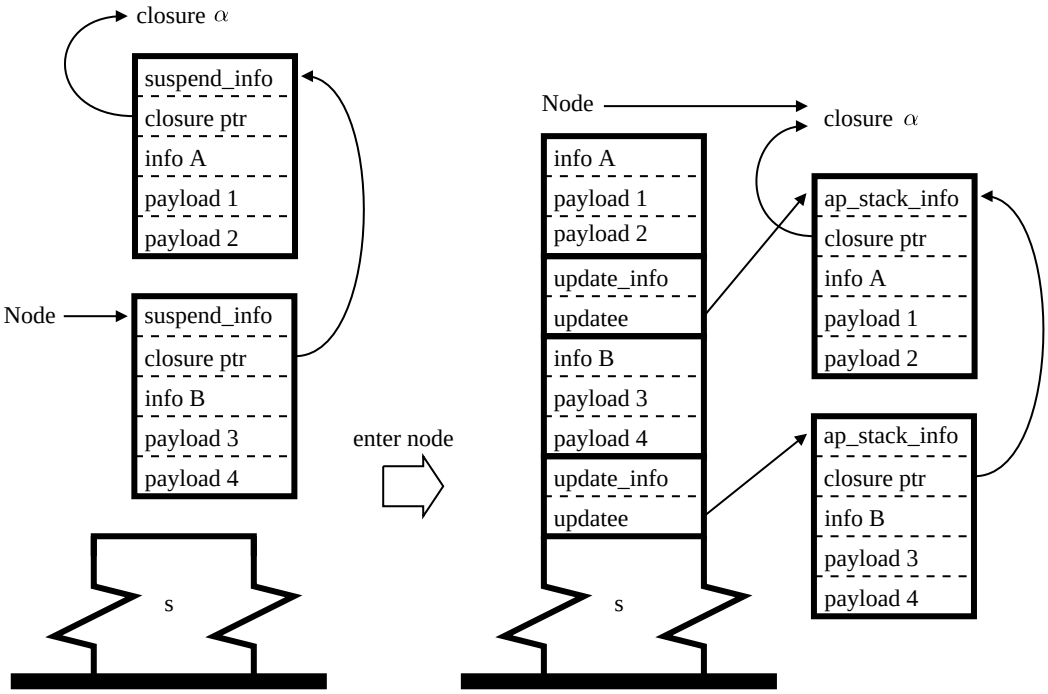


Figure 10.2: Entering a suspension closure causes its contents to be copied onto the stack.

`suspend_info`, the second word points to the returned closure α, and the rest of the payload contains the stack frame l.

The real implementation is rather more complex than this. In particular, it will group several sequential stack frames together into one suspension closure rather than creating a new suspension for every frame.

10.2.3 Resume a Suspended Stack Frame: *(RESUME)*

If a suspension closure $\alpha \angle l$ is entered, then the entry code for the suspension closure will push an update frame, copy its suspended stack frame l back onto the stack, and then enter the closure α. Abortion may create chains of suspension closures, linked together by their α fields. When the suspension at the start of the chain is entered, this will cause all of the suspensions in the chain to copy their frames back onto the stack. This is illustrated by Figure 10.2.

10.2.4 Abort an Update Frame: *(!UPD)*

If the topmost stack frame is an update frame then the abortion system behaves like rule *(!UPD)*. The closure referenced by the updatee α' is overwritten with an indirection to the current closure α. This is illustrated by Figure 10.3.

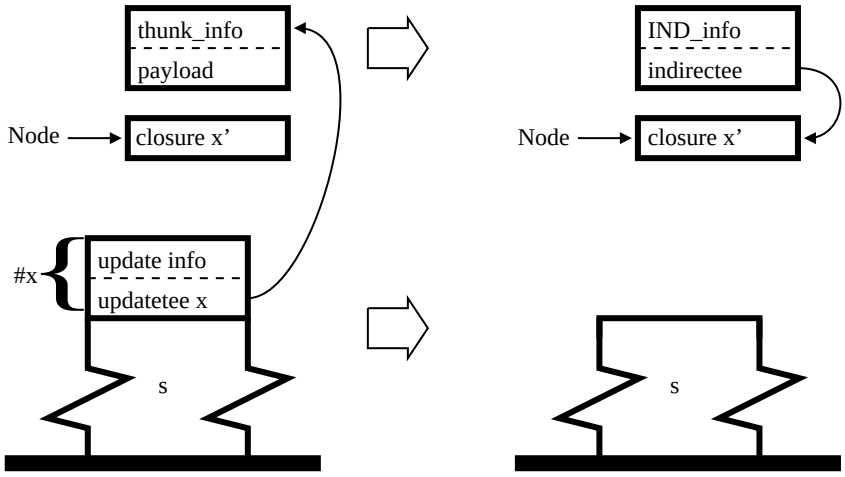


Figure 10.3: Aborting an update frame

If this is the outermost speculation frame we wish to abort:

(!END) $\Gamma; \circledcirc\alpha; (\{x\}E, B') : s; \quad \rightsquigarrow \quad \Gamma; \triangledown\alpha; (\{x\}E, B') : s;$
$B; \Pi \qquad\qquad B; \Pi[x \mapsto \Pi(x) + B_{\text{abort}}]$

If we wish to abort other speculation frames on the stack:

(!MORE) $\Gamma; \circledcirc\alpha; (\{x\}E, B') : s; \quad \rightsquigarrow \quad \Gamma[\alpha' \mapsto B\langle\alpha\rangle^x, \alpha'' \mapsto E[\alpha'/x]]; \circledcirc\alpha''; s;$
$B; \Pi \qquad\qquad B'; \Pi[x \mapsto \Pi(x) + B + B_{\text{abort}}]$
where α' and α'' are new

Figure 10.4: Alternative rules for aborting a speculation frame

10.2.5 Abort a Profiled Speculation: *(!SPEC)*

The way the abortion system treats a speculation frame depends on whether the speculation frame is the last speculation frame that needs to be aborted. The abortion system behaves as if rule *(!SPEC)* was replaced with rules *(!END)* and *(!MORE)* from Figure 10.4. Neither of these rules change the behaviour of abortion. Rule *(!SPEC)* is equivalent to *(!END)* followed by rule *(SPEC2)* from the semantics of Section 6.4.2. Similarly, rule *(!MORE)* is equivalent to rule *(!SPEC)* followed by rule *(!EXP)*.

If the topmost stack frame describes the outermost speculation that needs to be aborted then the abortion system will behave like rule *(!END)*. It will subtract B_{abort} from the goodness and then return to the body of the speculated $\mathbf{let}$. This will cause the program to continue with whatever it was planning to do once the speculation had finished.

If there are other speculation frames below this one that also need to be aborted then the abortion system will behave like rule *(!MORE)*. It will update the goodness and create a costed indirection in the same way as would be done if the speculation had completed normally (Sec-

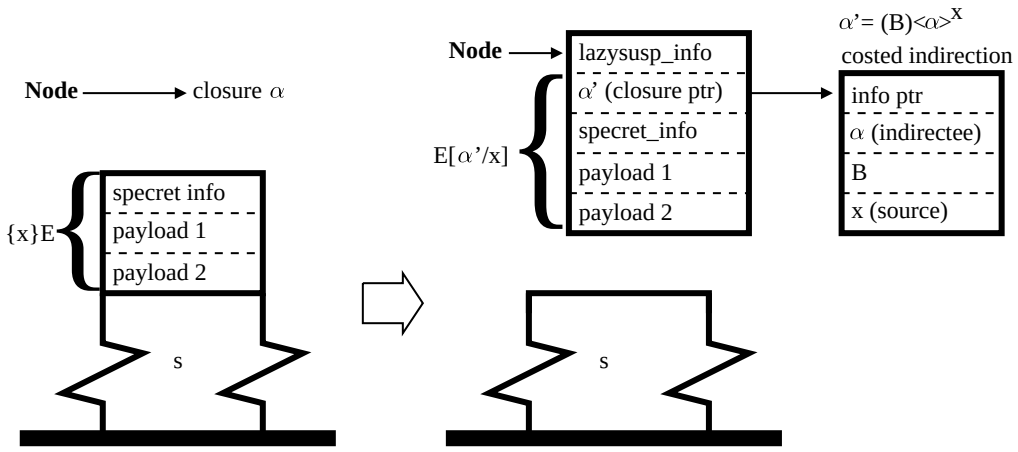


Figure 10.5: Aborting a speculative return frame

tion 9.2.2). It will then add B_{abort} to the blame to take account of the work done during abortion. Finally, it will create a thunk α'' in the heap, and set this to be the current closure.

When α'' is entered, it will push an update frame onto the stack and then evaluate the body $E[\alpha'/x]$ of the **let**. We illustrate the structure of α'' in Figure 10.5. The closure α'' is very similar to a suspension frame, as described in Section 10.2.2. The info pointer points to the special info table `lazy_susp_info`, the second word holds the costed indirection reference α' and the other words hold the speculative return frame $\{x\}E$.

If `lazy_susp_info` is entered, it will push an update frame, copy the speculation frame back onto the stack, and then return α' to it. The behaviour is thus the same as if α'' was a thunk whose body was $E[\alpha'/x]$. Note that, unlike a stack suspension, α'' will not demand the value of α'.

Every time a speculation frame is removed from the stack, the speculation depth is reduced. We thus take case to decrement $\mathbf{SpecDepth}$ whenever we remove a speculation frame from the stack, and to increment $\mathbf{SpecDepth}$ whenever we restore a speculation frame back onto the stack. In particular, the code for a lazy suspension must increment $\mathbf{SpecDepth}$ before copying its speculation frame back onto the stack, even though it returns to the speculation frame immediately.

CHAPTER 11

Debugging

Debugging has long been recognised as one of the greatest weaknesses of lazy functional languages [Wad98]. Conventional (strict, imperative) languages almost invariably use the "stop, examine, continue" paradigm (Section 11.1), but this approach does not work well for Lazy Evaluation. This difficulty has led to fascinating research in novel debugging techniques (Section 13.7).

In this chapter, we argue that conventional debugging techniques have perhaps been dismissed too quickly. We demonstrate that Optimistic Evaluation, combined with *transient tail frames* allow conventional debugging techniques to be successfully applied. Optimistic Evaluation significantly reduces the number of thunks built, and thus also their confusing affect on debugging, while transient tail frames allow tail-calls to be visible to the debugger without affecting space complexity (Section 11.3).

We have implemented these ideas in HsDebug, an addition to the GHC tool set (Section 11.5). Our debugger is, by design, "cheap and cheerful". Its results are not as predictable, nor as user-friendly, as those of (say) Hat [WCBR01]—but they come cheap. HsDebug can debug an entirely un-instrumented program, and it can do a lot better if the compiler deposits modest debug information (much like a conventional debugger). Furthermore, an arbitrary subset of the program can be compiled with debug information—in particular, the libraries need not be.

11.1 How the Dark Side Do It

A debugger has long been one of the standard tools that is provided with any strict, imperative, programming language implementation. The vast majority of these debuggers follow a "stop, examine, continue" model of debugging, as used by GDB [SP91]. Such debuggers are characterised by the following features:

- The programmer can request that execution stop at a *breakpoint*. A breakpoint may correspond to a point in the source code of the program. Alternatively, it may be the point at which some logical property becomes true.

- When a program is in its stopped state, the programmer can examine the state of the program. From this state, the programmer is able to obtain an understanding of how the program came to be in the state that it is.

- The programmer can call functions within the program and can directly manipulate the program state.

- Once the programmer has finished examining the state and adjusting their breakpoints, they can request that execution continues until the next breakpoint is hit.

One of the most important pieces of information that a debugger allows the programmer to observe is the call stack. In a strict language, the nesting of frames on the call stack will correspond directly to the nesting of function calls in the program source code. Consider the following program:

$$f\ x = \quad \mathbf{let}\ y = 3\ \text{`div'}\ x\ \mathbf{in}\ \mathit{Just}\ y$$

$$g = \quad \mathbf{case}\ f\ 0\ \mathbf{of} \quad \mathit{Just}\ y \rightarrow h\ y$$

If this program is executed eagerly, then the call stack will be as illustrated in Figure 11.1. When the division by zero error occurs, the stack will clearly show that this took place inside the evaluation of y, in the call to f made from g. It is likely that the stack will also hold the argument that f was called with.

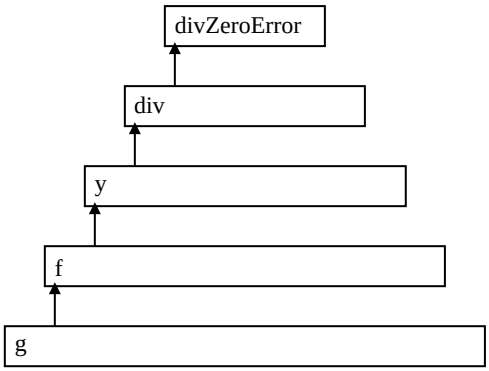


Figure 11.1: Strict Evaluation

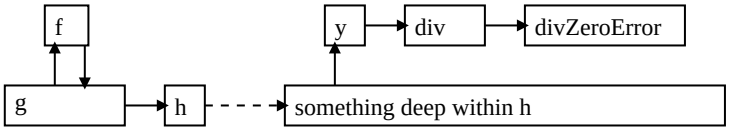


Figure 11.2: Lazy Evaluation with Tail Call Elimination

11.2 Failing to Debug Lazy Programs

What happens if we try using this style of debugging for a non-strict functional language such as Haskell [Pey03]? If the same program given earlier were to be evaluated in a typical lazy language implementation, then the call stack would be as illustrated in Figure 11.2. Lazy Evaluation has scrambled the execution order and tail call elimination has removed stack frames that would have provided useful information. The result is a mess that is very difficult to debug.

This clash between "stop, examine, continue" debugging and non-strict languages is considered to be so severe that, as far as we know, nobody has ever made a serious attempt to implement such a debugger for a non-strict language. In many ways, this rejection of conventional debugging models has been a good thing, as it has led to the development of several extremely powerful alternative approaches (see Section 13.7). However, we believe that conventional debugging techniques should not be written off. In the sections that follow, we explain a series of tweaks to Lazy Evaluation that have allowed us to produce an effective "stop, examine, continue" debugger for Haskell.

11.3 Eliminating Tail Call Elimination

One simple way to increase the amount of information available from stacks is to disable tail-call elimination. This provides us with extra stack frames that tell us more about the call chain that has taken place. This idea is not new; strict languages that implement tail call elimination often allow it to be disabled when a program is being debugged. For example, the CMU Com-

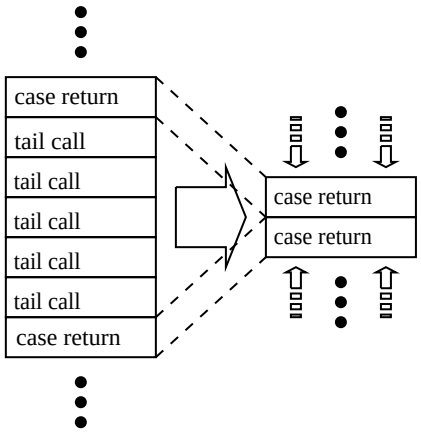


Figure 11.3: Tail call frames disappear if we have too many

mon Lisp [Mac92] environment disables tail call elimination if the debug level is set greater that "2".

While turning off tail call elimination makes debugging easier, it will also cause some programs to use massively more stack space than they would otherwise use. A tail recursive loop that would normally consume no stack space will now push a stack frame on every iteration.

Fortunately, there is no need for a program to fail with an "out of stack" error if it has a stack full of tail call return frames. The only purpose of tail call return frames is decorative; it is thus perfectly okay to delete them. Our solution is thus to delete all tail frames every time we run out of stack or perform a garbage collection. Figure 11.3 illustrates this concept.

It is important that we delete tail frames at a garbage collection, even if we are not short on stack. This is because our tails frames record all the arguments passed to a call, and so may be holding onto heap objects that would otherwise not be reachable. There are many ways in which this could be refined: for example, we could arrange to only delete tail frames if stack use or heap residency were above a pre-determined threshold, or we could arrange to only delete a selected subset of the tail frames.

11.4 Optimistic Evaluation

Disabling tail call elimination makes debugging significantly easier, but it does not quite bring us to the point at which a "stop, examine, continue" debugger becomes usable. Lazy Evaluation will still scramble the evaluation order, causing expressions to be evaluated on stacks that are different to the stack in which the expression was defined.

Fortunately, Optimistic Evaluation causes a program to be evaluated largely eagerly, and so significantly reduces the scrambling effect of Lazy Evaluation. When a program is being debugged, it is likely that the programmer cares less about speed and more about clarity, compared to a normal execution. Optimistic Evaluation thus uses different default tuning parameters when

running in debug mode, causing it to have worse performance, but also causing it to use Lazy Evaluation less often.

Although we have implemented our debugger on top of Optimistic Evaluation, the same techniques could also be applied to Eager Haskell (Section 13.2).

11.5 HsDebug

HsDebug is a "stop, examine, continue" debugger for Haskell. It has been implemented as part of GHC and currently lives on the Optimistic Evaluation branch in the GHC public CVS. While HsDebug has a long way to go before it becomes as powerful a tool as GDB, it is already very useful. The current feature set includes the following:

- Any program compilable with GHC can be debugged
- Breakpoints can be set in any Haskell function
- The original arguments of all calls on the stack can be inspected
- Closures on the heap can be pretty printed
- Functions and thunks can be pretty printed—giving their source location, and free variables.
- Exceptions can be intercepted
- The program can be single-stepped

All in all, HsDebug feels very similar to GDB and should feel familiar to anyone who is already comfortable with GDB. HsDebug is currently very rough round the edges, and source level debugging is currently incomplete, but it has already shown itself to be a useful tool.

Programs compiled for HsDebug run slightly slower than normal. This is partly due to the need to turn off some of the more confusing code transformations and partly due to the extra overhead of pushing and removing tail frames. We discuss performance further in Section 12.7.4.

We give a log from a real HsDebug session in Appendix D.

Part IV

Conclusions

CHAPTER 12

Results

How well does Optimistic Evaluation work in practice? What is the performance effect of adjusting the many tunable parameters? In this chapter we attempt to answer these questions.

- We start, in Section 12.1, by explaining how the tests described in this chapter were carried out.
- In Section 12.2 we analyse the performance of our fastest version of Optimistic Evaluation.
- In Section 12.3 we look at the online profiler, and explore the performance effect of several changes that can be made to it.
- In Section 12.4 we present various statistics that give an insight into the way that Optimistic Evaluation works.
- In Section 12.5 we look at semi-tagging, and analyse its effect on both Optimistic Evaluation and Lazy Evaluation.
- In Section 12.6 we look at the effect of Optimistic Evaluation on heap usage.
- Finally, in Section 12.7, we collect together various statistics that do not seem to belong anywhere else.

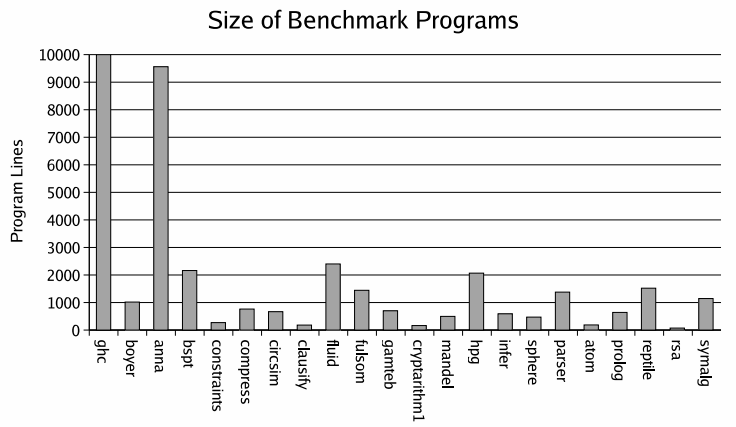


Figure 12.1: Sizes of Benchmark Programs

12.1 How the Tests were Carried Out

In this section we explain how we carried out our tests. Section 12.1.1 discusses the benchmark programs we used, while Section 12.1.2 explains the way in which we ran them.

12.1.1 The Benchmarks

The benchmarks programs we chose are a selection of the programs from the NoFib [Par92] benchmark suite. NoFib is the standard benchmark suite for Haskell compilers and the programs from this set have been used to evaluate the performance of a number of other systems. Most of the programs we use are from the `real` subset of the suite, with a few taken from the `spectral` subset. For some graphs we have also added the GHC compiler itself. While GHC is not part of NoFib, it is probably the biggest and most widely used Haskell program in existence and so it is an interesting program to test.

Figure 12.1 shows the size of each of our benchmark programs. GHC is huge, at around 150,000 lines of code. `anna` is also fairly large, at 9,000 lines of code. The other programs are significantly smaller, with an average of 935 lines of code each. Despite their small sizes, many of these programs are quite realistic; indeed several of them are the kernels of large applications.

The default runtimes for the NoFib programs are very variable and typically very short. We extended the input data for our chosen programs so that they all run for around fifteen seconds. This gives our online profiler sufficient time to warm up and so gives a fairer impression of performance. We could have achieved similar performance on smaller benchmarks by using a persistent speculation configuration, however we chose not to do this for the reasons outlined in Section 9.4.3.

The programs we used as benchmarks were selected before any performance results had been obtained for them. Programs were selected according to the ease with which we could

make them run for around fifteen seconds, and not based on any other criteria.[1]

12.1.2 The Testing Procedure

All tests were run on a machine with a 750Mhz Pentium III processor and 256Mbytes of memory. We ran each test once as a warm-up and then five more times. The published results are the average of the five test runs. In order to makes the performance more consistent, we disabled pre-emptive thread switching (using the `-C0` flag to the runtime system).

All runtimes were measured using the `gettimeofday` system call under Linux. When tests were run we ensured that no other software was running on the machine.

Performance results are given as a *relative runtime*. This is the runtime of a benchmark, expressed as a percentage of the runtime of the same benchmark with a different evaluator. In most cases, runtimes will be expressed relative to our best performing version of Optimistic Evaluation. If runtimes are expressed relative to a different evaluator, then this will be stated in the subtitle of the graph.

12.2 Performance

12.2.1 Runtime

Figure 12.2 shows the effect that Optimistic Evaluation has on run time. The height of a column in the graph represents the amount of time that that benchmark took to run under Optimistic Evaluation, expressed as a proportion of the time that the benchmark took to run under normal GHC. The GHC compiler that we compare against is the version that our implementation of Optimistic Evaluation forked from. Both compilers were run with all optimisations enabled, including a strictness analyser.

These results are very encouraging. The average speedup is just under 20% and several programs speed up by 40% or more. Perhaps more critically, no program slows down by more than 7%; indeed only one program slows down by more than 1% and only three programs slow down at all. We explore the reasons for these slowdowns in Section 12.2.2.

As one would expect, the results depend on the nature of the program. If a program has a strict inner loop that the strictness analyser solves, then we have little room for improvement. Similarly, if the inner loop in inherently lazy, then there is nothing we can do to improve things, and indeed the extra overhead of having a branch on every `let` will slow things down. In the case of `rsa` Optimistic Evaluation had virtually no effect because `rsa` spends almost all of its time inside a library written in `C`.

[1] The only exception to this is `fem` which we decided not to use because its performance was very variable when evaluated using normal GHC.

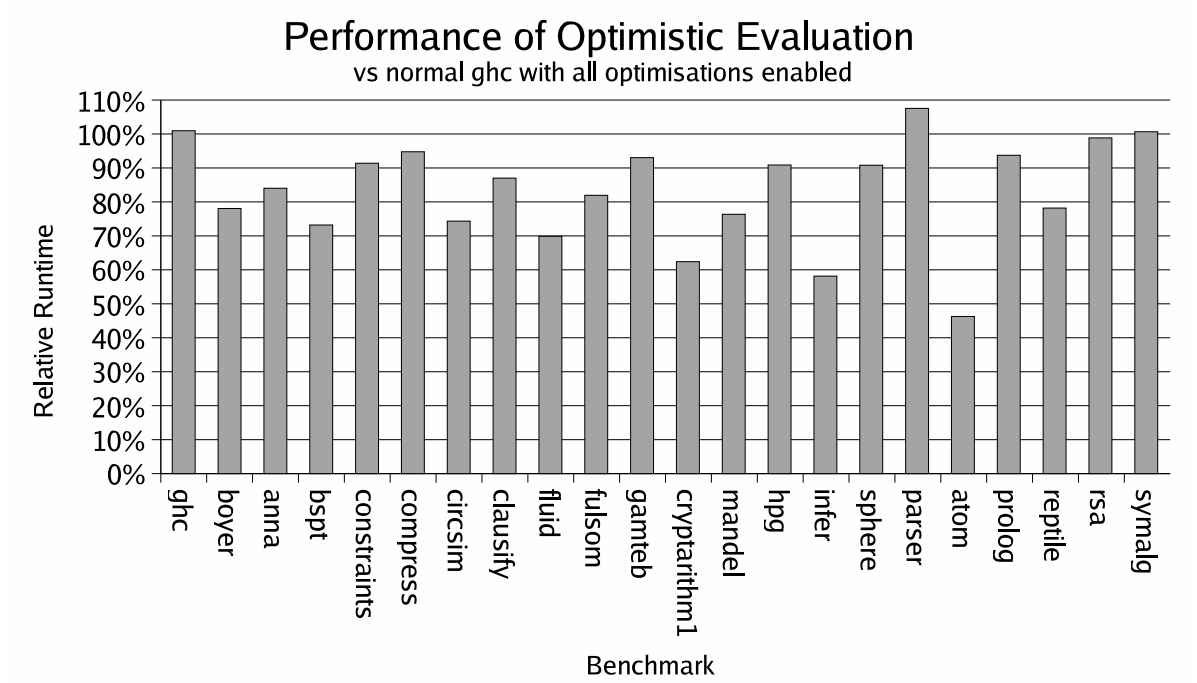


Figure 12.2: Benchmark run-time relative to normal GHC

The observed performance improvement is significantly greater than the error margin of the recorded runtimes. Of the 5 runs done for each program, the maximum recorded deviation from the mean runtime was below 1% in all but 5 benchmarks, and the geometric mean of the maximum deviations was only 0.56%.

12.2.2 The Overheads of Optimistic Evaluation

Optimistic Evaluation imposes considerable overheads on evaluation. The greatest of these is the cost of doing a test and branch on every `let` expression, but the cost of profiling is also significant. Figure 12.3 shows the overhead imposed by these features. For each benchmark, the respective column shows the extent to which that program slows down if it is compiled for Optimistic Evaluation, but is evaluated with a speculation configuration that evaluates everything lazily. For convenience, the origin of this graph is placed at 100%. The height of a column is thus the additional time taken.

As one would expect, all programs run more slowly than under normal GHC. The program that runs slowest is `parser`; slowing down by over 30%. Given this poor baseline, the profiler did well to achieve only a 7% slowdown for `parser` in Figure 12.2.

12.2.3 Code Size

Figure 12.4 shows the effect that Optimistic Evaluation has on code size. Programs increase in size significantly (36% on average). This is partly due to the need to generate lazy and eager

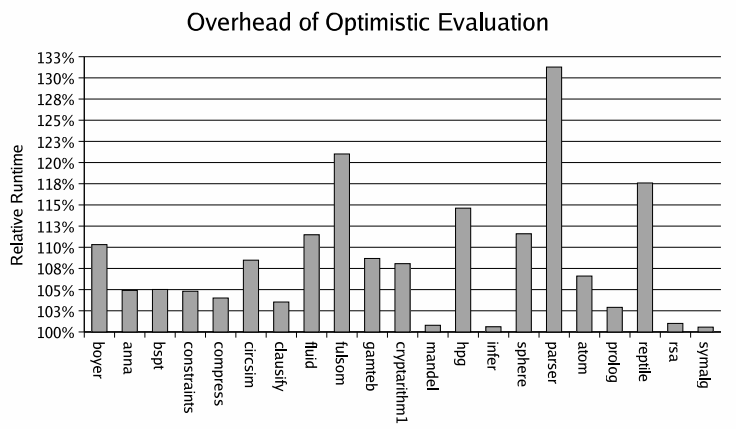


Figure 12.3: Performance Overhead of Optimistic Evaluation, when Evaluating Entirely Lazily

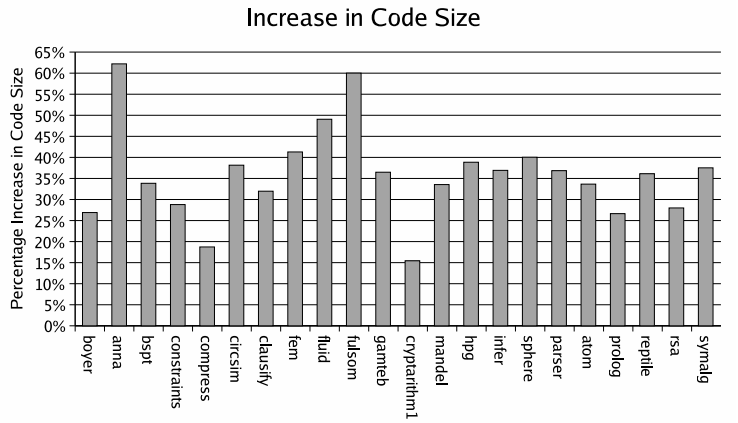


Figure 12.4: Increase in Code Size due to Optimistic Evaluation

versions of expressions, partly due to the effect of semi-tagging, and also partly due to various aspects of the implementation that we have not described in this thesis (in particular, the way we deal with vectored returns). We believe that this code bloat can be significantly reduced, but have yet to demonstrate this.

12.3 Profiling

12.3.1 What Happens if We Turn Profiling Off?

Figures 12.5 and 12.6 illustrate the performance effect of turning off profiling. The height of each column represents the amount of time taken when profiling is turned off, relative to the amount of time taken when it is turned on. Although profiling is turned off, abortion is still enabled; the evaluator thus behaves similarly to Eager Haskell (see Section 13.2).

As one can see, profiling is not always a win. Two programs speed up significantly if profiling is turned off; these are programs for which the profiler was being overly cautious.

Similarly, four programs speed up slightly; these are programs for which the profiler was not needing to do much and so was just wasting time.

At the other end of the spectrum, some programs slow down massively if the profiler is not enabled. `constraints` slows down by a factor of over 150, while `fulsom`, `hpg` and `infer` also slow down by large amounts.

Figure 12.7 shows the performance of unprofiled Optimistic Evaluation relative to normal GHC. Although the average performance is a 34% slowdown, most of this is due to four programs. If we exclude the four slowest programs then we get an average speedup of 13%.

12.3.2 How Often should We Profile?

Although our idealised model considers profiling to be random, our implementation is actually periodic.[2] The rate of profiling is governed by a variable called the *skip count*. The *skip count* is the number of unprofiled periods that take place between consecutive profiled periods (Illustrated by Figure 12.8). If the skip count is zero, then all periods are profiled.

As we explained in Section 9.4.3, our profiler profiles a program at a variable rate. The profiler thus has two variables that can be adjusted:

- **Max Skip Count:** The maximum allowed value for the skip count.
- **Skip Count Increment:** The amount that is added to the skip count if a profiled period passes without any adjustment needing to be made to the speculation configuration.

Figure 12.9 shows the effect of varying the maximum skip count. The vertical axis is the geometric mean of the relative runtimes of all the benchmark programs. Runtimes are taken relative to the case in which the maximum skip count is 200 (the default value). We can see from this graph that very low skip counts cause the program to run very slowly, because a large proportion of time is taken up with profiling. If the program is profiled constantly (skip count of zero) then the average performance is reduced by over 25%.

Figure 12.10 gives more detail. In this graph a separate curve is drawn for each benchmark. There are too many benchmarks for it to be practical to give a key; however this graph should give an impression of the general spread of results. We can see from this graph that most programs have fairly consistent performance once the maximum skip count is greater than 32. The exceptions are `bspt` and `parser` which can slow down by as much as 11% for some values.

Figure 12.11 shows the effect of varying the skip count increment. The vertical axis is again the geometric mean of the relative runtimes of all the benchmark programs, this time taken relative to the case in which the skip count increment is 3. There is less of a definite pattern here; no reasonable value of skip count increment varies the performance by more than 2.5%.

[2] Given that the profiling frequency is constantly varying, we believe that it is unlikely that any program will have a long term correlation with the profiler's sampling behaviour.

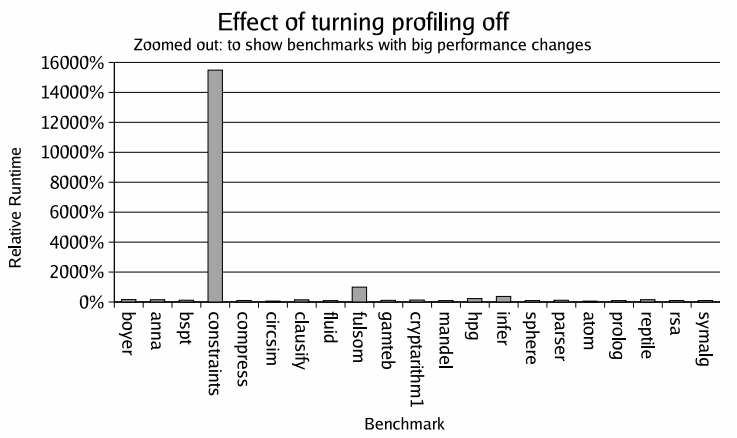


Figure 12.5: Performance Effect of Turning Profi ling Off : Zoomed Out

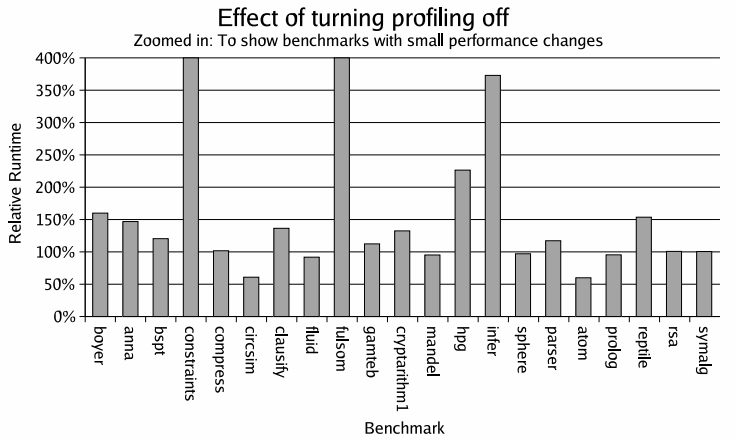


Figure 12.6: Performance Effect of Turning Profi ling Off : Zoomed In

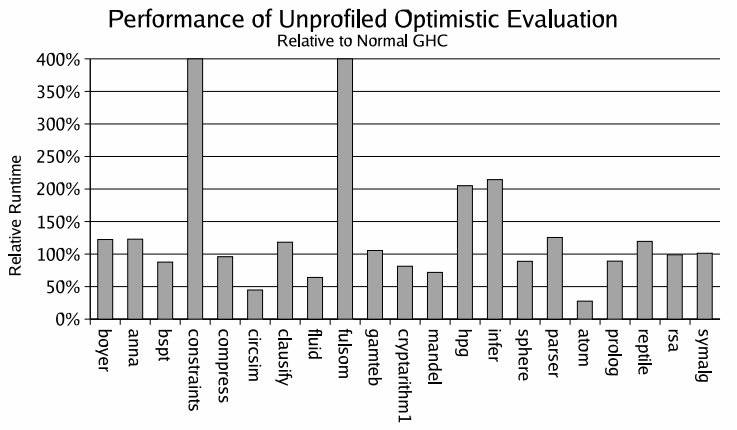


Figure 12.7: Performance of Unprofi led Optimistic Evaluation, relative to Normal GHC

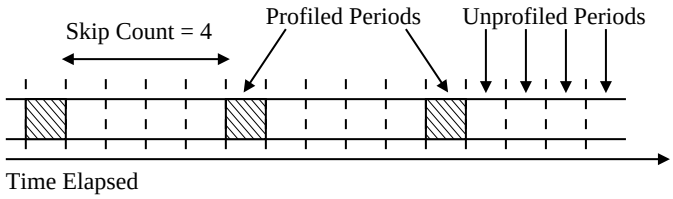


Figure 12.8: The *Skip Count* is the number of unprofi led periods that take place between consecutive profi led periods

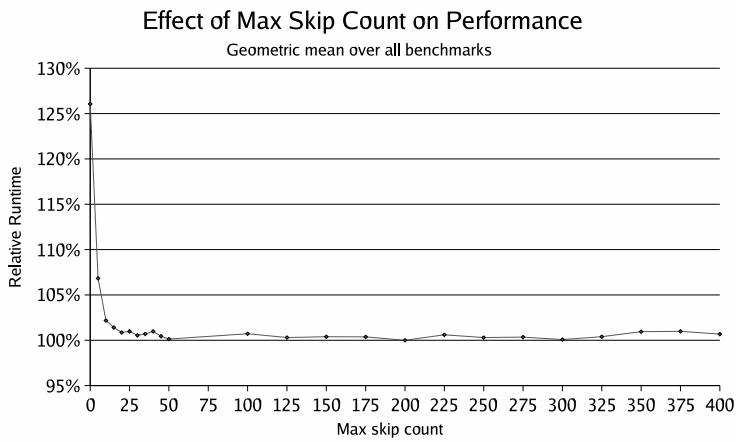


Figure 12.9: Performance Effect of Adjusting the “max sample period” Tunable—geometric mean

12.3.3 What should the Thunk Cost Be?

The *thunk cost* is the cost of building a thunk, as estimated by the profiler. The profiler will add this amount to the goodness of a `let` every time Optimistic Evaluation avoids building a thunk for that `let` (see Section 9.2.1). The thunk cost is expressed in bytes of heap allocation (see Section 9.1.1 for the reasons why).

Figure 12.12 shows the effect of varying the thunk cost. If the thunk cost is too low, then too little speculation takes place, and performance is reduced. However, if the thunk cost is too high, then too much speculation takes place, too much work is wasted, and performance again suffers. Note that Optimistic Evaluation outperforms Lazy Evaluation even with a thunk cost of zero. This is because a `let` that is always used will never accumulate any long-term badness and so will still be speculated.

Figure 12.13 gives the curves for the individual benchmark programs. One can see that, while some programs follow the trend in Figure 12.12 some are unaffected, or even speed up as the thunkcost is increased. These are programs that are almost entirely strict, for which the profiler is being overly conservative.

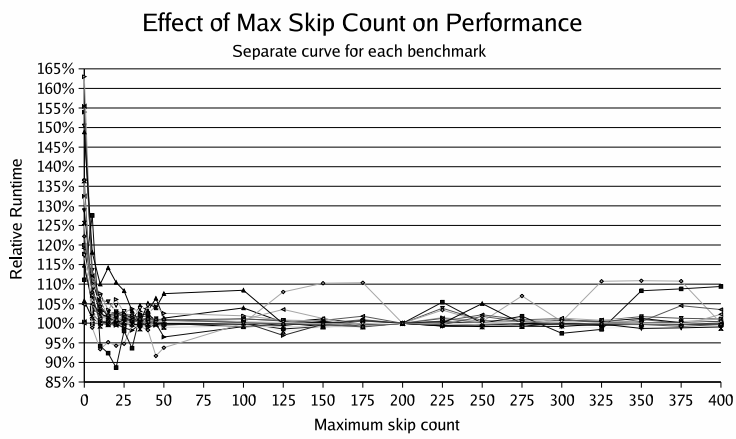


Figure 12.10: Performance Effect of Adjusting the “max sample period” Tunable—all benchmarks

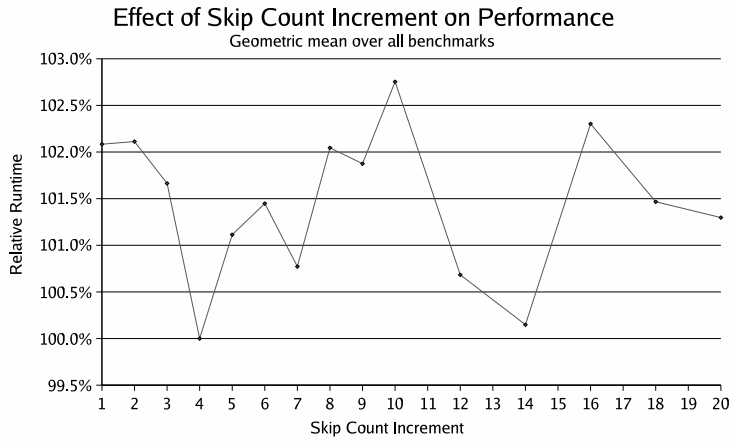


Figure 12.11: Performance Effect of Adjusting the “profi ling backoff” Tunable—geometric mean

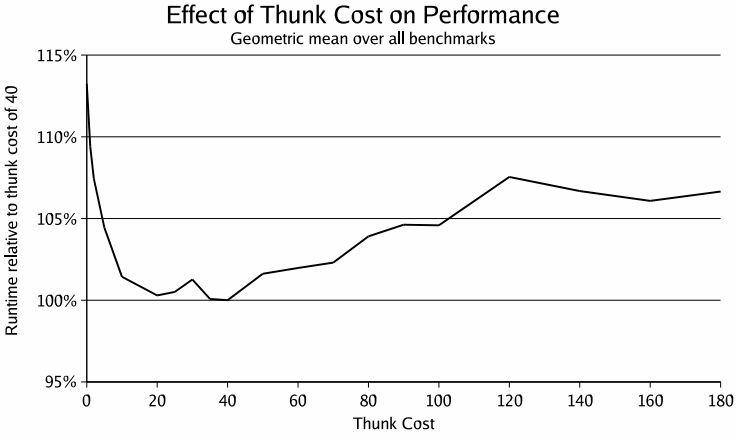


Figure 12.12: Performance Effect of Adjusting the “thunk cost” Tunable—geometric mean

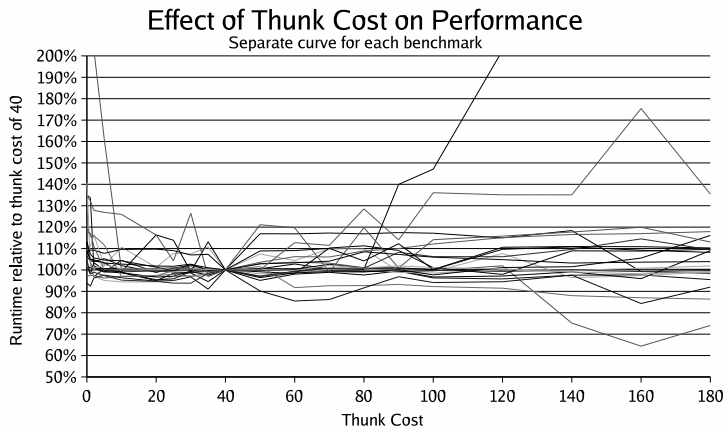


Figure 12.13: Performance Effect of Adjusting the “thunk cost” Tunable—all benchmarks

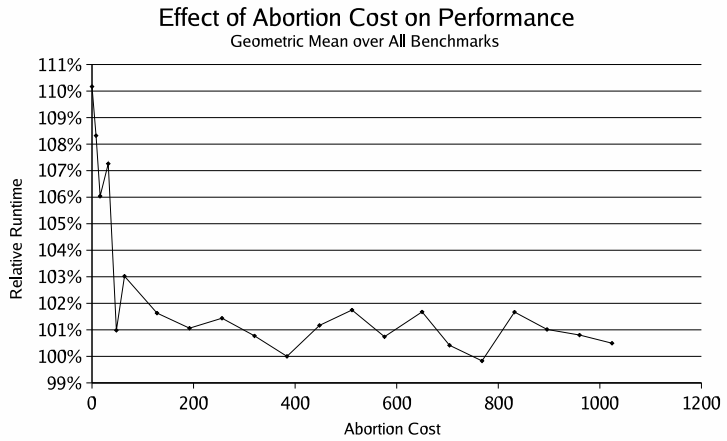


Figure 12.14: Performance Effect of Adjusting the “abortion cost” Tunable—geometric mean

12.3.4 What should the Abortion Cost Be?

The abortion cost, B_{abort}, is the cost incurred in aborting a venture, as estimated by the profiler. The profiler will add this amount to the wasted work of a **let** every time a venture for that **let** is aborted. As with the thunk cost, the abortion cost is expressed in bytes of heap allocation.

Figure 12.14 shows the effect of varying the abortion cost. If the abortion cost is very low then too much abortion takes place and programs slow down. Once the abortion cost is greater than around 100 there is relatively little effect. Figure 12.15 shows us that, while most programs are largely insensitive to the abortion cost, provided that it is above 100, a few programs can vary their performance by up to 20%. We are not sure why some programs vary so much. A limited investigation suggests that such programs often have a critical point where punishment for an abortion may cause an important **let** to stop being speculated.

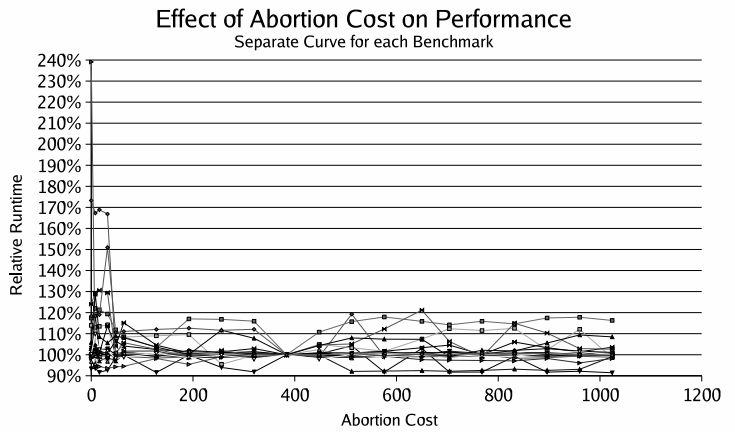


Figure 12.15: Performance Effect of Adjusting the “abortion cost” Tunable—all benchmarks

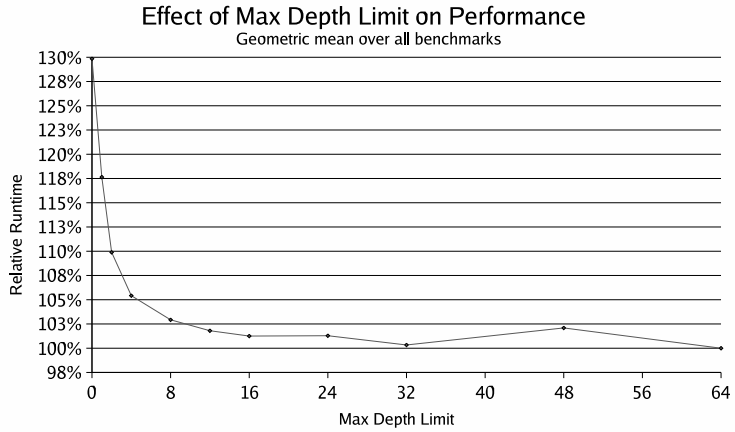


Figure 12.16: Performance Effect of Adjusting the “max depth limit” Tunable—geometric mean

12.3.5 What should the Maximum Allowed Speculation Depth Be?

The “max depth limit” is the maximum depth to which any `let` is allowed to speculate. This value is one of the parameters used to determine how goodness is mapped to a depth limit (Section 9.1.2).

Figure 12.16 shows the effect that the maximum depth limit has on performance. If the maximum depth limit is too low then speculation is overly constrained and performance suffers. The exact choice of maximum depth limit does not seem to matter much so long as it is greater than 8.

12.3.6 What should the Initial Speculation Depth Be?

The “initial depth limit” is the depth limit that every `let` has when a program starts, before any profiling has taken place. Figure 12.17 shows the effect that this has on performance. If the initial depth limit is zero, then no speculation can ever take place, and performance suffers. Other choices of initial depth limit make relatively little difference as the depth limit will be

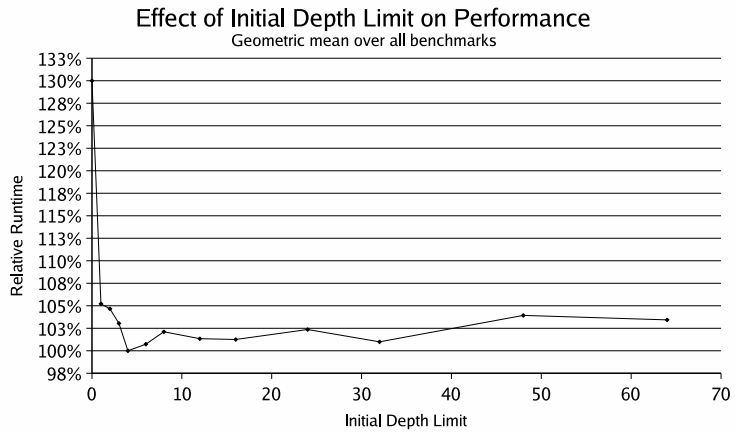


Figure 12.17: Performance Effect of Adjusting the “initial depth limit” Tunable—geometric mean

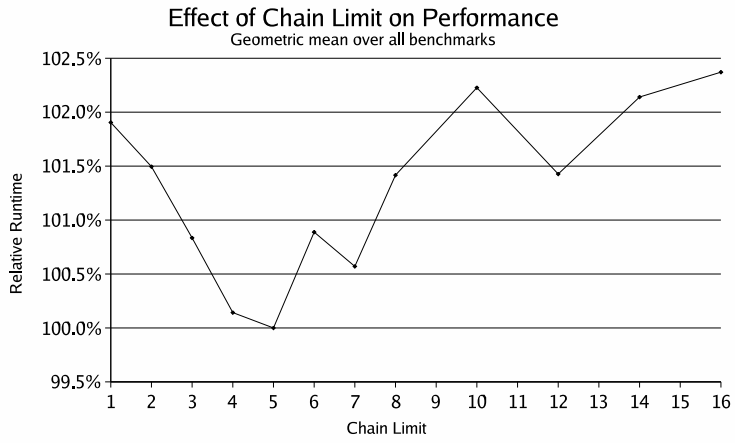


Figure 12.18: Performance Effect of Adjusting the “chain length” Tunable—geometric mean

quickly corrected by the profiler.

12.3.7 What should the Chain Limit Be?

In Section 9.4.1 we presented the idea of *profile chaining*, in which an unprofiled venture can be turned into a profiled venture if it demands a costed indirection. In that section, we also discussed the idea of a *chain limit* which is a bound on the number of chaining operations that any unit of blame can pass through. Figure 12.18 and 12.19 show the effect of varying the chain limit. While a limit of 5 gives the best performance, it is hard to tell whether this is just an artifact of our selection of benchmarks; indeed there is no consistent pattern followed by all benchmarks.

12.3.8 How Quickly should We Abort?

Figures 12.20 and 12.21 illustrate the effect of varying the $\mathrm{MAXBLAME}$ value, as described in Section 10.1. If abortion takes place too quickly then performance suffers, but the exact value

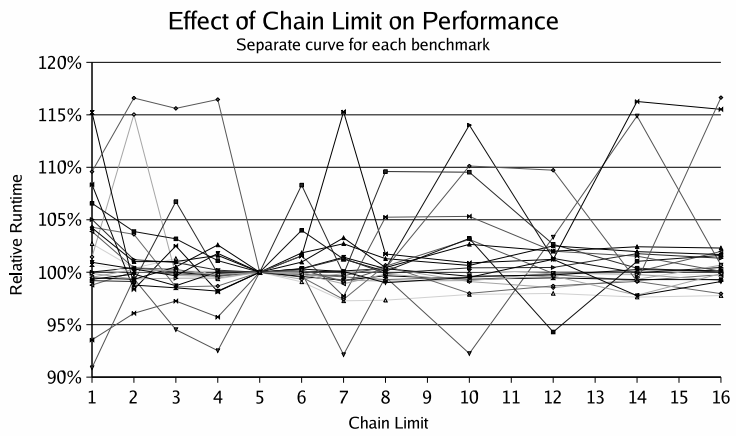


Figure 12.19: Performance Effect of Adjusting the “chain length” tunable—all benchmarks

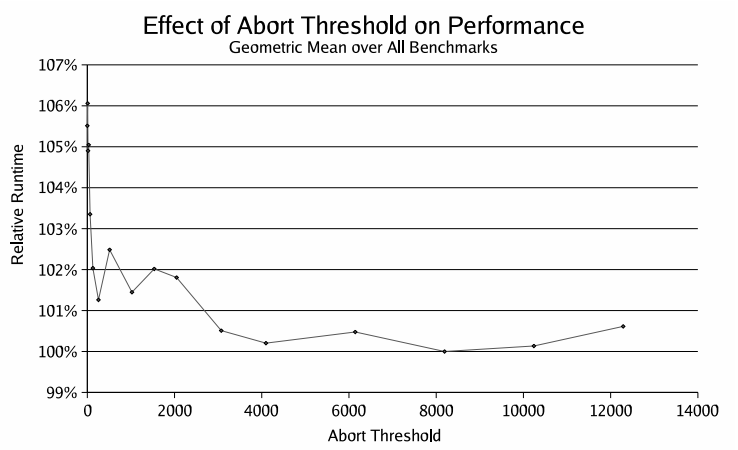


Figure 12.20: Performance Effect of Adjusting the $MAXBLAME$ Tunable—geometric mean

makes little difference, provided it is greater than around 4000. As with most other variables, the majority of benchmarks show little variation, however a few are quite sensitive to the choice of MAXBLAME value.

12.4 Metrics

In this section we present various statistics that provide an insight into the way that Optimistic Evaluation works.

12.4.1 Goodness as an Estimate of Performance

The online profiler estimates the performance of Optimistic Evaluation using a metric called *goodness* (see Chapter 6); but is this estimate accurate? Figure 12.22 compares the real speedup of each of our benchmarks with the total goodness, as measured by the profiler. The speedup is calculated relative to the fully lazy evaluator of Figure 12.3.

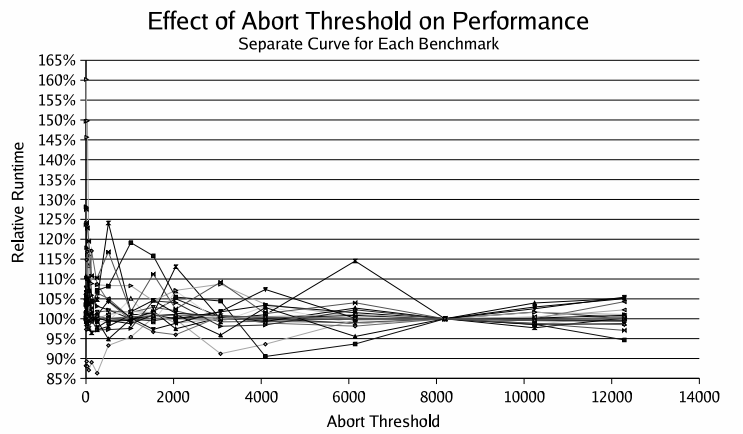


Figure 12.21: Performance Effect of Adjusting the $MAXBLAME$ Tunable—geometric mean

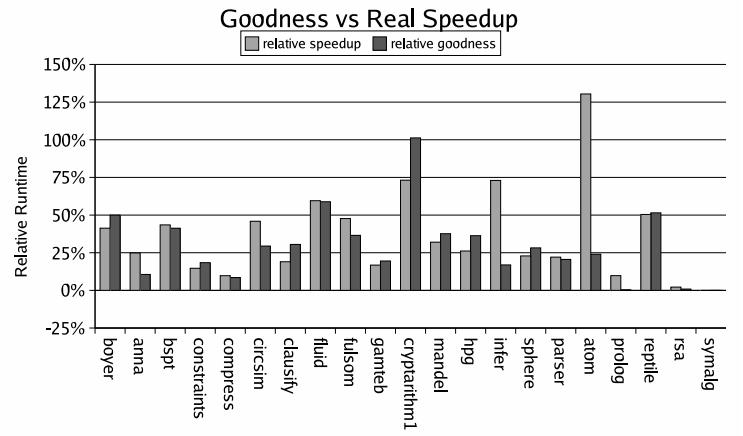


Figure 12.22: Goodness compared to Real Speedup

We can see from Figure 12.22 that, while the goodness is far from being a perfect measure of performance, it is usually pretty reasonable. The only programs whose performance estimate is off by a large amount are `infer`, `atom` and `prolog`. It is reassuring that in all these cases, goodness underestimated, rather than overestimated performance. These results suggest that our current profiler is estimating performance reasonably well, but that there is still room for improvement.

12.4.2 Heap Allocation as an Estimate of Work

The profiler uses heap allocation as an estimate of work done (Section 9.1.1), but is it a good estimate? Figure 12.23 gives the allocation rates for each of our benchmarks, expressed in bytes allocated per second of runtime. If heap allocation is a good estimate of runtime then we would expect the allocation rate to be the same for all benchmarks. In practice it varies by around a factor of five. Given that each of these allocation rates is the average allocation rate for an entire program run, it is likely that the local allocation rate varies significantly more. This

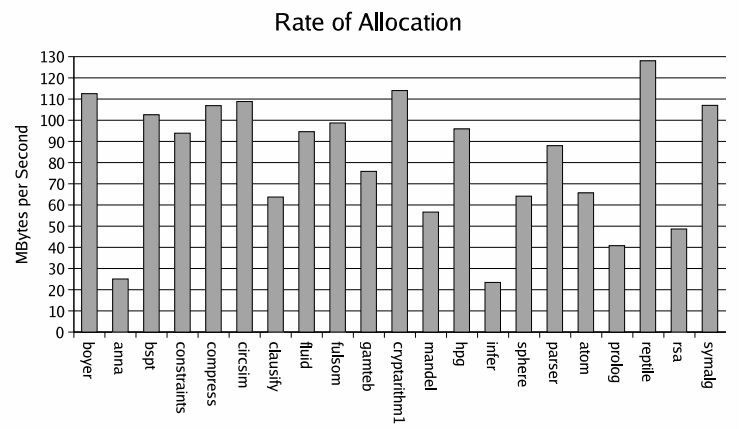


Figure 12.23: Rate of allocation

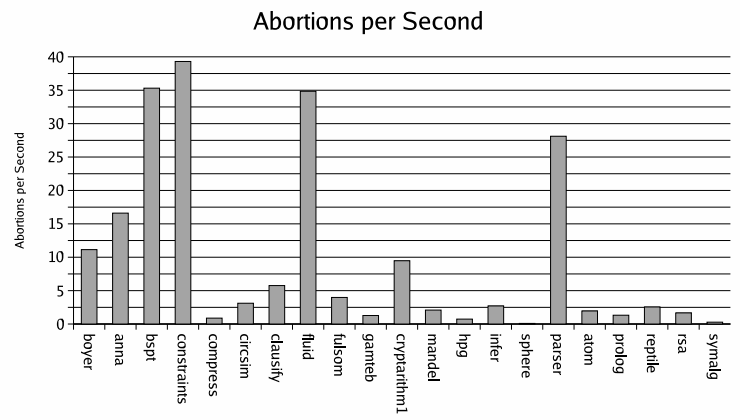


Figure 12.24: Abortions per second

suggests that, although measuring blame with heap allocation seems to work well in practice, cycle counting (Section 12.7.2) is probably the more reliable approach.

12.4.3 How Common is Abortion?

Figure 12.24 gives figures for the rate of abortion for each of our benchmarks. The benchmark that aborts most frequently is `constraints`, which aborts around 40 times a second. Most programs abort significantly more rarely, with a mean of around 10 abortions per second.

12.4.4 How Much Speculation Takes Place?

Figure 12.25 shows the proportion of `let` evaluations that are speculated for each of our benchmarks. We can see that no benchmark speculates less than 30% of its `let` evaluations, and that, on average, a program will speculate 62% of its `let` evaluations.

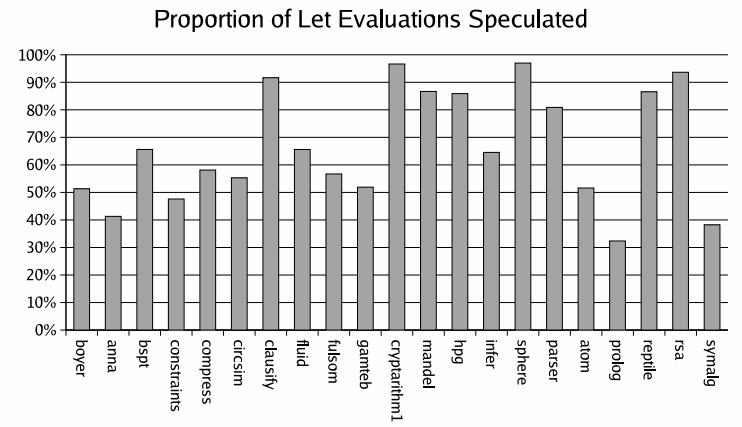


Figure 12.25: Percentage of `let` evaluations that are speculated

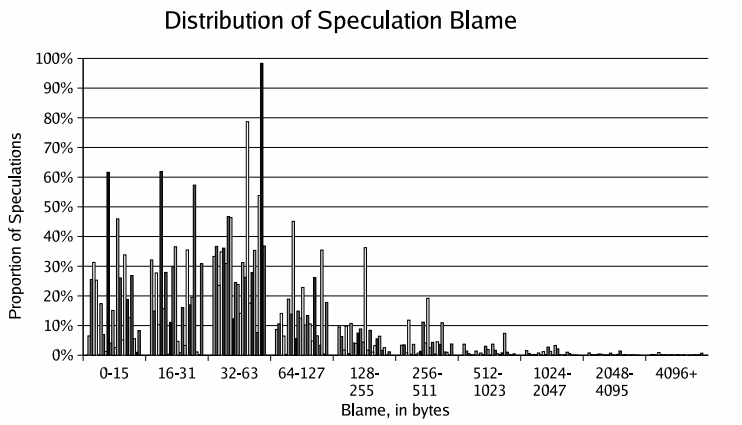


Figure 12.26: Distribution of Blame

12.4.5 What is the Distribution of Speculation Sizes?

Figures 12.26 and 12.27 show the distribution of blame and localwork (Section 3.3.3). In these graphs, each group of columns represents a range of possible sizes, and each column represents the proportion of speculations that had blame or local work within that range, for a particular benchmark. We can see from these graphs that very few speculations allocate more than 64 bytes locally, or accumulate more than 128 bytes of blame. We can also see that a significant proportion of speculations allocate more than 32 bytes locally.

12.4.6 What Proportion of Speculations are Used?

Figure 12.28 shows the proportion of speculations that turn out to be needed, grouped by the amount of work that was blamed on them. We can see that the vast majority of speculations are used, but that the profiler allows very cheap speculations to be used more rarely than others.

One might worry that the last benchmark seems to only need a small proportion of its speculations. This benchmark is `symalg`, which only performs a total of 425 speculations. This is

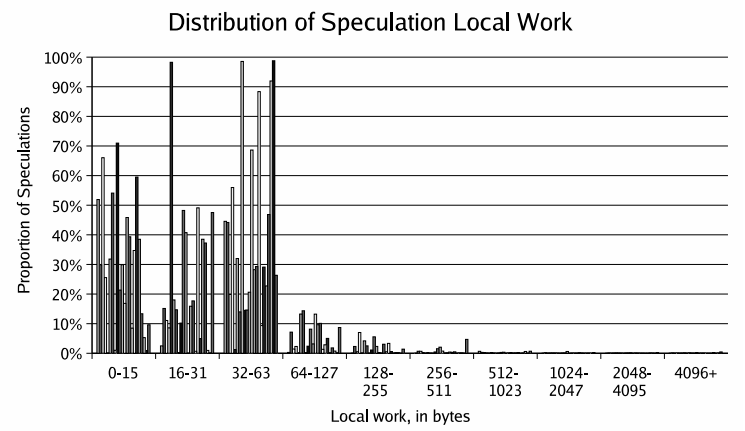


Figure 12.27: Distribution of Local Work

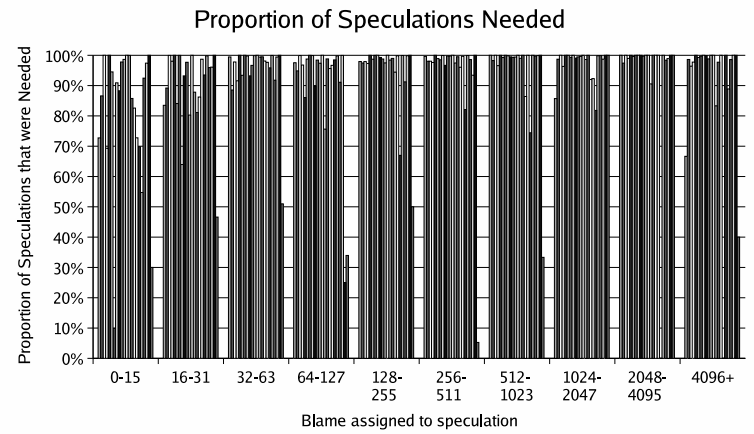


Figure 12.28: Proportion of Speculations that were Needed

not sufficient to allow the profiler to warm up, or to waste more than a tiny amount of work.

12.4.7 How much Chunky Evaluation Takes Place?

Figure 12.29 shows the distribution of speculation depths at which speculations took place. The horizontal axis is the speculation depth (see Section 8.1.2) while the vertical axis is the proportion of speculations that took place at that depth. Each curve represents the behaviour of a particular benchmark. We can see from this graph that the vast majority of speculations take place at depths less than 10, but also that a significant proportion of speculations take place at depths greater than 3.

12.4.8 How Long do Profile Chains Get?

Figure 12.30 shows the distribution of chain lengths (see Section 9.4.1) for the different benchmarks. These tests were done with a chain limit of 4. As we can see, many chains never last for longer than one link, but once a chain has grown to length 2, it has a very good chance of

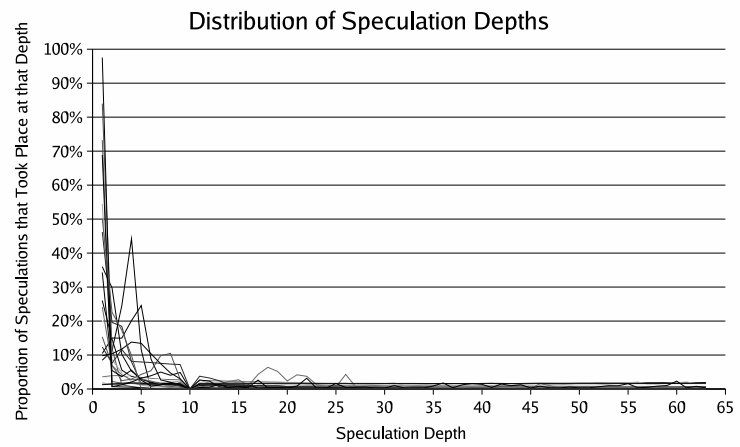


Figure 12.29: Distribution of Speculation Depths at which Speculations Take Place

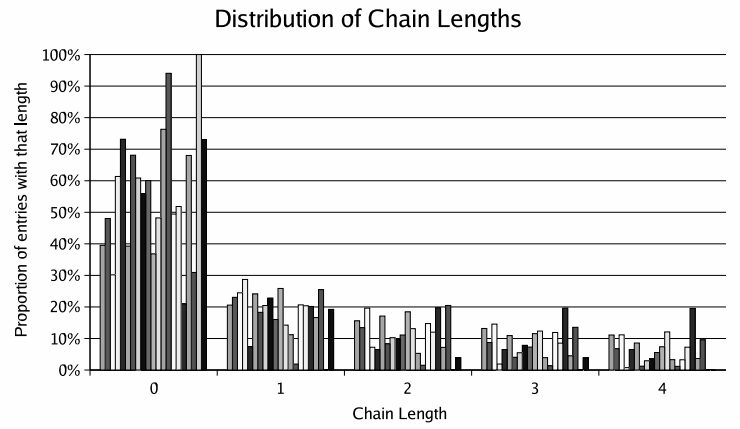


Figure 12.30: Distribution of Chain Lengths

lasting to length 4 (or beyond if the chain limit is greater).

12.5 Semi-Tagging

12.5.1 What is the Performance Effect of Semi-Tagging?

Figure 12.31 shows the performance effect of semi-tagging on both Optimistic Evaluation and Normal GHC. We can see that, while semi-tagging only improves the performance of Lazy Evaluation by an average of 2%, it improves the performance of Optimistic Evaluation by an average of 7%. Indeed, semi-tagging has a greater performance benefit for Optimistic Evaluation on every benchmark except `anna`, `fulsom`, and `symalg`. A significant proportion of the speedup from Optimistic Evaluation is due to the fact that it allows semi-tagging to be more effective.

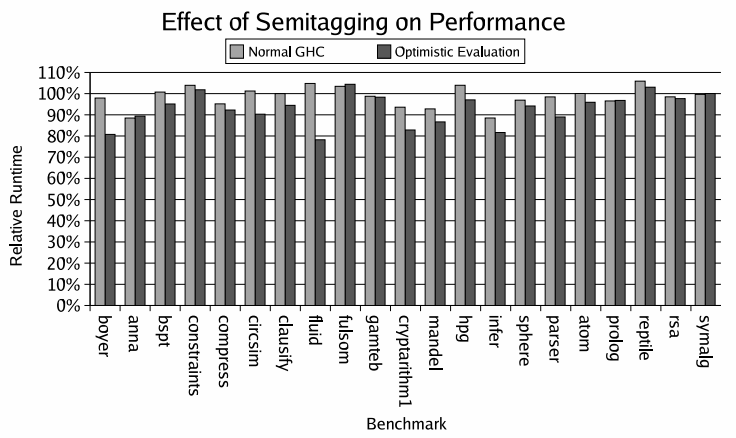


Figure 12.31: Performance Effect of Semi-Tagging for both Optimistic Evaluation and Normal GHC

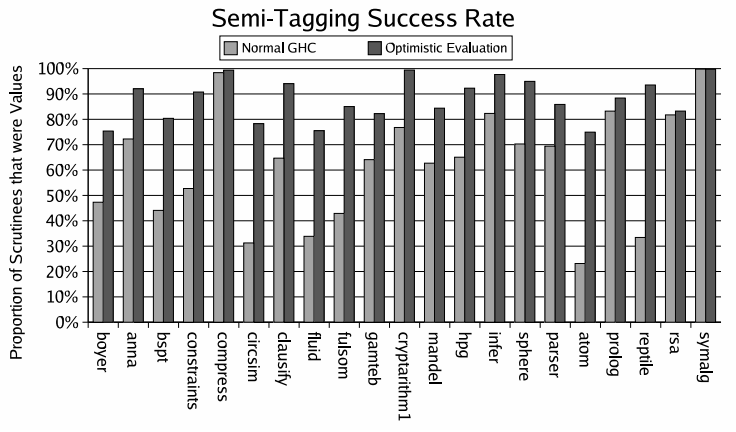


Figure 12.32: Percentage of Semi-Tagging Attempts that Find a Value

12.5.2 How Often does Semi-Tagging Succeed?

Figure 12.32 shows the success rate of semi-tagging for both Lazy Evaluation and Optimistic Evaluation. As this graph shows, Optimistic Evaluation dramatically increases the proportion of **case** scrutinees that turn out to be values. This explains the performance results in Figure 12.31.

12.6 Heap Usage

In this Section, we explore the effect of Optimistic Evaluation on heap usage.

12.6.1 Space Leaks

Some programs are extremely inefficient when executed lazily because they contain a space leak. People often post such programs on the `haskell` mailing list, asking why they are performing badly. One recent example was a simple word counting program [Mau02]. The inner loop (slightly simplified) was the following:

```
count :: [Char] -> Int -> Int -> Int -> (Int,Int)
count []       _ nw nc = (nw, nc)
count (c:cs) new nw nc =
  case charKind c of
    Normal  -> count cs 0 (nw+new) (nc+1)
    White   -> count cs 1 nw       (nc+1)
```

Every time this loop sees a character, it increments its accumulating parameter `nc`. Under Lazy Evaluation, a long chain of addition thunks builds up, with length proportional to the size of the input file. By contrast, the optimistic version evaluates the addition speculatively, so the program runs in constant space. Optimistic Evaluation speeds this program up so much that we were unable to produce an input file that was both small enough to allow the lazy implementation to terminate in reasonable time, and large enough to allow the optimistic implementation to run long enough to be accurately timed!

12.6.2 Heap Residency

Figure 12.33 illustrates the effect that Optimistic Evaluation has on heap residency: the maximum amount of data live in the heap during the program run. As this graph illustrates, while Optimistic Evaluation often reduces heap residency, the effect is not particularly pronounced. This suggests that, for the programs we have benchmarked, the performance improvements achieved by Optimistic Evaluation are not due to removal of space leaks.

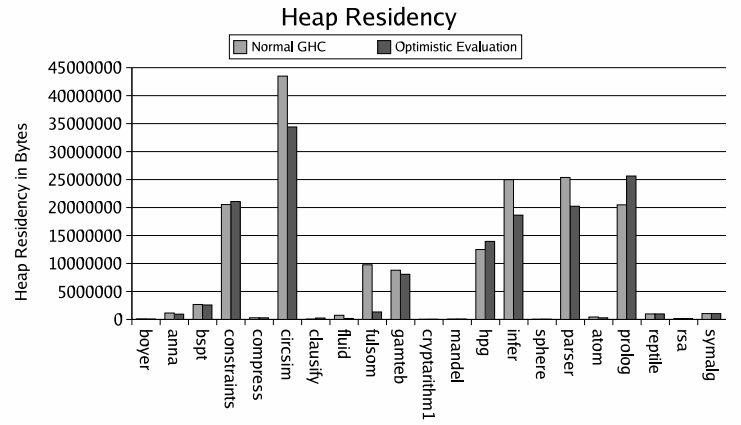


Figure 12.33: Effect of Optimistic Evaluation on Heap Residency

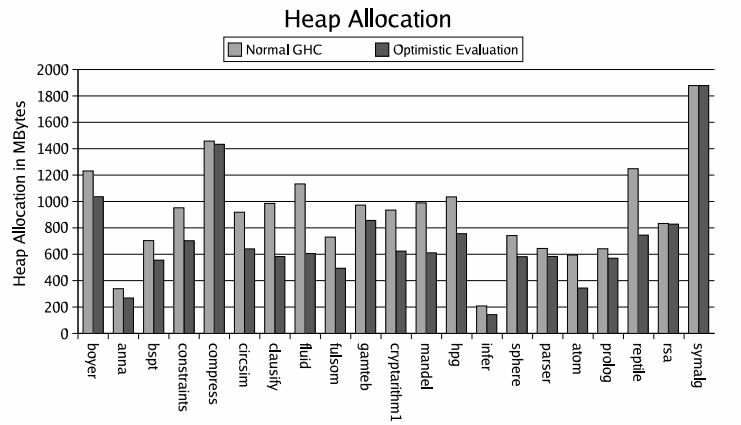


Figure 12.34: Effect of Optimistic Evaluation on Heap Allocation

This is what one would expect: the programs in the NoFib suite have been written by expert programmers, who can be assumed to have taken care to avoid space leaks when writing their programs. It thus seems likely that the programs that we have benchmarked do not contain any space leaks that Optimistic Evaluation could remove. This does not however mean that removal of space leaks in unimportant; while expert programmers may be able to avoid space leaks, they are a common problem for novice programmers. It might be interesting to see the effect of Optimistic Evaluation on a corpus of programs written by inexperienced programmers and see if the improvements are more pronounced.

Unlike most of our other graphs, Figure 12.33 shows absolute heap residency rather than relative residency. This is to avoid programs like `rsa` and `sphere` distorting the overall picture, despite the fact that their heap residency is negligible.

12.6.3 Heap Allocation

Figure 12.34 illustrates the effect that Optimistic Evaluation has on heap allocation: the total amount of storage allocated during the program run, including memory that was reclaimed

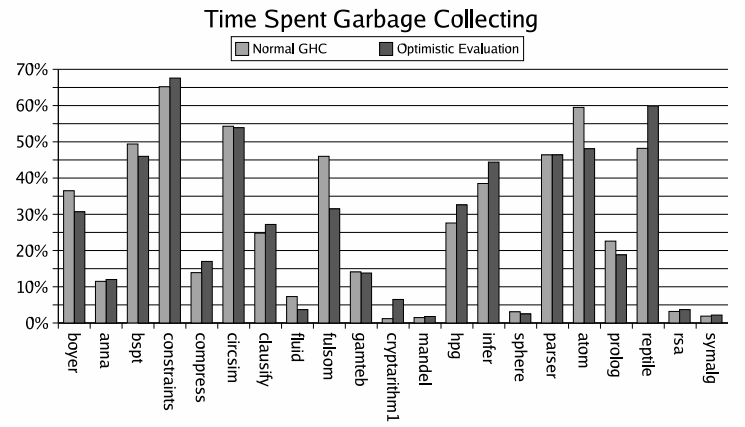


Figure 12.35: Proportion of Time Spent Garbage Collecting

by the garbage collector. As the graph illustrates, Optimistic Evaluation reduces the memory allocation of every benchmark program we tested. This is because it avoids allocating many of the thunks that Lazy Evaluation would create in the heap.

12.6.4 Proportion of Time spent Garbage Collecting

Figure 12.35 shows the proportion of time spent in the garbage collector. As this graph illustrates, many benchmarks spend a large proportion of their runtime in the garbage collector; thus the amount of time spent garbage collecting is very important. We can also see from this graph that Optimistic Evaluation has relatively little effect on the amount of time spent garbage collecting; indeed it increases it by around 1% on average. This backs up the conclusions drawn form Figure 12.33: the performance benefits of Optimistic Evaluation on this benchmark set do not come from avoiding space leaks.

12.6.5 Heap Profiling

We implemented two heap profiling schemes, which we described in Section 9.3. Figures 12.36 and 12.37 illustrate the effect of a profiler extension that punishes a **let** every time a costed indirection for that **let** is garbage collected. The horizontal axis is the factor by which the blame of a costed indirection is multiplied before being subtracted from the goodness of its **let**. We can see that this extension does not gain much on average (only around 1.5%), but that it can improve the performance of one program (`clausify`) by 20%.

Figure 12.38 illustrates the performance of the second profiling extension described in Section 9.3. This scheme adds a "lazy gc cost" to the goodness of a **let** every time the garbage collector collects a profiled thunk created for that **let**. This extension was found to always cause a performance reduction, with the reduction increasing as the "lazy gc cost" increased.

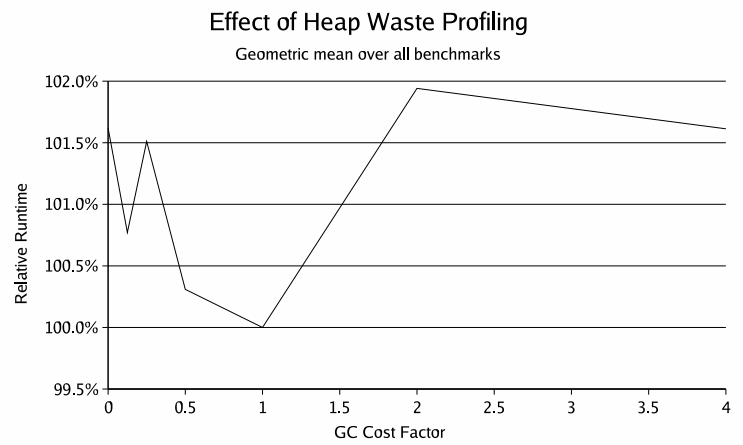


Figure 12.36: Performance Effect of Adjusting the “garbage collection cost” Tunable—geometric mean

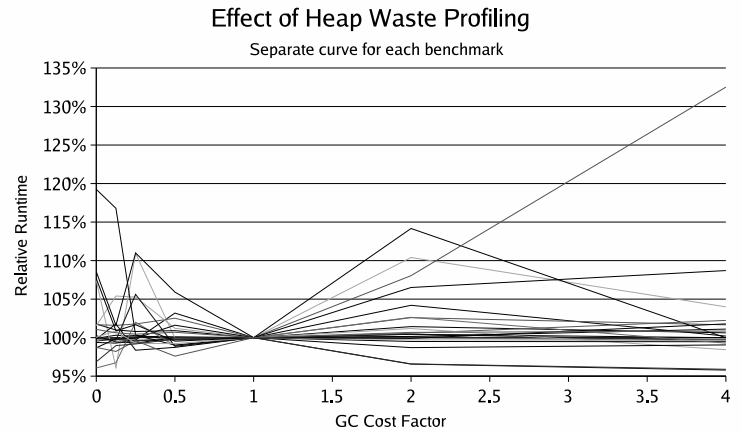


Figure 12.37: Performance Effect of Adjusting the “garbage collection cost” Tunable—all benchmarks

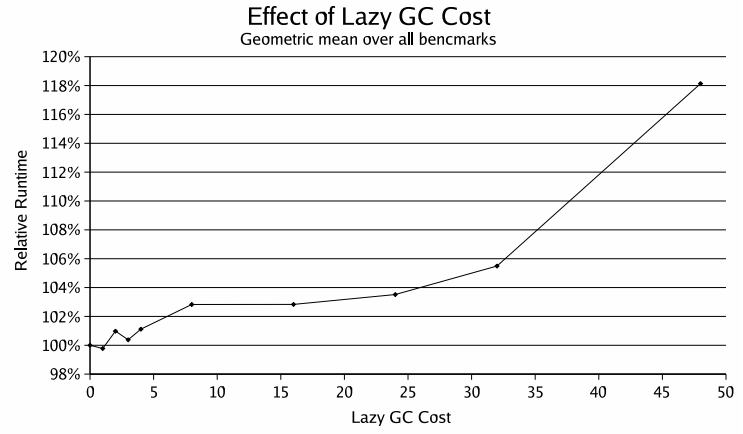


Figure 12.38: Performance Effect of Adjusting the “lazy gc cost” Tunable—geometric mean

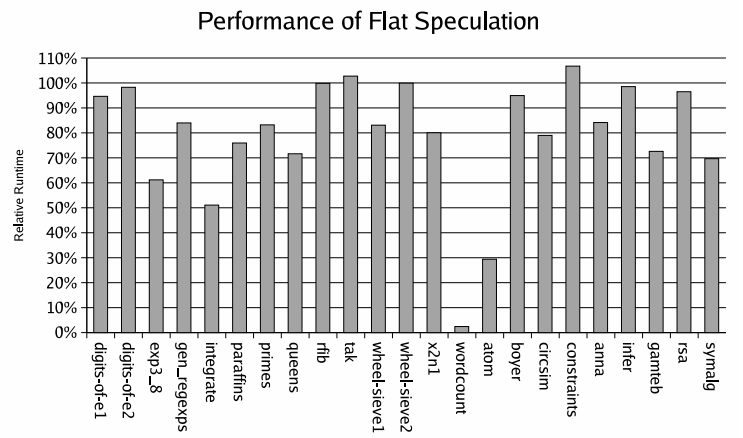


Figure 12.39: Performance of Flat Speculation

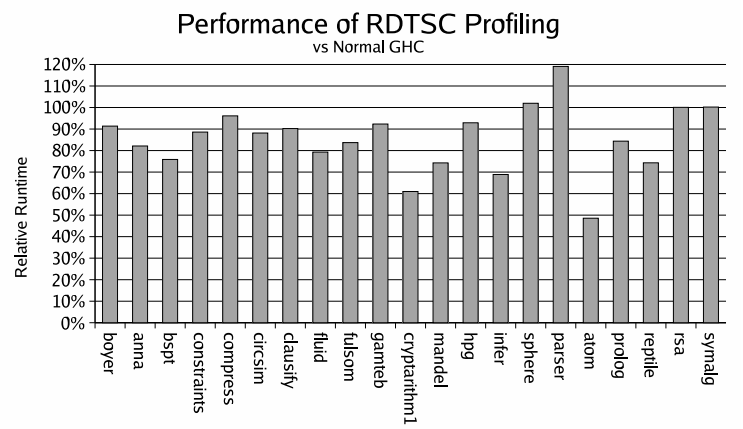


Figure 12.40: Performance of RDTSC Profi ling

12.7 Miscellanea

12.7.1 How Good is Flat Speculation?

In Section 8.2 we described an implementation technique called *Flat Speculation*. It is not fair to compare our implementation of Flat Speculation directly with our main implementation because it was forked from an earlier version of GHC, it relied on persistent speculation configurations (Section 9.4.3), it used a different profiler, it used a different semi-tagging mechanism, and it was not stable enough to run the larger benchmarks. One can however obtain a rough idea of its performance from Figure 12.39, which compares the performance of Flat Speculation to the performance of the GHC compiler that it forked from. This graph shows an average speedup of around 20% (excluding `wordcount`)—suggesting that Flat Speculation is roughly comparable to our primary implementation technique.

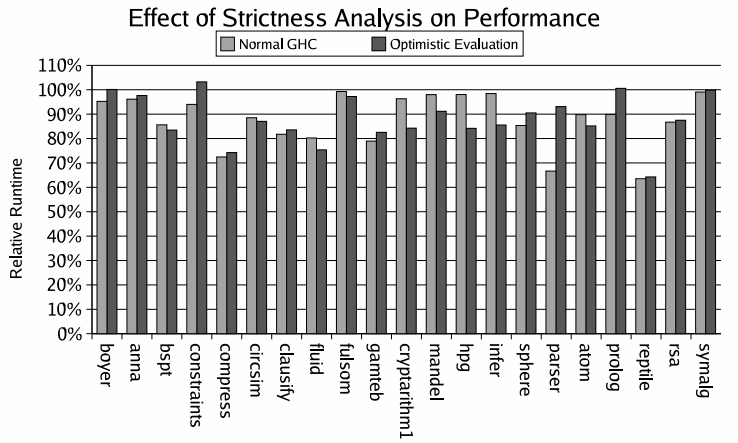


Figure 12.41: Effect of Strictness Analysis on Performance

12.7.2 How Good is RDTSC Profiling?

Figure 12.40 shows the performance of `rdtsc` profiling, as described in Section 9.4.4. The average performance improvement was 16%, compared to 20% for the heap allocation approach; however our implementation of `rdtsc` profiling is less mature than our implemenation of heap profiling, and we believe that similar performance should be attainable with further work.

12.7.3 What Effect does Strictness Analysis Have?

Figure 12.41 shows the effect of strictness analysis on both Lazy Evaluation and Optimistic Evaluation. Somewhat surprisingly, strictness analysis seems to bring about roughly the same performance gains under Optimistic Evaluation as it does under Lazy Evaluation. We are confused by this result: we would expect strictness analysis to have less effect under Optimistic Evaluation as the two techniques would be trying to avoid the same thunks. We find it particularly interesting that there are some benchmarks for which strictness analysis has a greater effect on Optimistic Evaluation than on Lazy Evaluation. We believe that this unexpected result may be due to the effect that strictness analysis has on the GHC optimiser: often quite complicated transformations may be made to a program, based on strictness information.

12.7.4 How Expensive are Transient Tail Frames?

In Section 11.3 we discussed transient tail frames, and explained how they make debugging easier. Figure 12.42 shows the performance overhead that transient tail frames cause. The average slowdown is 69%. This overhead can be significantly reduced if only part of a program is compiled with transient tail frames.

One program runs over 10 times slower.[3] This program contains a simple function that calls itself recursively with a large number of arguments. When this function executes normally, it

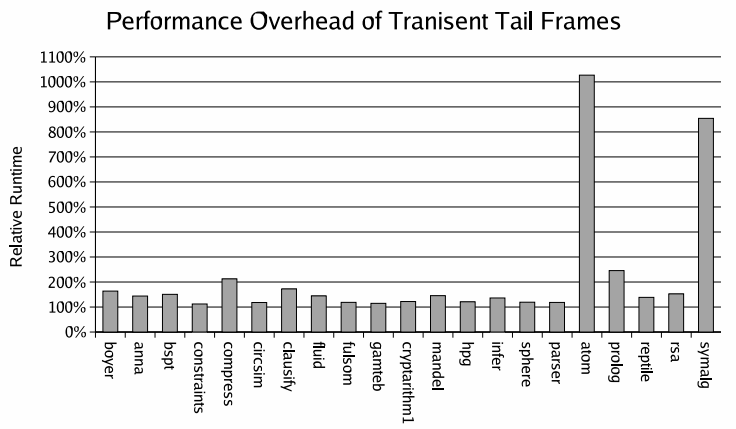


Figure 12.42: Performance Overhead of Transient Tail Frames

does not need to push anything on the stack at all; however when it executes with tail frames enabled, it has to push a large frame on the stack for every recursive call. We believe that it should be possible to extend our implementation of tail frames to avoid being caught out by functions such as this. One approach would be to detect such functions statically and only push tail frames for them if explicitly requested by the programmer. Alternatively, we could use a static analysis to discover which arguments could have differed from the previous recursive call, and only include these in the tail frame. Another possible fix would be to push tail frames onto a special circular stack, and thus only remember the last N calls, thus avoiding the overhead of removing them during garbage collection.

12.7.5 Is Lazy Blackholing Worth the Extra Complexity?

In Section 8.4 we discussed Lazy Blackholing and explained why it causes problems for Optimistic Evaluation. Figure 12.43 shows the performance effect of turning off Lazy Blackholing for Optimistic Evaluation: it actually speeds programs up by an average of 1%. These results suggest that lazy blackholing is not worth the considerable extra complexity it entails.

The results with lazy blackholing enabled were obtained using an implementation that included sufficient fixes to allow all the benchmark programs to run; however it is possible that bugs such as those described in Section 8.4 may account for some of the observed performance difference.

[3]We previously [EP03a] reported that the worst performance we had managed to provoke for transient tail-frames was a factor of 3 slowdown. This was at an early stage in the development of HsDebug, when we had not tested many programs.

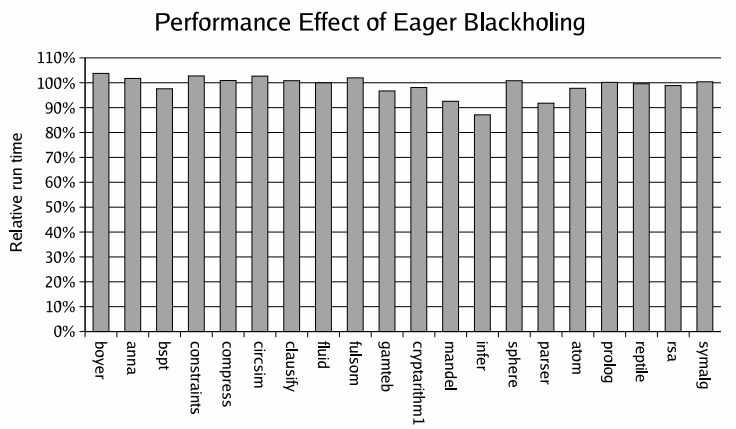


Figure 12.43: Performance of Eager Blackholing relative to Lazy Blackholing

CHAPTER 13

Related Work

In this chapter, we describe previous work that has similarities to the work presented in this thesis.

- In Section 13.1 we discuss *Static Hybrid Evaluation Strategies*. These are evaluation strategies that, like Optimistic Evaluation, attempt to combine lazy and eager evaluation; however, unlike Optimistic Evaluation, the choice of how to evaluate each **let** is made at compile time, rather than at runtime.

- In Section 13.2 we discuss *Eager Haskell*. Eager Haskell is an evaluation strategy that, like Optimistic Evaluation, uses speculative evaluation to improve the performance of non-strict programs. The main difference from our work is that, while we use an online profiler to decide which **let** expressions to speculate, Eager Haskell speculates all **let** expressions.

- In Section 13.3 we discuss *Speculative Evaluation for Multiprocessor Parallelism*. Various language implementations have used speculative evaluation to increase the number of concurrent tasks in a program and so find work for idle processors to do. While Optimistic Evaluation tries to speculate small expressions, parallel speculation strategies try to speculate large expressions. This leads to very different approaches.

- In Section 13.4 we discuss *Speculative Evaluation for Uniprocessor Parallelism*. If a processor has a large number of instruction units then a compiler may perform speculative operations on instruction units that would otherwise be idle. While these ideas could be applied to non-strict languages, we are not aware of any work that does this.

- In Section 13.5 we discuss previous work that is similar to the cost model that we presented in Chapter 5. While many previous cost models have been presented, we are not aware of any other model that is high level, composable, and able to model any hybrid of lazy and eager evaluation.

- In Section 13.6 we discuss previous work that has used profiling. While much such work exists, we are not aware of any previous work that has used profiling to direct the evaluation of non-strict programs.

- In Section 13.7 we discuss previous debuggers for non-strict languages and compare them to the debugger described in Chapter 11. While many such debuggers have been written, no previous work implements the stop-examine-continue model that our debugger uses.

- In Section 13.8 we discuss previous work that has attempted to reduce the amount of space that non-strict programs require.

- Finally, in Section 13.9 we discuss evaluation strategies that are not hybrids of lazy and eager evaluation.

13.1 Static Hybrid Strategies

Much previous work has attempted to combine Lazy Evaluation with Eager Evaluation, however most of this work has focused on static analyses or static annotations, rather than a dynamic, adaptive strategies.

In the subsections that follow, we discuss Strictness Annotations (Section 13.1.1), Strictness Analysis (Section 13.1.2) and Cheapness Analysis (Section 13.1.3).

13.1.1 Strictness Annotations

Strictness annotations are perhaps the simplest way of combining lazy and eager evaluation. Rather than letting the language implementation decide which expressions should be evaluated eagerly, the programmer gives this information explicitly in their program.

The 'seq' function

Haskell includes a seq 'function' that can be used to force the immediate evaluation of an expression.

$$seq : a \rightarrow b \rightarrow b$$

`seq` forces evaluation of its first argument and then returns its second argument. `seq` is the cause of much controversy in the Haskell community [Voi02, Mar02] due to the fact that it breaks several laws that would otherwise hold for Haskell programs.

Most other non-strict languages have similar constructs, for example Clean [BvEvLP87, NSvEP91] has a special `let!` construct that evaluates its right hand side eagerly. Both Haskell and Clean also allow data types to include strictness information, causing an implicit `seq` to be applied to any closure whose value is placed in a strict field.

Laziness Annotations

Many strict languages allow one to use annotations to introduce laziness. At the simplest level, one can implement Lazy Evaluation in a strict language by representing a thunk as an object with an `evaluate method`. At a slightly higher level, many strict languages provide syntactic sugar that makes it easy to embed Lazy Evaluation; for example the `lazy` keyword in O'Caml [LRVD98] or `delay` and `force` in Scheme [KCR98].

In practice, a typical non-strict program will only make essential use of laziness in a small proportion of its expressions; it is thus fairly easy for a programmer to mark these expressions as being lazy. Many people argue that laziness annotations are good style; if a program relies on Lazy Evaluation, this should be a deliberate policy decision and thus should be made explicit in the program text.

13.1.2 Strictness Analysis

Most compilers for non-strict languages make use of a static analysis called *Strictness Analysis* to determine which expressions should be evaluated eagerly. Strictness Analysis attempts to find expressions that are certain to be evaluated under Lazy Evaluation and which can thus be evaluated eagerly without risking poor performance or non-termination. There has been a lot of work done on Strictness Analysis [Myc81, BHA86, MN92, WH87] and the technology is now very mature.

Online Profiling vs Static Analysis

A strictness analyser has to be very careful because it risks a lot if it makes a mistake. If Strictness Analysis were to mistakenly decide to evaluate an expensive unneeded expression eagerly then the program could waste the total cost of all evaluations of that expression. Without abortion or adaption there is no way for the system to recover from a bad decision. Strictness Analysis is thus forced to be very conservative.

By contrast, Optimistic Evaluation risks a lot less if it makes a mistake. If Optimistic Evaluation mistakenly decides to evaluate an expensive unneeded expression eagerly, then the profiler and abortion mechanism ensure that only a relatively small amount of work can be wasted before the expression will revert to being evaluated lazily. This allows Optimistic Evaluation to be a lot more aggressive than Strictness Analysis.

Strictness Analysis can make Programs Slower

It should be noted that Strictness Analysis is not guaranteed to improve the performance of a program. Consider for example the following program:[1]

$$\begin{aligned} f\ 0 &= [] \\ f\ n &= n : f(n-1) \end{aligned}$$

$$\begin{aligned} g\ [] &= 0 \\ g\ (x:xs) &= 1 + g(xs) \end{aligned}$$

$$main\ =\ g\ (f\ 1000000000)$$

The recursive calls to f are needed, and thus a strictness analyser may make them eager. However, if f is evaluated eagerly then $f\ 1000000000$ will build up an enormous structure in the heap and so evaluation of $main$ will require a lot of space. By contrast, if f was evaluated lazily then the list producer f and the list consumer g would run in lock-step, causing $main$ to run in constant space. If heap space is increased then garbage collection time can increase (Section 9.3.1) and so Strictness Analysis is likely to make this program run more slowly.

As we explained in Section 9.3.2, we have implemented a crude mechanism which we believe should prevent this problem occurring with Optimistic Evaluation; however we believe that further work is required in this area.

[1] This particular program can be simplified by deforestation [Wad90a, Wad84, GLP93], however not all such programs can.

13.1.3 Cheapness Analysis

Cheapness Analysis [Myc80, Fax00] is another static analysis in the same vein as Strictness Analysis. Like Strictness Analysis, Cheapness Analysis examines the static text of a program and attempts to find expressions that can be safely evaluated eagerly. The key difference is that while Strictness Analysis looks for expressions that are guaranteed to be needed, Cheapness Analysis looks for expressions that are guaranteed to be cheap.

How it works

Is $x+1$ cheap to evaluate? It depends on whether x is already evaluated. If x is an argument to a function then we may need to examine all calls to the function—and that is not straightforward in a higher-order program. Faxén solves this problem using a sophisticated whole-program flow analysis. Unfortunately, being a whole-program analysis, it causes problems for separate compilation. These problems are probably soluble—for example by compiling multiple clones of each function, each suitable for a different evaluation pattern—but they further complicate the implementation. This is why we were not able to implement cheap eagerness in GHC for comparison purposes.

Dynamic Cheap Eagerness

A further development, Dynamic Cheap Eagerness [Fax01], uses a more complicated analysis to find cheap recursive functions and arrange for them to call themselves eagerly up to a certain depth. The effect is very similar to the speculation depth limits used by Optimistic Evaluation, however the limits are determined at compile time by a static analysis, rather than being determined at runtime by a profiler.

Performance

Faxén reports some promising speedups, generally in the range 0-25% relative to his baseline compiler, but these figures are not directly comparable to ours. As Faxén is careful to point out, (a) his baseline compiler is a prototype, (b) his strictness analyser is “very simple”, and (c) all his benchmarks are small. The improvements from Cheapness Analysis may turn out to be less persuasive if a more sophisticated strictness analyser and program optimiser were used, which is our baseline. (Strictness Analysis does not require a whole-program flow analysis, and readily adapts to separate compilation.)

Worst Case Performance

How cheap is cheap? Cheap Eagerness uses a fairly arbitrary threshold for deciding whether an expression is cheap. Importantly, this threshold is considerably greater than the cost of building a thunk. If Cheap Eagerness eagerly evaluates a lot of cheap expressions that are not needed then performance can be considerably worse than Lazy Evaluation. This problem is made worse by Dynamic Cheap Eagerness: although the body of a recursive function may be cheap, this cost must be multiplied by the number of times that the function is allowed to call itself recursively.

It is of course possible to improve the worst case performance by reducing the threshold below which an expression is considered to be cheap, and by reducing the depth to which a recursive function is allowed to be called. However such a change will reduce the amount of eager evaluation that can take place and will make the average performance worse.

13.1.4 Other Static Optimisations

There exist many other static optimisations that can improve the performance of non-strict programs. A particularly interesting example is the GRIN project [Boq99]. GRIN does a lot of clever optimisations, but perhaps its most interesting optimisation is its thunk inlining transformation. GRIN performs a whole-program flow analysis for a program, attempting to discover which thunks can be evaluated at any particular point. Consider for example the following program:

$$
\begin{array}{lll}
f\ x = & x + 1 & \\
 & & \\
g\ y = & \textbf{let}\ p = y + 1 & \textbf{in} \\
 & \textbf{let}\ q = y + 2 & \textbf{in} \\
 & (fp) + (fq) &
\end{array}
$$

GRIN's flow analysis will realise that, inside the body of f, x must be either p or q. It will thus translate this program into something rather like the following program:

$$
\begin{array}{lll}
f\ x & \textbf{case}\ x & \textbf{of} \\
 & \quad ThunkP\ y \rightarrow y + 1 + 1 & \\
 & \quad ThunkQ\ y \rightarrow y + 2 + 1 & \\
 & & \\
g\ y & \textbf{let}\ p = ThunkP\ y & \textbf{in} \\
 & \textbf{let}\ q = ThunkQ\ y & \textbf{in} \\
 & (fp) + (fq) &
\end{array}
$$

We can see that the code for the thunks $y + 1$ and $y + 2$ has been inlined at the point at which the thunks might be demanded. This optimisation, together with the further optimisations that

it enables, can significantly reduce the cost of Lazy Evaluation.

Another interesting static optimisation is Wansbrough's usage analysis [Wan02, WP99]. This uses an analysis similar to that used for linear types [Wad90b, Bak95, TWM95] to detect thunks that can only be used at most once. If a thunk can be used at most once then the implementation can avoid the cost of pushing and later entering an update frame (Section 7.4.2).

13.2 Eager Haskell

Eager Haskell [Mae02b, Mae02a] was developed simultaneously, but independently, from our work. Its basic premise is identical: use Eager Evaluation to improve the performance of non-strict programs, together with an abortion mechanism to back out when eagerness turns out to be over-optimistic. The critical difference between Eager Haskell and Optimistic Evaluation is that while Optimistic Evaluation uses an online profiler to decide which expressions should be evaluated eagerly, Eager Haskell evaluates everything eagerly.

13.2.1 Code Generation

The code generated by Eager Haskell is largely the same as would be generated for a conventional strict language such as ML [MTHM97]. `let` expressions never evaluate lazily, and so there is no need to generate any code for lazy thunks or to generate branches at the beginning of `let` expressions.

13.2.2 Abortion

What distinguishes Eager Haskell from a conventional strict language implementation is its abortion mechanism. Like Optimistic Evaluation, Eager Haskell uses abortion to back out of speculations that turn out to be too expensive; however the abortion mechanism in Eager Haskell is very different to that used by Optimistic Evaluation.

Optimistic Evaluation considers abortion to be a tool of last resort, using profiling as its primary mechanism for preventing wasteful speculation. As we showed in Section 12.4.3, abortion is a rare event, taking place only if a speculation has been running for a very long time. If speculations spawned by a `let` are frequently aborted then it is likely that the profiler will avoid speculating that `let` in the future.

By contrast, Eager Haskell uses abortion as its primary mechanism for coping with wasteful speculations. This leads to very different design choices. Abortion in Eager Haskell is frequent and periodic: once a given amount of heap has been allocated, all execution will be aborted. The Eager Haskell abortion mechanism does not care how long any active speculations have been running for, or even whether the program is speculating at all—abortion is purely periodic.

The mechanism used for implementing abortion itself is also very different from that of Optimistic Evaluation. While Optimistic Evaluation manages abortion in the runtime system, Eager Haskell instead puts the abortion code for an expression in the expression code itself. Once the abortion flag is set, any subsequent function call will return a suspension instead of evaluating its body. If a **case** expression finds that its scrutinee is a suspension then it will return a suspension itself. Similarly, a function application will return a suspension if the function being applied evaluates to a suspension. Abortion continues right back to the root of the stack. Once abortion has completed, evaluation is restarted by forcing the suspension for the root computation.

Eager Haskell has gone to considerable effort to make abortion very efficient. The abortion check at the entry to a function is cunningly combined with a stack check, making it essentially free. Similarly, various tricks are employed to allow large chunks of stack to be aborted and restored without having to be repeatedly copied between the heap and stack. Eager Haskell's abortion mechanism is thus considerably more efficient than that used by Optimistic Evaluation.

13.2.3 Chunky Evaluation

Eager Haskell's periodic abortion gives it a form of chunky evaluation, but with chunk sizes being limited by abortion frequency rather than by speculation depth. To see the effect that this has, consider a tree structure in which the first child of a node is always unneeded, but the second child is always needed. Optimistic Evaluation will quickly learn to evaluate the second child of a node chunkily but to evaluate the first child lazily. By contrast, Eager Haskell will always evaluate the tree in a depth first manner. It will follow the first child of a node and may be aborted before it has a chance to evaluate the second child.

13.2.4 Worst Case Performance

Eager Haskell is able to guarantee that, in the worst case, it will be only a constant factor slower than Lazy Evaluation; however the constant factor involved is quite large: Maessen reports that the `constraints` program from the NoFib suite slows down by a factor of over 100 [Mae02a]. Bad behaviour such as this occurs when there are many expressions that are expensive and unused; Eager Haskell will evaluate them all eagerly, and is thus likely to spend a considerable proportion of its time doing unnecessary work.

Increasing the abortion frequency will reduce the amount of wasted work, but it will also increase the overheads, as the proportion of runtime taken up by abortion increases. It is thus the case that, in order to achieve good performance on average, one must accept bad performance in the worst case.

The worst case performance of Eager Haskell relative to Lazy Evaluation is arguably unimportant. Rather than seeing Eager Haskell as a new way to evaluate existing Haskell programs,

it is perhaps better to see Eager Haskell as a new language which is largely compatible with Haskell, but which has different semantics. If a program is written with Eager Haskell in mind then it is not hard to ensure that Eager Haskell will evaluate it efficiently. In particular, Eager Haskell has support for laziness annotations that can be used by a programmer to mark expressions that should not be speculated. One might argue that annotating expressions that are potentially expensive and unnecessary is good programming style, regardless of the evaluation strategy being used.

13.2.5 Average Performance

The average performance of Eager Haskell is considerably better than its worst case performance but is, at least at present, inferior to that of Optimistic Evaluation. In Maessen's thesis [Mae02b] he claims an average slowdown of 60% relative to GHC, compared with an average speedup of around 20% for Optimistic Evaluation.

It is likely that Eager Haskell could be made significantly faster. Unlike us, Maessen wrote his compiler from scratch; it is thus considerably more primitive than ours and has many ways in which it could be made significantly faster. It seems reasonable to assume that, for suitably written programs, a more advanced implementation of Eager Haskell should be able to approach the speed of a high-performance strict functional languages such as O'Caml [LRVD98].

In some benchmarks Eager Haskell achieves performance that is better than Optimistic Evaluation; however one has to bear in mind that Optimistic Evaluation incurs considerable overheads in its switchable `let` expressions and online profiler; both of which are essential if the worst case behaviour is to be bounded.

13.3 Speculative Evaluation for Multiprocessor Parallelism

The parallel programming community has been making use of speculation to exploit multiple processors for a long time [Bur85]. There, the aim is to make use of spare processors by arranging for them to evaluate expressions that are not (yet) known to be needed. There is a large amount of work in this field, of which we can only cite a small subset.

13.3.1 Basic Principles

Optimistic Evaluation and Speculative Evaluation for Multiprocessor Parallelism look superficially very similar: both systems evaluate expressions without knowing that they are needed. However there are important differences.

Under Optimistic Evaluation, every cycle spent in a speculative evaluation is a cycle that could instead have been spent doing useful work. It is thus vitally important that the expression

being speculated is actually needed. By contrast, speculative evaluation for multiprocessor parallelism will typically perform speculative evaluations on processors that would otherwise be idle. While such systems would prefer it if the expressions being speculated were needed, the issue is far less pressing.

There is a similar dichotomy regarding the sizes of speculative evaluations. In Optimistic Evaluation, each speculative evaluation saves the fairly-constant amount of work that would be required to build a thunk, but potentially wastes the amount of work spent performing the speculative evaluation. Optimistic Evaluation thus aims to perform a large number of small speculations. By contrast, in a parallel setting, each speculative evaluation wastes the fairly-constant amount of work required to fork a new process, but potentially-saves the amount of work spent performing the speculative evaluation. Parallel systems thus aim to perform a small number of large speculations.

These two differences cause multiprocessor speculation systems to be designed in very different ways to Optimistic Evaluation.

13.3.2 Making Speculation Efficient

A lot of work in the parallel community focuses on making parallel speculation efficient; however most of this work is inapplicable to Optimistic Evaluation.

One interesting example is Local Speculation. Local Speculation [MJG93, Cha98] does some speculations on the local processor when it would otherwise be waiting. This avoids the overhead of transferring work to another processor and allows shorter speculations to be worthwhile.

Another strand of work attempts to aggregate tiny threads into larger, compound threads that can be spawned together and thus share their thread startup costs [Tra88, SCG95]. In some ways this is closer to our work: Lazy Evaluation is a bit like parallel evaluation scheduled on a uniprocessor, while Eager Evaluation instead aggregates the thread for the right hand side of a **let** into the thread for the body. However the issues addressed are very different; as Schauser puts it "the difficulty is not what *can* be put in the same thread, but what *should* be ... given communication and load-balancing constraints". Furthermore, such thread partitioning systems are static, whereas our approach is dynamic.

13.3.3 What To Speculate

Which expressions should be evaluated speculatively?

The simplest answer is to evaluate everything speculatively. This is the approach taken by leniently evaluated dataflow languages such as Id [Nik91] and pH [NA01]. A lenient language will evaluate $\mathbf{let}\ x = E\ \mathbf{in}\ E'$ by starting a new task for E and then evaluating E'. Eager

Haskell (See Section 13.2 has its roots in the world of lenient languages, and can be seen as being rather like a sequential implementation of pH.

Another approach is to provide an explicit language construct that lets the user specify which expressions should be speculated. MultiLisp [Osb89, Hal85] allows expressions of the form `(future E)` that specify that the expression E should be evaluated in parallel with the current task, in the hope that its value turns out to be useful [FF95].

Clearly some speculative tasks are more likely to be useful than others. There has thus been considerable work on assigning priorities to tasks. At the simplest level, one can give speculative tasks lower priorities than mandatory tasks, ensuring that a speculative task never prevents a mandatory task from running. Extensions of this include systems in which the priority of a task takes into account user assigned priorities [Bur85], or the depth of speculation [Mat93], or its reachability from other tasks [Par91, PD89].

We are not aware of any work in the parallel speculation community that uses online profiling to decide which expressions should be speculated; however it is possible that an online profiler might be useful for this purpose.

13.3.4 Abortion

Abortion is not essential in a parallel speculation system. If an unnecessary speculative task does not terminate then the worst that can happen is that it blocks execution of lower priority speculative tasks—it cannot cause the whole program to not terminate. Nevertheless, it is still desirable for a speculative task to be aborted if it is certain that its result will not be needed. One common approach is to use a garbage collector to find speculative tasks whose result closures are not reachable from any other task [BH77, HK82, GP81].

13.4 Speculation for Uniprocessor Parallelism

Modern processors typically have a large number of instruction units which can execute in parallel. If a compiler is not able to find mandatory work for all instruction units, then it may choose to perform speculative work on instruction units that would otherwise be idle. Unlike speculative evaluation for multiprocessors, such systems aim to find small speculative tasks, and, unlike Optimistic Evaluation, all work that we are aware of selects such tasks at compile time.

Although we believe that these techniques could work well for non-strict languages, we are not aware of any work that has done this.

13.4.1 Speculation in Imperative Languages

Much work on uniprocessor speculation focuses on automatically extracting parallelism from imperative programs. Rather than speculatively evaluating pure expressions, one instead speculatively performs imperative commands. The issues involved here are more complex than for purely functional languages, as illustrated by the following program:

```
do C
if (B){
   do C'
}
```

A compiler may be able to improve performance by executing the commands C and C' simultaneously. In order for this to be sound, it must be the case that C does not change any state read by C' (*data speculation*) and that C' would actually be executed by normal sequential evaluation (*control speculation*). If either of these requirements turn out to be false, then the speculative execution of C' must be *reverted*, undoing any imperative actions it may have performed.

13.4.2 Scheduling Instructions

A compiler can schedule speculative tasks statically by interleaving speculative instructions with mandatory ones. There has been particular interest in doing this for Intel's IA64 [ia600] processor, which has a large number of instruction units and no dynamic instruction reordering [LCH+03].

Simultaneous Multithreading architectures [TEL95, Gwe99, Eme99] allow multiple threads to run simultaneously on one processor, all of which have access to the same instruction units. Speculative Multithreading architectures [Kri99, MG99] extend this by providing hardware support for thread reversion and sequential dependencies between threads. Such architectures make it much easier for a compiler to exploit multiple instruction units because the interleaving of tasks is done dynamically by the processor, rather than statically by the compiler. Considerable work has been done on compiling programs efficiently for such architectures [BF02, PO03].

While offline profiling has been used to assist in the scheduling of uniprocessor parallelism [BF02], we are not aware of any work that uses online profiling; indeed it seems unlikely that online profiling would be worthwhile.

13.4.3 IO Speculation

Some programs spend a large proportion of their time waiting for pages to be loaded from disk. The performance of such programs can be improved by speculatively executing the instructions that should be performed once the page has loaded [CG99, FC03]. In this case, the hope is that the speculative task will request pages that the mandatory task will need when it awakes, thus allowing several pages to be fetched in parallel and reducing the effects of IO latency.

13.4.4 Branch Prediction

Perhaps the best known example of control speculation is *branch prediction* [Smi81]. Branch Prediction is a feature of most modern processors that speculatively evaluates the side of a branch that the processor believes is most likely to be used. Although Branch Prediction is normally carried out at runtime by the processor, it is also possible to do it using offline profiling [FF92] or statically in the compiler [BL93].

13.5 Models of Cost

There has been much previous work on cost models. In this Section we discuss several previous models of cost and compare them to the model that we presented in Chapter 5

13.5.1 PSL Computation Graphs

Perhaps the most similar cost model to ours is the *PSL Computation Graphs* of Greiner and Blelloch [GB99, GB96]. Computation graphs are used to analyse the cost of evaluating a program using a data-driven parallel evaluation strategy. Nodes represent unit cost computations, and edges represent sequential dependencies: If a node x contains a link to a node y then this means that y cannot start until x has finished.

Every computation graph has two distinguished nodes, known as the *source* and *minimum sink*. The *source* represents the start of the evaluation, while the *minimum sink* represents the computation that produces the result of the computation. When drawn graphically, subgraphs are drawn as triangles in which the top corner links to the source and the bottom left corner is linked to by the minimum sink. Unlike cost graphs, computation graphs contain only serialisation information: they do not contain values.

Figure 13.1 gives a computation graph for evaluation of the expression $E\ E'$, given that E evaluates to $\lambda x.E''$. This graph contains subgraphs for the evaluation of E, E' and E''. We can see that when the evaluation starts, the virtual machine will do one unit of work (represented by a black circle) and then start evaluation of both the function E and the argument E'. Once the function has been evaluated to a value, the virtual machine will do another unit of work, and

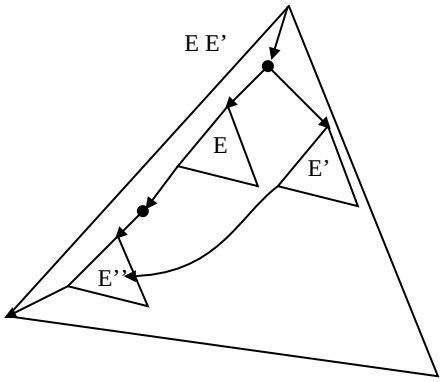


Figure 13.1: A PSL Computation Graph for $E\ E'$, given that E evaluates to $\lambda x.E''$

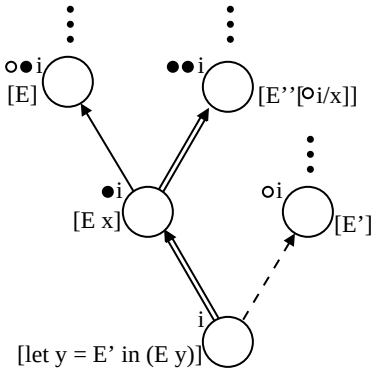


Figure 13.2: A Cost Graph for $\mathbf{let}\ y = E'\ \mathbf{in}\ E\ y$, given that E evaluates to $\lambda x.E''$

then start evaluation of the function body E''. The graph for the function body will be generated with x bound to the minimum sink of E'. From this graph, one can see that E and E' can be done in parallel, but that some computations within E'' may need to wait for E' to complete.

For comparison, Figure 13.2 shows a cost graph for the same program. We can see that there are some similarities between cost graphs and Computation Graphs, but there are also significant differences. Cost graphs describe *what* computations *must* take place; while computation graphs describe *when* computations *can* take place. The root node of a cost graph corresponds to both the source and minimum sink of a computation graph. The correspondence between the two approaches is illustrated by Figure 13.3 which shows the nodes from Figure 13.2 but with edges similar to those used by a computation graph.

Unlike cost graphs, computation graphs do not distinguish between eager edges and demand edges. Indeed, such a distinction does not make sense in their model as *(i)*, their semantics assumes that all computations in the computation graph are executed at some point, and *(ii)*, their semantics is data-driven rather than demand driven: a computation is assumed to start as soon as all the computations linking to it have finished. It is not obvious whether computation graphs could be easily adapted to describe a mixed eager/lazy evaluation strategy.

Another important difference between computation graphs and cost graphs is that a com-

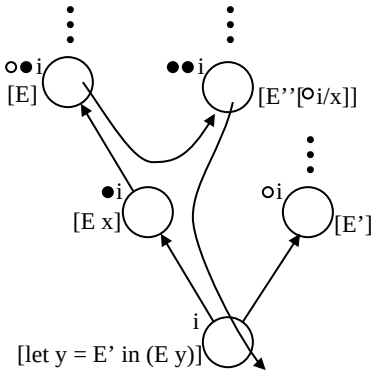


Figure 13.3: Half-way between a Cost Graph and a Computation Graph

putation graph cannot be infinite. This makes sense for computation graphs as it is assumed that every computation in the computation graph will be performed at some point; however it makes them unsuitable for our purposes, in which infinite-cost evaluations must be represented. Greiner and Blelloch present the semantics of computation graphs using a big-step operational semantics which explicitly performs all computations. It is not clear whether this semantics could be easily adapted to cope with infinite graphs.

13.5.2 Circuit Semantics

Circuit Semantics [Dan98, BD96] represent programs using directed graphs similar to those commonly drawn in electronics. Contrary to what the name might suggest, these graphs are acyclic. The basic idea is similar to that of PSL Computation Graphs in that links represent sequential dependencies between computational steps.[2]

Circuits are described only very informally, their main purpose being to graphically justify a step-counting semantics for data-driven parallel evaluation. This semantics is then used to compare the intensional expressiveness of various parallel operators.

13.5.3 Step Counting Models

Many previous models of cost work by instrumenting an evaluation semantics to include a count of the number of steps taken by evaluation. Such models are sometimes referred to as *profiling semantics*. Sands [San95a] gives an operational semantics for cost and uses it to prove various theorems about cost equivalence. Roe [Roe90] gives a denotational semantics that uses time-stamps to calculate the number of parallel steps required to calculate an expression using Lenient Evaluation. Rebon [RPHL02] and Rosendahl [Ros89] uses step counting to place bounds on the time and space complexity of a program. Santos [San95b] uses a step-counting semantics to define an ordering on expression costs which he uses to prove the efficiency of a number

[2] And indeed, both concepts have emerged from Carnegie Mellon University, so the similarity is unsurprising.

of optimisations. Sansom [San94] uses a step-counting semantics to define the behaviour of an offline execution profiler.

Unlike our Cost Graphs, such models are strongly tied to a particular evaluation strategy and are usually not composable.

13.5.4 Other Cost Models

The cost graphs that we produce are in some ways similar to the graphs used by graph reduction [Wad71, ET96], but graph reduction systems consider a graph to be an intermediate data-structure used in evaluation. They do not attempt to define a unique graph that represents the work done by an evaluation.

Wadler [Wad88] uses Strictness Analysis to determine that part of the result of a lazy expression that will be needed, and thus determine the cost of evaluating that expression.

13.5.5 Other Graphs

Our cost graphs bear similarities to the evaluation order independent structures produced by various lazy debuggers [NS97, WCBR01], however we are using these structures for a very different purpose.

Our cost graphs are also very similar to Dynamic Dependence Graphs [AH90]. Dynamic Dependence Graphs are specific to a particular evaluation strategy, and we are not aware of any formal model for them.

13.6 Profiling

Much work has been devoted to profilers and the way in which they can be used to direct optimisations; indeed there are entire workshops dedicated to the subject [FDD01, RV04]. It is only practical for us to cite a small subset of this work.

13.6.1 Sampled Profiling

The idea of profiling a program for only a small proportion of its execution has been around for a long time. One of the best known examples is GProf [GKM83], which looks at a program at a number of sample points and gathers statistics that tell the programmer where their program is spending its time. Many other sampling profilers exist, including Digital's DCPI [ABD+97], and Mortonosi's cache behaviour profiler [MGA93]. Other work avoids the overhead of profiling almost completely by using a separate processor to profile a program while it runs [ZS01].

The idea of having code that can actively switch between profiled and unprofiled modes is also well known. Arnold and Ryder [AR01] describe a system in which normal and profiled ver-

sions are compiled for every code block. Normally, the program will run in an unprofiled mode, but the runtime system can switch the program into profiled mode during profiling periods.

13.6.2 Feedback Directed Optimisation

Feedback Directed Optimisation [FDD01] is a widely used technique in static compilers. A program is run in a special profiling mode, recording statistics about the behaviour of the program. These statistics are then used by the compiler to make optimisation choices when compiling a final version of the program. Many commercial compilers use this technique. In principle we could do the same, compiling the configuration Σ into the program; we have not done so as this would make it harder for us to adapt our evaluation strategy at runtime.

13.6.3 Online Profiling

Online Profiling is used in many existing language implementations, including several implementations for the Java [GM96] language [Sun01, BCF+99, Arn02]. One of the first implementations to use such techniques was for Self [HÖ95]. These systems use similar techniques to Optimistic Evaluation, but do not apply them to laziness.

13.7 Debuggers for Non-Strict Languages

In Chapter 11 we described HsDebug: a debugger that is built on top of Optimistic Evaluation. There has been much prior work on debugging of non-strict languages.

13.7.1 Tracing

Most previous work on debugging of Lazy programs has focused on tracing. Systems such as Freja [NS97, NS96, Nil01], Buddha [Pop98] and Hat [SR97, WCBR01] augment a program so that it creates a trace as it executes. This trace gives a history of the evaluation that took place. For each value (e.g. 5), a link can be provided to the redex that evaluated to produce that value (e.g. $3 + 2$)) and to the larger evaluation that this evaluation was part of (e.g. $f(3)$).

Once such a trace has been built up, it can be explored in many different ways. Hat allows one to look at any object on the heap and find out how it came to be created. Other work allows evaluations to be observed in the order in which they would have taken place under Eager Evaluation, creating a similar environment to a traditional debugger [NF92, NF94].

Hat and Buddha run the program to completion before exploring the debug trace. While this simplifies the implementation, it makes debugging of IO awkward. In a traditional debugger, one can step over IO actions and observe the effect that the actions have on the outside world. This is made significantly more difficult if all actions take place before debugging starts; indeed,

Pope [Pop98] says that it is assumed that Buddha will only be applied to the sub-parts of a program that do not perform IO operations. Freja does better in this respect by building its trace while debugging.

One drawback of trace based debugging approaches is performance. If every evaluation is to be logged, then a very large amount of information must be recorded. Not only does the recording of such information take time—it also takes space. Freja works round this problem by only storing a small amount of trace information and then re-executing the program if more is needed, however this is quite tricky to implement, particular when IO is involved. There has also been considerable work on reducing the amount of trace information generated for redex trails [SR98].

HsDebug is definitely less powerful and less elegant than trace based debuggers. It is however simpler and faster, and does not require extra space.

13.7.2 Cost Centre Stacks

Cost Centre Stacks [MJ98, San94] extend a program so that it maintains a record of the call chain that the current expression would have, were it being evaluated strictly. The information obtainable from a cost centre stack is thus very similar to that available from the real stack under Optimistic Evaluation, or the trail created by Freja. Cost Centre Stacks were developed for use in profiling; however it is plausible that a debugger could be written that made use of them. Such a debugger could show the user the current cost-centre stack rather than the actual execution stack, providing the user experience of strict evaluation, without having to actually evaluate the program strictly. We believe that this approach may be worth exploring.

13.7.3 HOOD

HOOD [Gil00] can be seen as an extension of traditional “printf” debugging. The programmer adds annotations to the program that allow intermediate program states to be observed. HOOD goes a lot further than “printf” debugging by allowing lazy values and functions to be observed only to the extent to which they have been used. A sophisticated viewer application [Rei01] allows the programmer to view and manipulate traces resulting from an execution. While HOOD is extremely powerful, the need to add manual annotations can make it awkward to use.

13.7.4 Time Travel Debugging

Time Travel Debuggers extend the “stop, examine, continue” model further by allowing the program to run backwards to a breakpoint as well as forwards. Often one will find that the reason for something going wrong is that part of the program state has become invalid. In such a case, it can be extremely useful to run the program backwards from the point at which something

went wrong, to the point at which the state became invalid. Examples of time travel debuggers include the excellent O'Caml debugger [LRVD98] and the now sadly defunct SML/NJ debugger [TA95, TA90]. Many of the features of Time Travel Debugging can also be achieved by Tracing, and vice versa.

13.8 Reducing Space Usage

As we remarked in Section 12.6, some non-strict programs suffer from poor performance due to space leaks. In this section we discuss previous work that has attempted to prevent such space leaks.

13.8.1 Stingy Evaluation

Stingy Evaluation [vD89] is an evaluation strategy designed to reduce space leaks such as the one described in Section 12.6. When evaluating a `let` expression, or during garbage collection, the evaluator does a *little* bit of work on the expression, with the hope of evaluating it, and avoiding having to build a thunk. As with Eager Haskell, all expressions are eagerly evaluated, however the amount of evaluation done before abortion is significantly smaller, with only very simple evaluations allowed. Often this small amount of work will not be useful, causing some programs to run slower. Stingy evaluation was implemented in the LML [AJ89] compiler.

A more modest approach to the same problem is presented by Wadler [Wad87] who uses the garbage collector to evaluate any record selector whose record constructor has become a value. This avoids space leaks that can otherwise result if a record constructor contains links to other closures that are not of interest to the selector. Sparud [Spa93] solves many cases of the same problem by generating particularly clever code for pattern matches.

13.8.2 Heap Profiling

Rather than attempting to remove space leaks automatically, it is perhaps preferable to provide the programmer with a profiler which allows the programmer to understand why the space leak is occurring and which will show them how they might go about fixing it; this is the approach taken by heap profilers [RR96b, RR96a, RW92]. These profilers are extremely sophisticated and provide information that makes it fairly easy for a programmer to see which unevaluated thunks are causing them to leak memory. Given this information, the programmer can add annotations to their program to cause the offending expressions to be evaluated strictly.

13.9 Other Approaches to Evaluation

This thesis explores the space between Lazy Evaluation and Eager Evaluation. However there any many other evaluation strategies which do not fall into this space. In this Section, we discuss two such evaluation strategies: partial evaluation, and optimal evaluation.

13.9.1 Partial Evaluation

Partial Evaluation [CD93, JGS93, PEP97], also known as *Program Specialisation* is an evaluation technique that can perform evaluation steps underneath lambda expressions. Consider the following example:

$$\begin{array}{rll} f\ x\ y = & \textbf{case}\ x & \textbf{of} \\ & \quad \mathit{Red} \rightarrow y + 1 & \\ & \quad \mathit{Green} \rightarrow y + 4 & \end{array}$$

$$g\ y = \quad f\ \mathit{Green}\ y$$

A partial evaluator might partially evaluate g's call to f, giving the following definition for g:

$$g\ y = y + 4$$

Partial evaluation at compile time is used in many optimising compilers for mainstream languages such as C. It is also frequently done at runtime by dynamically optimising compilers for languages such as Java.

13.9.2 Optimal Lambda Reduction

Optimal Lambda Reduction [Lév78, Lév80, Lam90] is a fascinating approach to evaluation that aims to maximise the amount of work that is shared by different applications of a function.

Consider the following lambda expression (taken from [Lam90]):

$$\begin{array}{l} (\lambda g.(g(g(\lambda x.x)))) \\ \quad (\lambda h.(\\ \quad\quad (\lambda f.(f(f(\lambda z.z)))) \\ \quad\quad\quad (\lambda w.(h(w(\lambda y.y)))))) \end{array}$$

If we are to evaluate this using normal graph reduction, then there are two possible redexes in this expression.

Outer: Apply $(\lambda g.(g(g(\lambda x.x))))$ to the term beginning $\lambda h. \ldots$

Inner: Apply $(\lambda f.(f(f(\lambda z.z))))$ to the term beginning $\lambda w. \ldots$

If we perform the outer reduction, then, when g is applied twice, the inner reduction will have to be applied twice to reduce the function body. This is because, while graph reduction allows the sharing of terms, it does not allow the sharing of function bodies between functions that have been applied to different arguments.

Performing the inner reduction wastes work too. In this case, the two applications of f will mean that we have to perform the application of h twice. We can't do the application of h before doing the other two reductions as the value of h isn't known until we have performed the outer reduction. In this case, conventional graph reduction will duplicate work, irrespective of the evaluation order chosen.

The essential problem here is that conventional graph reduction copies the body of a function when the function is applied to an argument. This prevents the sharing of any reductions that might take place in this function body.

Optimal Lambda Reduction gets round this problem by arranging for the bodies of functions to be shared even after the function has been applied to several different arguments. This is accomplished using a complex system of *fan ins* and *fan outs* that arrange that variables in a function body be seen as being bound to different things depending on where the function body is being observed from.

While Optimal Reduction does minimise the number of beta reductions, it also introduces a large amount of book-keeping complexity that significantly reduces its efficiency. Asperti and Chroboezek [AC97] have shown that it is possible to create an implementation of Optimal Reduction that is capable of outperforming implementations of strict and lazy languages for some example programs. However the programs they use are quite unusual, and do not resemble the kinds of programs that people typically write. It is hard to know whether Optimal Reduction is capable of giving real performance increases for real programs. Asperti and Chroboezek suggest that the availability of Optimal Reduction could cause people to change their programming style to favour styles of programming that would otherwise be impractically inefficient - just as Lazy Evaluation has done in the past.

CHAPTER 14

Conclusions

In this chapter, we summarise the work described in this thesis, draw attention to its strengths and shortcomings, and discuss further work.

Performance

Optimistic Evaluation improves the performance of the Haskell programs we tested by an average of 20% relative to the best performing compiler previously available. Moreover, it does this without slowing any programs down by more than 7%. To put this into context, recall that Strictness Analysis improves performance by around 13% (Section 12.7.3), and that Optimistic Evaluation achieves its 20% speedup relative to a compiler that already uses Strictness Analysis.

We believe that it should be possible to improve the performance of Optimistic Evaluation further. In particular, we believe that we could achieve better performance if we designed a new compiler specifically for Optimistic Evaluation, rather than extending GHC.

Worst Case Behaviour

Despite being an adaptive evaluation technique, the behaviour of Optimistic Evaluation is well understood. Not only has it been demonstrated to be efficient on average, but, with the aid of a new denotational semantics, we have shown that we can bound its worst case behaviour with respect to a simplified cost model.

This does not mean that it is impossible that any program will go slower than predicted by this bound; our formal cost model is only approximate, and our profiler further approximates this model. However we are confident that neither of these approximations will ever be wrong

by more than a reasonable constant factor, and so believe that it is unlikely that any program can slow down by an unreasonably large amount.

While Optimistic Evaluation can make some programs go slower, its worst case behaviour is better than that of either Strictness Analysis or Cheapness Analysis, both of which are widely accepted. Perhaps more relevantly, we are not aware of any other dynamically optimising language implementation of similar complexity whose worst case behaviour is understood as well as that of Optimistic Evaluation.

Predictability

One criticism that could be levied against Optimistic Evaluation is that its performance is fairly unpredictable. Minor changes to a program, or even to its input data, can cause the profiler to react in a different way, causing performance to change. This problem is not unique to Optimistic Evaluation, but is a problem common to the majority of dynamically optimising language implementations.

While Optimistic Evaluation can cause programs to execute significantly faster than they would under Lazy Evaluation, we believe that it is unwise for a programmer to rely on this. If it is absolutely essential for the efficient execution of a program that a particular expression be evaluated eagerly, then we believe that the programmer should mark this with a strictness annotation.

Complexity

While Optimistic Evaluation does give good performance results, it has to go to a lot of effort to achieve this. Optimistic Evaluation is a very complex evaluation strategy to implement. Speculation, abortion, profiling, and semi-tagging are all complex ideas, all have knock-on effects throughout the runtime system, and all are difficult to debug when they go wrong. Although we had originally planned to merge Optimistic Evaluation into the main branch of GHC, it now seems likely that we will not do this—it is simply too difficult to maintain.

Non-Strict Languages

Perhaps the biggest weakness of Optimistic Evaluation is that it is an implementation technique for non-strict languages. During the course of this research, we have come to the conclusion that, although non-strict languages seem superficially appealing, they are not, in general, a good idea. While Lazy Evaluation is often useful, we do not believe that it is wise to make it the default evaluation strategy for all expressions. Although much has been written about the supposed expressive beauty of non-strict languages, most non-strict programs we have investigated contain only a small number of expressions for which laziness is useful, and it is usually

obvious which expressions these are. We now believe that, if a program makes essential use of laziness, then this should be a deliberate design decision, and the way in which laziness is used should be stated explicitly in the program text.

This does not however make Optimistic Evaluation completely redundant. Optimistic Evaluation obtains many of its biggest wins when applying chunky evaluation to infinite data structures — a technique that is equally applicable in languages with explicity laziness. In addition, Non-Strict languages are just one example of a general class of problems in which a computer may find itself having to decide whether or not to do some work that may or may not turn out to be useful. We thus believe that, even if it turns out that non-strict languages are a bad idea, the basic principles of Optimistic Evaluation still have practical value.

Summary

We have achieved what we set out to do: we have designed and implemented an evaluation strategy that significantly improves the performance of non-strict programs. If one wishes to evaluate non-strict programs fast, and one is prepared to expend considerable effort to do this, then we believe Optimistic Evaluation is a good technique to use. The biggest question is perhaps whether one should be implementing a non-strict language in the first place.

APPENDIX A

Proof that Meaning is Preserved

In this Appendix, we prove the property referred to in Section 4.4:

$$\Gamma; c; s \longrightarrow_{\Sigma} \Gamma', c'; s' \quad \Rightarrow \quad \mathcal{M}[\![\Gamma, c, s]\!] = \mathcal{M}[\![\Gamma', c', s']\!]$$

We proceed casewise, showing that the property holds for all rules defining $\longrightarrow$. For each rule, we show that the left hand side (LHS) of the rule has the same meaning as the right hand side (RHS) of the rule.

A.1 Evaluation Preserves Meaning

Case: *(VAL)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; V; s]\!] \\
&= \mathcal{M}[\![\Gamma[\alpha \mapsto (\!|V|\!)]; V; s]\!] && \text{fresh } \alpha \text{ and Theorem 4.3.3} \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha \mapsto (\!|V|\!)]]\!]_\rho \ \sqcup \ \mathcal{S}[\![s]\!]_\rho \, \mathcal{E}[\![V]\!]_\rho) \, \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha \mapsto (\!|V|\!)]]\!]_\rho \ \sqcup \ \mathcal{S}[\![s]\!]_\rho \, \mathcal{E}[\![\alpha]\!]_\rho) \, \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \text{ and } \mathcal{H}[\![-]\!] \\
&= \mathcal{M}[\![\Gamma[\alpha \mapsto V]; \triangledown\alpha; s]\!] \\
&= RHS
\end{aligned}$$

Case: *(VAR)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; \alpha; s]\!] \\
&= \mathcal{M}[\![\Gamma; \circledcirc\alpha; s]\!] \quad \text{defn of } \mathcal{M}[\![-]\!] \text{ and } \mathcal{C}[\![-]\!] \\
&= RHS
\end{aligned}$$

Case: *(DEM1)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma[\alpha \mapsto (\!|V|\!)]; \circledcirc\alpha; s]\!] \\
&= \mathcal{M}[\![\Gamma[\alpha \mapsto (\!|V|\!)]; \triangledown\alpha; s]\!] \quad \text{defn of } \mathcal{M}[\![-]\!] \text{ and } \mathcal{C}[\![-]\!] \\
&= RHS
\end{aligned}$$

Case: *(DEM2)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha; s]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha \mapsto \langle\alpha'\rangle]]\!]_\rho \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\alpha \mapsto \mathcal{E}[\![\alpha']\!]_\rho) \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{H}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\alpha \mapsto \mathcal{E}[\![\alpha']\!]_\rho) \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\alpha']\!]_\rho)\ \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \\
&= \mathcal{M}[\![\Gamma[\alpha \mapsto \langle\alpha'\rangle]; \circledcirc\alpha'; s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}$$

Case: *(DEM3)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma[\alpha \mapsto E]; \circledcirc\alpha; s]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha \mapsto E]]\!]_\rho \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\alpha \mapsto \mathcal{E}[\![E]\!]_\rho) \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{H}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\alpha \mapsto \mathcal{E}[\![E]\!]_\rho) \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![E]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \mathcal{S}[\![\#\alpha : s]\!]_\rho\, \mathcal{E}[\![E]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mathcal{M}[\![\Gamma; E; \#\alpha : s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}$$

Case: *(RESUME)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma[\alpha \mapsto \alpha'\angle l]; \circledcirc\alpha; s]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha \mapsto \alpha'\angle l]]\!]_\rho \ \sqcup\ (\mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\alpha \mapsto \mathcal{E}[\![\alpha'\angle l]\!]) \ \sqcup\ (\mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho))\ \epsilon && \text{defn of } \mathcal{H}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\alpha \mapsto \mathcal{E}[\![\alpha'\angle l]\!]) \ \sqcup\ (\mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\alpha'\angle l]\!]_\rho))\ \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\mathcal{S}[\![\#\alpha : s]\!]_\rho\, \mathcal{E}[\![\alpha'\angle l]\!]_\rho))\ \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\mathcal{S}[\![\#\alpha : s]\!]_\rho\, (\mathcal{L}[\![l]\!]_\rho\, \rho(\alpha'))))\ \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\mathcal{S}[\![l : \#\alpha : s]\!]_\rho\, \rho(\alpha')))\ \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mathcal{M}[\![\Gamma; \circledcirc\alpha'; l : \#\alpha : s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}$$

Case: *(UPD)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; \triangledown\alpha; \#\alpha' : s]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \mathcal{S}[\![\#\alpha' : s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\alpha' \mapsto \mathcal{E}[\![\alpha]\!]_\rho) \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha' \mapsto \langle\alpha\rangle]]\!]_\rho \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{H}[\![-]\!] \\
&= \mathcal{M}[\![\Gamma[\alpha' \mapsto \langle\alpha\rangle]; \triangledown\alpha; s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}
$$

Case: *(APP1)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; E\ \alpha; s]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![E\ \alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, (\mathcal{E}[\![E]\!]_\rho\, \rho(\alpha))\ )\ \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \mathcal{S}[\![@\alpha : s]\!]_\rho\, \mathcal{E}[\![E]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mathcal{M}[\![\Gamma; E; @\alpha : s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}
$$

Case: *(APP2)*

For convenience, let us define:

$$\rho' = (\alpha \mapsto \lambda v.\mathcal{E}[\![E]\!]_{\rho[x \mapsto v]})$$

$$
\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma[\alpha \mapsto (\!|\lambda x.E|\!)]; \triangledown\alpha; @\ \alpha' : s]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha \mapsto (\!|\lambda x.E|\!)]]\!]_\rho \ \sqcup\ \mathcal{S}[\![@\ \alpha' : s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \rho' \ \sqcup\ \mathcal{S}[\![@\ \alpha' : s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \text{ and } \mathcal{H}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \rho' \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, (\rho(\alpha)\ \rho(\alpha'))\ \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \text{ and } \mathcal{E}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \rho' \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, (\lambda v.\mathcal{E}[\![E]\!]_{\rho[x \mapsto v]}\ \rho(\alpha'))\ \epsilon && \text{defn of } \rho \text{ and } \rho' \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \rho' \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, (\mathcal{E}[\![E]\!]_{\rho[x \mapsto \rho(\alpha')]})\ )\ \epsilon && \text{beta reduction} \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \rho' \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![E[\alpha'/x]]\!]_{\rho[x \mapsto \rho(\alpha')]})\ \epsilon && \text{Theorem 4.3.1} \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ \rho' \ \sqcup\ \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![E[\alpha'/x]]\!]_\rho)\ \epsilon && \text{Theorem 4.3.2} \\
&= \mathcal{M}[\![\Gamma[\alpha \mapsto (\!|\lambda x.E|\!)]; E[\alpha'/x]; s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \text{ and } \mathcal{H}[\![-]\!] \\
&= RHS
\end{aligned}
$$

Case: *(LAZY)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; \mathbf{let}\ x = E\ \mathbf{in}\ E'; s]\!] \\
&= \mathcal{M}[\![\Gamma[\alpha \mapsto E]; \mathbf{let}\ x = E\ \mathbf{in}\ E'; s]\!] && \text{new } \alpha \text{ and Theorem 4.3.3} \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma[\alpha \mapsto E]]\!]_\rho \sqcup \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\mathbf{let}\ x = E\ \mathbf{in}\ E']\!]_\rho)\ \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma[\alpha \mapsto E]]\!]_\rho \sqcup \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![E']\!]_{\rho[x \mapsto \mathcal{E}[\![E]\!]_\rho]})\ \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma[\alpha \mapsto E]]\!]_\rho \sqcup \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![E'[\alpha/x]]\!]_{\rho[x \mapsto \mathcal{E}[\![E]\!]_\rho]})\ \epsilon && \text{Theorem 4.3.1 and defn of } \mathcal{H}[\![-]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma[\alpha \mapsto E]]\!]_\rho \sqcup \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![E'[\alpha/x]]\!]_\rho)\ \epsilon && \text{Theorem 4.3.2} \\
&= \mathcal{M}[\![\Gamma[\alpha \mapsto E]; E'[\alpha/x]; s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}
$$

Case: *(SPEC1)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; \mathbf{let}\ x = E\ \mathbf{in}\ E'; s]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma]\!]_\rho \sqcup \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![\mathbf{let}\ x = E\ \mathbf{in}\ E']\!]_\rho)\ \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma]\!]_\rho \sqcup \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![E']\!]_{\rho[x \mapsto \mathcal{E}[\![E]\!]_\rho]})\ \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma]\!]_\rho \sqcup \mathcal{S}[\![(\{x\}E') : s]\!]_\rho\, \mathcal{E}[\![E]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mathcal{M}[\![\Gamma; E; (\{x\}E' : s)]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}
$$

Case: *(SPEC2)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; \nabla\alpha; (\{x\}E) : s]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma]\!]_\rho \sqcup \mathcal{S}[\![(\{x\}E) : s]\!]_\rho\, \mathcal{E}[\![\alpha]\!]_\rho)\ \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma]\!]_\rho \sqcup \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![E]\!]_{\rho[x \mapsto \mathcal{E}[\![\alpha]\!]_\rho]})\ \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma]\!]_\rho \sqcup \mathcal{S}[\![s]\!]_\rho\, \mathcal{E}[\![E[\alpha/x]]\!]_\rho)\ \epsilon && \text{Theorem 4.3.1} \\
&= \mathcal{M}[\![\Gamma; E[\alpha/x]; s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}
$$

A.2 Abortion Preserves Meaning

We can similarly prove that all the rules defining '$\rightsquigarrow$' preserve the meaning of an expression with respect to $\mathcal{M}[\![-]\!]$. Formally, we prove:

$$\Gamma; c; s \rightsquigarrow \Gamma'; c'; s' \quad \Rightarrow \mathcal{M}[\![\Gamma; c; s]\!] = \mathcal{M}[\![\Gamma'; c'; s']\!]$$

Case: *(!EXP)*

The proof for *(!EXP)* is almost identical to the proof for *(VAL)*:

$$\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; E; s]\!] \\
&= \mathcal{M}[\![\Gamma[\alpha \mapsto E]; E; s]\!] && \text{new } \alpha \text{ and Theorem 4.3.3} \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha \mapsto E]]\!]_\rho \ \sqcup \ \mathcal{S}[\![s]\!]_\rho \, \mathcal{E}[\![E]\!]_\rho)\, \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha \mapsto E]]\!]_\rho \ \sqcup \ \mathcal{S}[\![s]\!]_\rho \, \mathcal{E}[\![\alpha]\!]_\rho)\, \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \text{ and } \mathcal{H}[\![-]\!] \\
&= \mathcal{M}[\![\Gamma[\alpha \mapsto E]; \circledcirc\alpha; s]\!] \\
&= RHS
\end{aligned}$$

Case: *(!RET)*

The proof for *(!RET)* is trivial:

$$\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; \triangledown\alpha; s]\!] \\
&= \mathcal{M}[\![\Gamma; \circledcirc\alpha; s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}$$

Case: *(!SPEC)*

The proof for *(!SPEC)* is almost identical to the proof for *(SPEC2)*:

$$\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; \circledcirc\alpha; (\{x\}E) : s]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup \ \mathcal{S}[\![(\{x\}E) : s]\!]_\rho \, \mathcal{E}[\![\alpha]\!]_\rho)\, \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup \ \mathcal{S}[\![s]\!]_\rho \, \mathcal{E}[\![E]\!]_{\rho[x \mapsto \mathcal{E}[\![\alpha]\!]_\rho]})\, \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mu\rho\ (\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup \ \mathcal{S}[\![s]\!]_\rho \, \mathcal{E}[\![E[\alpha/x]]\!]_\rho)\, \epsilon && \text{Theorem 4.3.1} \\
&= \mathcal{M}[\![\Gamma; E[\alpha/x]; s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}$$

Case: *(!UPD)*

The proof for *(!UPD)* is almost identical to the proof for *(UPD)*:

$$\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; \circledcirc\alpha; \#\alpha' : s]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup \ \mathcal{S}[\![\#\alpha' : s]\!]_\rho \mathcal{E}[\![\alpha]\!]_\rho)\, \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup \ (\alpha' \mapsto \mathcal{E}[\![\alpha]\!]_\rho) \ \sqcup \ \mathcal{S}[\![s]\!]_\rho \mathcal{E}[\![\alpha]\!]_\rho)\, \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha' \mapsto \langle\alpha\rangle]]\!]_\rho \ \sqcup \ \mathcal{S}[\![s]\!]_\rho \mathcal{E}[\![\alpha]\!]_\rho)\, \epsilon && \text{defn of } \mathcal{H}[\![-]\!] \\
&= \mathcal{M}[\![\Gamma[\alpha' \mapsto \langle\alpha\rangle]; \circledcirc\alpha; s]\!] && \text{defn of } \mathcal{M}[\![-]\!] \\
&= RHS
\end{aligned}$$

Case: *(!ABORT)*

The proof for *(!ABORT)* is almost the reverse of the proof for *(RESUME)*:

$$
\begin{aligned}
LHS &= \mathcal{M}[\![\Gamma; \circledcirc\alpha; (l : s)]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\mathcal{S}[\![l : s]\!]_\rho\ \rho(\alpha)))\ \epsilon && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\mathcal{S}[\![s]\!]_\rho\ (\mathcal{L}[\![l]\!]_\rho\ \rho(\alpha))))\ \epsilon && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\alpha' \mapsto \mathcal{L}[\![l]\!]_\rho\ \rho(\alpha)) \ \sqcup\ (\mathcal{S}[\![s]\!]_\rho\ \rho(\alpha')))\ \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \text{ for new } \alpha' \\
&= \mu\rho(\mathcal{H}[\![\Gamma]\!]_\rho \ \sqcup\ (\alpha' \mapsto \mathcal{E}[\![\alpha \angle l]\!]) \ \sqcup\ (\mathcal{S}[\![s]\!]_\rho\ \rho(\alpha')))\ \epsilon && \text{defn of } \mathcal{E}[\![-]\!] \\
&= \mu\rho(\mathcal{H}[\![\Gamma[\alpha' \mapsto \alpha \angle l]]\!]_\rho \ \sqcup\ (\mathcal{S}[\![s]\!]_\rho\ \rho(\alpha')))\ \epsilon && \text{defn of } \mathcal{H}[\![-]\!] \\
&= \mathcal{M}[\![\Gamma[\alpha' \mapsto \alpha \angle l]; \circledcirc\alpha'; s]\!] && \text{defn of } \mathcal{M}[\![s]\!] \\
&= RHS
\end{aligned}
$$

Soundness of Bounded Evaluation

Having shown that $\longrightarrow$ and $\rightsquigarrow$ are sound, it easy easy to see that $\curvearrowright$ must also be sound as every $\curvearrowright$ transition corresponds to either an $\longrightarrow$ transition or an $\rightsquigarrow$ transition.

APPENDIX B

Proof that Costed Meaning is Preserved

In this appendix, we prove the property referred to in Section 5.6.4:

$$T; c; s \longrightarrow T'; c'; s' \;\; \Rightarrow \;\; \mathcal{M}[\![T; c; s]\!] = \mathcal{M}[\![T'; c'; s']\!]$$

We start with two important lemmas, and then proceed to show that the meaning of the state is preserved by each rule defining '$\longrightarrow$'.

B.1 Lemmas

Lemma B.1.1 (Following Indirections)
From the definitions of $\mathcal{S}[\![-]\!]$, $\mathcal{F}(-)$, and $\lfloor - \rfloor$ we can observe that:

$$\gamma(i) = (i', d, e) \;\; \Rightarrow \;\; \mathcal{S}[\![s]\!]_\gamma \; i \; = \; \mathcal{S}[\![s]\!]_\gamma \; i'$$

Lemma B.1.2 (Substitution)
From the definition of $\mathcal{E}[\![-]\!]$ and the extension given in Section 5.6.1, we can observe that:

$$\mathcal{E}[\![E[i'/x]]\!]_\beta \; i \; \gamma \;\; = \;\; \mathcal{E}[\![E]\!]_{\beta[x \mapsto i']} \; i \; \gamma$$

B.2 Proof of Soundness for Evaluation Rules

Case: *(VAL)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![T; V \rhd i; s]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![V]\!]_\emptyset\ i\ \gamma && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[i \mapsto V]]\!] \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) && \text{defn of } \mathcal{T}[\![-]\!] \\
&= \mathcal{M}[\![T[i \mapsto V]; \triangledown i; s]\!] \ =\ RHS && \text{defn of } \mathcal{M}[\![-]\!]
\end{aligned}$$

Case: *(VAR)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![T; i' \rhd i; s]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![i']\!]_\emptyset\ i\ \gamma && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup (i \mapsto (i', \emptyset, \emptyset)) && \text{defn of } \mathcal{E}[\![-]\!] \text{ and Section 5.6.1} \\
&= \mu\gamma.\ \mathcal{T}[\![T[i \mapsto \langle i', \emptyset, \emptyset\rangle]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i') && \text{defn of } \mathcal{T}[\![-]\!] \text{ and Lemma B.1.1} \\
&= \mathcal{M}[\![T[i \mapsto \langle i', \emptyset, \emptyset\rangle]; \circledcirc i'; s]\!] \ =\ RHS && \text{defn of } \mathcal{M}[\![-]\!]
\end{aligned}$$

Case: *(DEM1)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![T[i \mapsto (\!|V|\!)]; \circledcirc i; s]\!] \\
&= \mathcal{M}[\![T[i \mapsto (\!|V|\!)]; \triangledown i; s]\!] \ =\ RHS && \text{defn of } \mathcal{M}[\![-]\!]
\end{aligned}$$

Case: *(DEM2)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![T[i \mapsto \langle i', d, e\rangle]; \circledcirc i; s]\!] \\
&= \mathcal{M}[\![T[i \mapsto \langle i', d, e\rangle]; \circledcirc i'; s]\!] \ =\ RHS && \text{defn of } \mathcal{M}[\![-]\!] \text{ and Lemma B.1.1}
\end{aligned}$$

Case: *(DEM3)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![T[i \mapsto E]; \circledcirc i; s]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[i \mapsto E]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ i\ \gamma && \text{defn of } \mathcal{T}[\![-]\!] \\
&= \mathcal{M}[\![T; E \rhd i; s]\!] \ =\ RHS && \text{defn of } \mathcal{M}[\![-]\!]
\end{aligned}$$

Case: *(APP1)*

$$\begin{aligned}
LHS &= \mathcal{M}[\![T; E\ i' \rhd i; s]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![E\ i']\!]_\emptyset\ i\ \gamma && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup (i \mapsto (\bullet i, \{\circ i\}, \emptyset)) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma \sqcup \mathcal{F}(\circ i)^{\bullet i}_{i'} && \text{defn of } \mathcal{E}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[i \mapsto \langle \bullet i, \{\circ i\}, \emptyset\rangle]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma \sqcup \mathcal{F}(\circ i)^{\bullet i}_{i'} && \text{defn of } \mathcal{T}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[i \mapsto \langle \bullet i, \{\circ i\}, \emptyset\rangle]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ \bullet i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma \sqcup \mathcal{F}(\circ i)^{\bullet i}_{i'} && \text{Lemma B.1.1} \\
&= \mu\gamma.\ \mathcal{T}[\![T[i \mapsto \langle \bullet i, \{\circ i\}, \emptyset\rangle]]\!]_\gamma \sqcup (\mathcal{S}[\![(@\ i', \bullet i) : s]\!]_\gamma\ \circ i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mathcal{M}[\![T[i \mapsto \langle \bullet i, \{\circ i\}, \emptyset\rangle]; E \rhd \circ i; (@\ i', \bullet i) : s]\!] \ =\ RHS && \text{defn of } \mathcal{M}[\![-]\!]
\end{aligned}$$

Case: *(APP2)*

$$
\begin{array}{lll}
LHS = \mathcal{M}[\![T[i' \mapsto (\!|\lambda x.E|\!)]; \triangledown i'; (@\ i'', i) : s]\!] & & \\
\quad = \mu\gamma.\ \mathcal{T}[\![T[i' \mapsto (\!|\lambda x.E|\!)]]\!]_\gamma \sqcup (\mathcal{S}[\![(@\ i'', i) : s)]\!]_\gamma\ i') & \text{defn of } \mathcal{M}[\![-]\!] \\
\quad = \mu\gamma.\ \mathcal{T}[\![T[i' \mapsto (\!|\lambda x.E|\!)]]\!]_\gamma \sqcup \mathcal{F}(i')^i_{i''}\ \gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) & \text{defn of } \mathcal{S}[\![-]\!] \\
\quad = \mu\gamma.\ \mathcal{T}[\![T[i' \mapsto (\!|\lambda x.E|\!)]]\!]_\gamma \sqcup \mathcal{E}[\![E]\!]_{(x \mapsto i'')}\ i\ \gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) & \text{defn of } \mathcal{F}(-) \text{ and } \mathcal{T}[\![-]\!] \\
\quad = \mu\gamma.\ \mathcal{T}[\![T[i' \mapsto (\!|\lambda x.E|\!)]]\!]_\gamma \sqcup \mathcal{E}[\![E[i''/x]]\!]_\emptyset\ i\ \gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) & \text{Lemma B.1.2} \\
\quad = \mathcal{M}[\![T[i' \mapsto (\!|\lambda x.E|\!)]; E[i''/x] \rhd i; s]\!] \ =\ RHS & \text{defn of } \mathcal{M}[\![-]\!]
\end{array}
$$

Case: *(LAZY)*

$$
\begin{array}{lll}
LHS = \mathcal{M}[\![T; \mathbf{let}\ x = E\ \mathbf{in}\ E' \rhd i; s]\!] & & \\
\quad = \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![\mathbf{let}\ x = E\ \mathbf{in}\ E']\!]_\emptyset\ i\ \gamma & \text{defn of } \mathcal{M}[\![-] \\
\quad = \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup (i \mapsto (\bullet i, \emptyset, \{\circ i\})) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma \sqcup \mathcal{E}[\![E']\!]_{(x \mapsto \circ i)}\ \bullet i\ \gamma & \text{defn of } \mathcal{E}[\![-]\!] \\
\quad = \mu\gamma.\ \mathcal{T}[\![T[i \mapsto (\bullet i, \emptyset, \{\circ i\}), \circ i \mapsto E]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![E']\!]_{(x \mapsto \circ i)}\ \bullet i\ \gamma & \text{defn of } \mathcal{T}[\![-]\!] \\
\quad = \mu\gamma.\ \mathcal{T}[\![T[i \mapsto (\bullet i, \emptyset, \{\circ i\}), \circ i \mapsto E]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![E'[\circ i/x]]\!]_\emptyset\ \bullet i\ \gamma & \text{Lemma B.1.2} \\
\quad = \mathcal{M}[\![T[i \mapsto (\bullet i, \emptyset, \{\circ i\}), \circ i \mapsto E]; E'[\circ i/x] \rhd \bullet i; s]\!] \ =\ RHS & \text{defn of } \mathcal{M}[\![-]
\end{array}
$$

Case: *(SPEC1)*

$$
\begin{array}{lll}
LHS = \mathcal{M}[\![T; \mathbf{let}\ x = E\ \mathbf{in}\ E' \rhd i; s]\!] & & \\
\quad = \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![\mathbf{let}\ x = E\ \mathbf{in}\ E']\!]_\emptyset\ i\ \gamma & \text{defn of } \mathcal{M}[\![-] \\
\quad = \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup (i \mapsto (\bullet i, \emptyset, \{\circ i\})) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma \sqcup \mathcal{E}[\![E']\!]_{(x \mapsto \circ i)}\ \bullet i\ \gamma & \text{defn of } \mathcal{E}[\![-]\!] \\
\quad = \mu\gamma.\ \mathcal{T}[\![T[i \mapsto (\bullet i, \emptyset, \{\circ i\})]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma \sqcup \mathcal{E}[\![E']\!]_{(x \mapsto \circ i)}\ \bullet i\ \gamma & \text{defn of } \mathcal{T}[\![-]\!] \\
\quad = \mu\gamma.\ \mathcal{T}[\![T[i \mapsto (\bullet i, \emptyset, \{\circ i\})]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma \sqcup \mathcal{E}[\![E'[\circ i/x]]\!]_\emptyset\ \bullet i\ \gamma & \text{Lemma B.1.2} \\
\quad = \mu\gamma.\ \mathcal{T}[\![T[i \mapsto (\bullet i, \emptyset, \{\circ i\})]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ \bullet i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma \sqcup \mathcal{E}[\![E'[\circ i/x]]\!]_\emptyset\ \bullet i\ \gamma & \text{Lemma B.1.1} \\
\quad = \mu\gamma.\ \mathcal{T}[\![T[i \mapsto (\bullet i, \emptyset, \{\circ i\})]]\!]_\gamma \sqcup (\mathcal{S}[\![(E'[\circ i/x] \rhd \bullet i) : s]\!]_\gamma\ \circ i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma & \text{defn of } \mathcal{S}[\![-]\!] \\
\quad = \mathcal{M}[\![T[i \mapsto (\bullet i, \emptyset, \{\circ i\})]; E \rhd \circ i; (E'[\circ i/x] \rhd \bullet i) : s]\!] \ =\ RHS & \text{defn of } \mathcal{M}[\![-]
\end{array}
$$

Case: *(SPEC2)*

$$
\begin{array}{lll}
LHS = \mathcal{M}[\![T; \triangledown i'; E \rhd i : s]\!] & & \\
\quad = \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![(E \rhd i) : s]\!]_\gamma\ i') & \text{defn of } \mathcal{M}[\![-]\!] \\
\quad = \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ i\ \gamma & \text{defn of } \mathcal{S}[\![-]\!] \\
\quad = \mathcal{M}[\![T; E \rhd i; s]\!] \ =\ RHS & \text{defn of } \mathcal{M}[\![-]\!]
\end{array}
$$

Case: *(OP1)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![T; j \ \otimes \ k \rhd i; s]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![j \otimes k]\!]_\emptyset\ i\ \gamma && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup (i \mapsto (\bullet i, \{j,k\}, \emptyset)) \sqcup (\mathcal{O}[\![\otimes]\!]\ \bullet i\ \lfloor j \rfloor_\gamma\ \lfloor k \rfloor_\gamma) && \text{defn of } \mathcal{E}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[i \mapsto (\bullet i, \{j,k\}, \emptyset)]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup (\mathcal{O}[\![\otimes]\!]\ \bullet i\ \lfloor j \rfloor_\gamma\ \lfloor k \rfloor_\gamma) && \text{defn of } \mathcal{T}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[i \mapsto (\bullet i, \{j,k\}, \emptyset)]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ \bullet i) \sqcup (\mathcal{O}[\![\otimes]\!]\ \bullet i\ \lfloor j \rfloor_\gamma\ \lfloor k \rfloor_\gamma) && \text{Lemma B.1.1} \\
&= \mu\gamma.\ \mathcal{T}[\![T[i \mapsto (\bullet i, \{j,k\}, \emptyset)]]\!]_\gamma \sqcup (\mathcal{S}[\![(\otimes k, \bullet i) : s]\!]_\gamma\ j) && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mathcal{M}[\![T[i \mapsto (\bullet i, \{j,k\}, \emptyset)]; \circledcirc j; (\otimes k, \bullet i) : s]\!] \ = \ RHS && \text{defn of } \mathcal{M}[\![-]\!].
\end{aligned}
$$

Case: *(OP2)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![T[j \mapsto (\!|n|\!)]; \triangledown j; (\otimes k, i) : s]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[j \mapsto (\!|n|\!)]]\!]_\gamma \sqcup (\mathcal{S}[\![(\otimes k, i) : s]\!]_\gamma\ j) && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[j \mapsto (\!|n|\!)]]\!]_\gamma \sqcup (\mathcal{O}[\![\otimes]\!]\ i\ \lfloor j \rfloor_\gamma\ \lfloor k \rfloor_\gamma) \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[j \mapsto (\!|n|\!)]]\!]_\gamma \sqcup (\mathcal{O}[\![\otimes]\!]\ i\ n\ \lfloor k \rfloor_\gamma) \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) && \text{defn of } \lfloor - \rfloor \text{ and } \mathcal{T}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[j \mapsto (\!|n|\!)]]\!]_\gamma \sqcup (\mathcal{S}[\![(n\otimes, i) : s]\!]_\gamma\ k) && \text{defn of } \lfloor - \rfloor \text{ and } \mathcal{T}[\![-]\!] \\
&= \mathcal{M}[\![T[j \mapsto (\!|n|\!)]; \circledcirc k; (n\otimes, i) : s]\!] \ = \ RHS && \text{defn of } \mathcal{M}[\![-]\!]
\end{aligned}
$$

Case: *(OP3)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![T[k \mapsto (\!|n'|\!)]; \triangledown k; (n\otimes, i) : s]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[k \mapsto (\!|n'|\!)]]\!]_\gamma \sqcup (\mathcal{S}[\![(n\otimes, i) : s]\!]_\gamma\ k) && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[k \mapsto (\!|n'|\!)]]\!]_\gamma \sqcup (\mathcal{O}[\![\otimes]\!]\ i\ n\ \lfloor k \rfloor_\gamma) \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[k \mapsto (\!|n'|\!)]]\!]_\gamma \sqcup (\mathcal{O}[\![\otimes]\!]\ i\ n\ n') \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) && \text{defn of } \lfloor - \rfloor \text{ and } \mathcal{T}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[k \mapsto (\!|n'|\!), i \mapsto (\!|n \tilde{\otimes} n'|\!)]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) && \text{defn of } \mathcal{T}[\![-]\!] \text{ and } \mathcal{O}[\![-]\!] \\
&= \mathcal{M}[\![T[k \mapsto (\!|n'|\!), i \mapsto (\!|n \tilde{\otimes} n'|\!)]; \triangledown i; s]\!] \ = \ RHS && \text{defn of } \mathcal{M}[\![-]\!]
\end{aligned}
$$

B.3 Proof of Soundness for Abortion Rules

Case: *(!EXP)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![T; E \rhd i; s]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) \sqcup \mathcal{E}[\![E]\!]_\emptyset\ i\ \gamma && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[i \mapsto E]]\!]_\gamma \sqcup (\mathcal{S}[\![s]\!]_\gamma\ i) && \text{defn of } \mathcal{T}[\![-]\!] \\
&= \mathcal{M}[\![T[i \mapsto E]; \circledcirc i; s]\!] \ = \ RHS && \text{defn of } \mathcal{M}[\![-]\!]
\end{aligned}
$$

Case: *(!SPEC)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![T; \circledcirc i; E \rhd j : s]\!] \\
&= \mathcal{M}[\![T; \triangledown i; E \rhd j : s]\!] \ = \ RHS \quad \text{defn of } \mathcal{M}[\![-]\!]
\end{aligned}
$$

Case: *(!ABORT)*

$$
\begin{aligned}
LHS &= \mathcal{M}[\![T; \circledcirc i; (f, j) : s]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![(f, j) : s]\!]_\gamma\ i && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T]\!]_\gamma \sqcup (\mathcal{S}[\![(f, j) : []]\!]_\gamma\ i) \sqcup (\mathcal{S}[\![s]\!]_\gamma\ j) && \text{defn of } \mathcal{S}[\![-]\!] \\
&= \mu\gamma.\ \mathcal{T}[\![T[j \mapsto i \angle f]]\!]_\gamma \sqcup\ \sqcup\ (\mathcal{S}[\![s]\!]_\gamma\ j && \text{defn of } \mathcal{T}[\![-]\!] \\
&= \mathcal{M}[\![T[j \mapsto i \angle f]; \circledcirc j; s]\!] = RHS && \text{defn of } \mathcal{M}[\![-]\!]
\end{aligned}
$$

Case: *(!RESUME)*

This is simply the inverse of the proof for *(!ABORT)*.

APPENDIX C

Proof that $programWork$ predicts $workDone$

C.1 Preliminaries

In this appendix, we provide the proof referred to in Section 5.6.5. That is, we prove:

$$\emptyset; E \rhd \epsilon; [\,] \longrightarrow^* T; c; s \;\Rightarrow\; workDone(T) \cup pendingWork(T; c; s, \Psi) \subseteq programWork(\mathcal{M}[\![T; c; s]\!], \Psi)$$

We prove this by induction over the number of $\longrightarrow$ transitions that are applied. In the base case, no $\longrightarrow$ transitions are applied, and $T; c; s = \emptyset; E \rhd \epsilon; [\,]$. The proof is thus trivial:

$$\begin{array}{lll}
& workDone(\emptyset) \cup pendingWork(\emptyset; E \rhd \epsilon; [\,], \Psi) & \\
= & \emptyset \cup pendingWork(\emptyset; E \rhd \epsilon; [\,], \Psi) & \text{defn of } workDone \\
= & \mathcal{W}\{\!\{\mathcal{C}\{\!\{E \rhd \epsilon\}\!\} \cup \mathcal{S}\{\!\{[\,]\}\!\}\}\!\}^{\Psi}_{\mathcal{M}[\![\emptyset; E \rhd \epsilon; [\,]]\!]} & \text{defn of } pendingWork \\
= & \mathcal{W}\{\!\{\{\epsilon\}\}\!\}^{\Psi}_{\mathcal{M}[\![\emptyset; E \rhd \epsilon; [\,]]\!]} & \text{defn of } \mathcal{C}\{\!\{-\}\!\} \text{ and } \mathcal{S}\{\!\{-\}\!\} \\
= & programWork(\mathcal{M}[\![\emptyset; E \rhd \epsilon; [\,]]\!]) & \text{defn of } programWork
\end{array}$$

The proof of the inductive step is rather harder. We know from the proof in Appendix B that $programWork(\mathcal{M}[\![T; c; s]\!], \Psi)$ will be preserved by evaluation transitions. The inductive step thus amounts to showing that the union of $workDone$ and $pendingWork$ never increases. That is:

$$\begin{array}{l}
\emptyset; E \rhd \epsilon; [\,] \longrightarrow^* T; c; s \;\wedge\; T; c; s \longrightarrow T'; c'; s' \;\Rightarrow \\
\qquad workDone(T) \cup pendingWork(T; c; s) \\
\qquad\qquad \subseteq \\
\qquad workDone(T') \cup pendingWork(T'; c'; s')
\end{array}$$

We proceed casewise by showing that, this property holds for every rule defining $\longrightarrow$.

For convenience, we make the following standard definitions for each rule, where, in each case, $T; c; s$ represents the old state, and $T'; c'; s'$ represents the new state:

D	$=$	$workDone(T)$	the work done by the initial state
D'	$=$	$workDone(T')$	the work done by the final state
P	$=$	$pendingWork(T; c; s, \Psi)$	the pending work for the initial state
P'	$=$	$pendingWork(T'; c'; s', \Psi)$	the pending work for the final state
γ	$=$	$\mathcal{M}[\![T; C; s]\!] \;=\; \mathcal{M}[\![T'; c'; s']\!]$	complete cost view (preserved by Appendix B

We now proceed to give the proof for each rule. Most of the the proofs follow the same pattern and are fairly trivial. We give a detailed proof for the first rule, but give less detail in the other rules.

C.2 Proof for Evaluation Transitions

Case: *(VAL)*

$$T; V \rhd i; s \;\longrightarrow\; T[i \mapsto V]; \triangledown i; s$$

In this case γ is defined as:

$$\begin{aligned}
\gamma &= \mathcal{M}[\![T, V \rhd i; s]\!] \\
&= \mu\gamma \,.\, \mathcal{T}[\![T]\!]_\gamma \;\sqcup\; (\mathcal{S}[\![s]\!]_\gamma\, i) \;\sqcup\; \mathcal{E}[\![V]\!]_\emptyset\, i\, \gamma && \text{defn of } \mathcal{M}[\![-]\!] \\
&= \mu\gamma \,.\, \mathcal{T}[\![T]\!]_\gamma \;\sqcup\; (\mathcal{S}[\![s]\!]_\gamma\, i) \;\sqcup\; (i \mapsto v) && \text{defn of } \mathcal{E}[\![-]\!] \text{ for some value } v
\end{aligned}$$

We can use this to prove that *(VAL)* preserves the union of $workDone$ and $pendingWork$:

$$\begin{aligned}
LHS &= D \;\cup\; \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^\Psi_\gamma && \text{defn of } pendingWork \\
&= D \;\cup\; \{i\} \;\cup\; \mathcal{W}\{\!\{\mathcal{S}\{\!\{s\}\!\}\}\!\}^\Psi_\gamma && \text{defn of } \mathcal{W}\{\!\{-\}\!\} \text{ and } \gamma \\
&= D' \;\cup\; \mathcal{W}\{\!\{\mathcal{S}\{\!\{s\}\!\}\}\!\}^\Psi_\gamma && \text{defn of } workDone \\
&= D' \;\cup\; P' \;=\; RHS && \text{defn of } pendingWork
\end{aligned}$$

Case: *(VAR)*

$$T; i' \rhd i; s \;\longrightarrow\; T[i \mapsto \langle i', \emptyset, \emptyset\rangle]; \circledcirc i'; s$$

$$\begin{aligned}
LHS &= D \;\cup\; \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^\Psi_\gamma && \text{defn of } pendingWork \\
&= D \;\cup\; \{i\} \;\cup\; \mathcal{W}\{\!\{\{i'\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^\Psi_\gamma && \text{defn } \mathcal{W}\{\!\{-\}\!\} \text{ and } \gamma \\
&= D' \;\cup\; P' \;=\; RHS && \text{defn of } workDone \text{ and } pendingWork
\end{aligned}$$

Case: *(DEM1)*

$$T[i \mapsto (\!|V|\!)]; \circledcirc i; s \;\longrightarrow\; T[i \mapsto (\!|V|\!)]; \triangledown i; s$$

$$\begin{array}{lll} LHS = D \cup \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} & & \text{defn of } pendingWork \\ \quad = D \cup \{i\} \cup \mathcal{W}\{\!\{\mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} & & \text{defn of } \mathcal{W}\{\!\{-\}\!\} \\ \quad = D' \cup P' = RHS & & \text{defn of } workDone \text{ and } pendingWork \end{array}$$

Case: *(DEM2)*

$$T[i \mapsto (i', d, e)]; \circledcirc i; s \;\longrightarrow\; T[i \mapsto (i', d, e)]; \circledcirc i'; s$$

We are currently only able to prove that *(DEM2)* reduces the amount of work expected, rather than that it preserves it, however we believe that, in the absence of abortion, *(DEM2)* should preserve expected work just like all the other evaluation rules do.

$$\begin{array}{lll} LHS = D \cup \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} & & \text{defn of } pendingWork \\ \quad \supseteq D \cup \mathcal{W}\{\!\{\{i'\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} & & \text{defn of } \mathcal{W}\{\!\{-\}\!\} \text{ and } \gamma \\ \quad = D' \cup P' = RHS & & \text{defn of } workDone \text{ and } pendingWork \end{array}$$

Case: *(DEM3)*

$$T[i \mapsto E]; \circledcirc i; s \;\longrightarrow\; T; E \rhd i; s$$

$$\begin{array}{lll} LHS = D \cup \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} & & \text{defn of } pendingWork \\ \quad = D' \cup \{i\} \cup \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} & & \text{defn of } workDone \\ \quad = D' \cup P' = RHS & & \text{defn of } pendingWork \text{ and } \mathcal{W}\{\!\{-\}\!\} \end{array}$$

Case: *(APP1)*

$$T; E\ i' \rhd i; s \;\longrightarrow\; T[i \mapsto \langle \bullet i, \{\circ i\}, \emptyset\rangle]; E \rhd \circ i; (@\ i', \bullet i) : s$$

$$\begin{array}{lll} LHS = D \cup \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} & & \text{defn of } pendingWork \\ \quad = D \cup \{i\} \cup \mathcal{W}\{\!\{\{\circ i, \bullet i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} & & \text{defn of } \mathcal{W}\{\!\{-\}\!\} \text{ and } \gamma \\ \quad = D' \cup P' = RHS & & \text{defn of } workDone \text{ and } pendingWork \end{array}$$

Case: *(APP2)*

$$T[i' \mapsto (\!|\lambda x.E|\!)]; \triangledown i'; (@\ i'', i) : s \;\longrightarrow\; T[i' \mapsto (\!|\lambda x.E|\!)]; E[i''/x] \rhd i; s$$

$$\begin{array}{lll} LHS = D \cup \mathcal{W}\{\!\{\{i', i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} & & \text{defn of } pendingWork \\ \quad = D \cup \{i'\} \cup \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} & & \text{defn of } \mathcal{W}\{\!\{-\}\!\} \text{ and } \gamma \\ \quad = D' \cup P' = RHS & & \text{defn of } workDone \text{ and } pendingWork \end{array}$$

Case: *(LAZY)*

We give rule *(LAZY)* in more detail, as it is more interesting than most other rules:

$$\begin{array}{rcl} T; \mathbf{let}\ x = E\ \mathbf{in}\ E' \rhd i; s & \longrightarrow & T[i \mapsto \langle \bullet i, \emptyset, \{\circ i\}\rangle, \circ i \mapsto E]; E'[\circ i/x] \rhd \bullet i; s \\ i & \notin & \Psi \end{array}$$

In this case, γ is defined as:

$$\begin{array}{lll} \gamma = \mathcal{M}[\![T, \mathbf{let}\ x = E\ \mathbf{in}\ E' \rhd i; s]\!] & & \\ \quad = \mu\gamma\ .\ \mathcal{T}[\![T]\!]_\gamma\ \sqcup\ (\mathcal{S}[\![s]\!]_\gamma\ i)\ \sqcup\ \mathcal{E}[\![\mathbf{let}\ x = E\ \mathbf{in}\ E']\!]_\emptyset\ i\ \gamma & & \text{defn of } \mathcal{M}[\![-]\!] \\ \quad = \mu\gamma\ .\ \mathcal{T}[\![T]\!]_\gamma\ \sqcup\ (\mathcal{S}[\![s]\!]_\gamma\ i)\ \sqcup\ (i \mapsto (\bullet i, \emptyset, \{\circ i\}))\ \sqcup\ \mathcal{E}[\![E]\!]_\emptyset\ \circ i\ \gamma\ \sqcup\ \mathcal{E}[\![E']\!]_\emptyset\ \bullet i\ \gamma & & \text{defn of } \mathcal{E}[\![-]\!] \end{array}$$

We can use this to prove that *(LAZY)* preserves the union of $workDone$ and $pendingWork$:

$$\begin{array}{lll} LHS = D\ \cup\ P & & \\ \quad = D\ \cup\ \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^\Psi_\gamma & & \text{defn of } \mathcal{C}\{\!\{-\}\!\} \\ \quad = D\ \cup\ \{i\}\ \cup\ \mathcal{W}\{\!\{\{\bullet i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^\Psi_\gamma & & \text{defn of } \mathcal{W}\{\!\{-\}\!\} \text{ and } \gamma, \text{ given } i \notin \Psi \\ \quad = D'\ \cup\ P'\ =\ RHS & & \text{defn of } workDone \text{ and } pendingWork \end{array}$$

Case: *(SPEC1)*

$$T; \mathbf{let}\ x = E\ \mathbf{in}\ E' \rhd i; s\ \longrightarrow\ T[i \mapsto \langle \bullet i, \emptyset, \{\circ i\}\rangle]; E \rhd \circ i; (E'[\circ i/x] \rhd \bullet i) : s$$

Rule *(SPEC1)* reuses the expansion of γ that we gave for *(LAZY)*:

$$\begin{array}{lll} LHS = D\ \cup\ P & & \\ \quad = D\ \cup\ \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^\Psi_\gamma & & \text{defn of } \mathcal{C}\{\!\{-\}\!\} \\ \quad = D\ \cup\ \{i\}\ \cup\ \mathcal{W}\{\!\{\{\circ i, \bullet i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^\Psi_\gamma & & \text{defn of } \mathcal{W}\{\!\{-\}\!\} \text{ and } \gamma, \text{ given } i \in \Psi \\ \quad = D'\ \cup\ P'\ =\ RHS & & \text{defn of } workDone \text{ and } pendingWork \end{array}$$

Case: *(SPEC2)*

$$\begin{array}{rcl} T; \triangledown i'; E \rhd i : s & \longrightarrow & T; E \rhd i; s \\ i & \in & \Psi \end{array}$$

$$\begin{array}{lll} LHS = D\ \cup\ \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^\Psi_\gamma & & \text{defn of } pendingWork \\ \quad = D'\ \cup\ P'\ =\ RHS & & \text{defn of } workDone \text{ and } pendingWork \end{array}$$

C.3 Proof for Abortion Transitions

Case: *(!EXP)*

$$T; E \rhd i; s \;\longrightarrow\; T[i \mapsto E]; \circledcirc i; s$$

$$\begin{aligned} LHS &= D \;\cup\; \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} && \text{defn of } pendingWork \\ &= D' \;\cup\; P' \;=\; RHS && \text{defn of } workDone \text{ and } pendingWork \end{aligned}$$

Case: *(!SPEC)*

$$T; \circledcirc i; E \rhd j : s \;\longrightarrow\; T; \triangledown i; E \rhd j : s$$

As one would expect, *(!SPEC)* transitions can decrease the expected work set:

$$\begin{aligned} LHS &= D \;\cup\; \mathcal{W}\{\!\{\{i,j\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} && \text{defn of } pendingWork \\ &\supseteq D \;\cup\; \mathcal{W}\{\!\{\{j\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} && \text{defn of } \mathcal{W}\{\!\{-\}\!\} \\ &= D' \;\cup\; P' \;=\; RHS && \text{defn of } workDone \text{ and } pendingWork \end{aligned}$$

Case: *(!ABORT)*

$$\mathcal{M}[\![T; \circledcirc i; (f,j) : s]\!] \;\longrightarrow\; T[j \mapsto i \angle f]; \circledcirc j; s$$

$$\begin{aligned} LHS &= D \;\cup\; \mathcal{W}\{\!\{\{i\} \cup \mathcal{S}\{\!\{(f,j) : s\}\!\}\}\!\}^{\Psi}_{\gamma} && \text{defn of } pendingWork \\ &= D \;\cup\; \mathcal{W}\{\!\{\{i,j\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} && \text{defn of } \mathcal{S}\{\!\{-\}\!\} \text{ and } \mathcal{W}\{\!\{-\}\!\} \\ &\supseteq D \;\cup\; \{i\} \;\cup\; \mathcal{W}\{\!\{\{j\} \cup \mathcal{S}\{\!\{s\}\!\}\}\!\}^{\Psi}_{\gamma} && \text{defn of } \mathcal{W}\{\!\{-\}\!\} \\ &= D' \;\cup\; P' \;=\; RHS && \text{defn of } workDone \text{ and } pendingWork \end{aligned}$$

Case: *(!RESUME)*

This is simply the inverse of the proof for *(!ABORT)*.

APPENDIX D

Example HsDebug Debugging Logs

In this appendix, we present some example debugging sessions using HsDebug. All the text shown is real output from HsDebug. The program being debugged in each case is the following:

```
 1: module Main (main,last') where
 2:
 3: import System ( getArgs )
 4:
 5: main :: IO ()
 6: main = do
 7:    [countstr] <- getArgs
 8:    let count = read countstr
 9:    let xs = [4*2, 5 `div` 0] ++ replicate count 42
10:    print (head xs, last' xs)
11:
12: last' (x:xs) = last' xs
13: last' [x] = x
```

This program is based on an example given in the online documentation for Hat [HAT]. We have altered the program slightly so as to make its behaviour dependent on program input, thus preventing GHC from optimising the entire program away.

All the examples shown are debugging a binary that has been compiled with the `-O2` flag to GHC, turning on all optimisations.

In all the examples, one can observe the following shortcomings in the current implementation of HsDebug:

- Source locations are sometimes a little wrong.
- Some function and closure names are messy ones made up by the compiler.

We intend to fix all of these problems, but we have not done so yet.

The log below shows how HsDebug can be used to find out where an exception is occurring:

```
bash-2.03$ hsdebug paperdemo2 3
(hsdebug) continue
Exception raised:
data: GHC.IOBase.PatternMatchFail.con <0>
        ["paperdemo." ++ 0x402c7e94]
(hsdebug) where
locals = ()
        args = (0x8099a43)
#0: Main.lvl3                   at paperdemo.hs:12
        update : 402c74c8
        args = ()
#1: Main.last'                  at paperdemo.hs:12
        args = ([])
#2: Main.last'                  at paperdemo.hs:12
        args = ([S# 42])
#3: Main.last'                  at paperdemo.hs:12
        args = ([S# 42, S# 42])
#4: Main.last'                  at paperdemo.hs:12
        args = ([S# 42, S# 42, S# 42])
#5: Main.last'                  at paperdemo.hs:12
        args = ([GHC.Real.lvl16.closure, S# 42, S# 42, S# 42])
#6: Main.last'                  at paperdemo.hs:12
        args = ([Main.a.closure, GHC.Real.lvl16.closure, S# 42, S# 42, S# 42])
#7: P4lu                        at inlined "print"
#8: xs.s4kR                     at paperdemo.hs:9
        env = ([Main.a.closure, GHC.Real.lvl16.closure, S# 42, S# 42, S# 42])
#9: Main.main                   at inlined "print"
        env = ("3")
```

```
        catch frame : GHC.TopHandler.topHandler.info
        startup code
end of stack
(hsdebug)
```

The next log shows how breakpoints can be used to observe the execution of a program. In this case we are placing a breakpoint on the entry to the `last'` function:

```
bash-2.03$ hsdebug paperdemo 3
(hsdebug) b Main.last'
Breakpoint set at address 0x804929c
(hsdebug) c
breakpoint hit: Main.last' (0x804929c)
args = ([Main.a.closure, GHC.Real.lvl16.closure, S# 42,
         S# 42, S# 42])
(hsdebug) c
breakpoint hit: Main.last' (0x804929c)
args = ([GHC.Real.lvl16.closure, S# 42, S# 42, S# 42])
(hsdebug) c
breakpoint hit: Main.last' (0x804929c)
args = ([S# 42, S# 42, S# 42])
(hsdebug) c
breakpoint hit: Main.last' (0x804929c)
args = ([S# 42, S# 42])
(hsdebug) c
breakpoint hit: Main.last' (0x804929c)
args = ([S# 42])
(hsdebug) c
breakpoint hit: Main.last' (0x804929c)
args = ([])
(hsdebug) c
Exception raised:
data: GHC.IOBase.PatternMatchFail.con <0>
        ["paperdemo." ++ 0x402c7e94]
(hsdebug)
```

In these logs, we can note the following:

- The reference to `demo.hs:9` should really be to `demo.hs:10`. This is a shortcoming of our current source location information, which gives everything in the body of a `let` a location corresponding to the start of the `let`.
- `4*2` and `5 ‘div‘ 0` have been hoisted up into top level closures by the compiler. The compiler has called them `a` and `lvl16`. Such closures are not currently speculated by Optimistic Evaluation.
- `lvl33`, `P4lu` and `xs.s4kR` are code blocks made up by the compiler. We intend to hide these cryptic names in the future.
- The `args` line tells us the arguments that the function was called with. We can see that the list of each call is the tail of the list of the previous call.
- Structures such as lists and integers are pretty printed automatically.
- When `print` is inlined, its arguments get mixed up with the body of `print` and so are given the srcloc `inlined "print"`. This is rather counter-intuitive, and we intend to change this.

HsDebug has many features not covered in these logs. In particular, it can pretty print closures in the heap and has special features to assist in debugging the GHC runtime system.

Bibliography

[ABD+97] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone. Technical report, Digitial Equipment Corporation Systems Research Center, July 1997.

[Abr90] Samson Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.

[AC97] Andrea Asperti and Juliusz Chroboczek. Safe operators: Brackets closed forever — optimizing optimal lambda-calculus implementations. *Applicable Algebra in Engineering, Communication and Computing*, 8(6):437–468, 1997.

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, White Plains, NY, June 1990.

[AJ89] Lennart Augustsson and Thomas Johnsson. The chalmers lazy-ML compiler. *The Computer Journal*, 32(2):127–141, April 1989.

[AJ94] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[AR01] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[Arn02] Matthew Arnold. *Online Profiling and Feedback-Directed Optimization of Java*. PhD thesis, Rutgers, The State University of New Jersey, October 2002.

[Bak95] Henry G. Baker. 'use-once' variables and linear objects — storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, pages 45–52, January 1995.

[BCF+99] Michael G. Burke, Jong-Doek Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, Vugranam C. Sreedhar, Harinin Srinivasan, and Jahn Whaley. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings of the ACM Java Grande Conference*, 1999.

[BD96] Stephen Brookes and Denis Dancanet. Circuit semantics and intensional expressivity. http://www-2.cs.cmu.edu/ ddr/, August 1996.

[BF02] Anasue Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 2002.

[BH77] Henry G. Baker and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the Symposium on AI and Programming Languages*, 1977.

[BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness Analysis for Higher-Order Functions. *Science of Computer Programming*, 7:249–278, 1986.

[BL93] Thomas Ball and James R. Larus. Branch prediction for free. In PLDI93 [PLD93], pages 300–313.

[Boq99] Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Sweden, April 1999.

[Bur85] F Warren Burton. Speculative computation, parallelism and functional programming. *IEEE Transactions on Computers*, C-34(12):1190–1193, December 1985.

[BvEvLP87] TH Brus, MCJD van Eckelen, MO van Leer, and MJ Plasmeijer. Clean — a language for functional graph rewriting. In G Kahn, editor, *Proceedings of the 1997 Conference on Functional programming languages and computer architecture*, pages 364–384. LNCS 274, Springer Verlag, September 1987.

[CD93] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In POPL93 [POP93].

[CG99] Fay Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, February 1999.

[Cha98] Manuel M. T. Chakravarty. Lazy thread and task creation in parallel graph reduction. In *International Workshop of Implementing Functional Languages*, Lecture Notes in Computer Science. Springer Verlag, 1998.

[Dan98] Denis R. Dancanet. *Intensional Investigations*. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1998.

[Eme99] Joel S. Emer. Simultaneous multithreading: Multiplying alpha's performance. In *Microprocessor Forum*, 1999.

[EP03a] Robert Ennals and Simon Peyton Jones. HsDebug : Debugging lazy programs by not being lazy (tool demo). In *Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell'03)*, August 2003.

[EP03b] Robert Ennals and Simon Peyton Jones. Optimistic Evaluation: An adaptive evaluation strategy for non-strict programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, Uppsala, August 2003. ACM.

[EPM03] Robert Ennals, Simon Peyton Jones, and Alan Mycroft. A cost model for non-strict evaluation. *Unpublished; Likely to be submitted somewhere soon*, 2003.

[ET96] Hartmut Ehrig and Gabriele Taentzer. Computing by graph transformation: A survey and annotated bibliography. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 59, 1996.

[Fax00] Karl-Filip Faxén. Cheap Eagerness: Speculative evaluation in a lazy functional language. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-00)*, 2000.

[Fax01] Karl-Filip Faxén. Dynamic Cheap Eagerness. In *Proceedings of the 2001 Workshop on Implementing Functional Languages*. Springer Verlag, 2001.

[FC03] Keir Fraser and Fay Chang. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[FDD01] *ACM Workshop on Feedback Directed and Dynamic Optimization (FDDO)*, 2001.

[FF92] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, in *SIGPLAN Notices*, 1992.

[FF95] Corman Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *Proceedings on the 22nd ACM Symposium on Principles of Programming Languages*, pages 209–220, 1995.

[FPC93] *ACM Conference on Functional Programming and Computer Architecture (FPCA'93)*, Cophenhagen, 1993. ACM.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce Duba, and Matthias Felleisen. The essence of compiling with continuations. In PLDI93 [PLD93], pages 237–247.

[GB96] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full-speculation. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 309–321, 1996.

[GB99] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full-speculation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21:240–285, 1999.

[Gil00] Andy Gill. Debugging haskell by observing intermediate data structures. In *Proceedings of the ACM SIGPLAN Haskell Worshop*, 2000.

[GKM83] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software Practice and Experience*, 3:671–685, August 1983.

[GLP93] Andy Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In FPCA93 [FPC93], pages 223–232.

[GM96] James Gosling and Henry McGilton. The Java Language Environment: a White Paper. Technical report, Sun Microsystems, 1996.

[GP81] Dale H. Grit and Rex L. Page. Deleting irrelevent tasks in an expression oriented multiprocessor system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3:49–59, January 1981.

[Gus98] Jorgen Gustavsson. A type-based sharing analysis for update avoidance and optimisation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, volume 34(1) of *ACM SIGPLAN Notices*, Baltimore, 1998. ACM.

[Gwe99] Linley Gwennap. MAJC gives VLIW a new twist. *Microprocessor Report*, 13(12), 1999.

[Ḧ95] Urs Hölzle. *Adaptive optimization for Self: reconciling high performance with exploratory programming*. Ph.D. thesis, Computer Science Department, Stanford University, March 1995.

[Hal85] Robert H Halstead. Multilisp - a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HAT] Hat - the haskell tracer. `http://haskell.org/hat/`.

[HK82] Paul Hudak and Robert M. Keller. Garbage collection and task deletion in distributed systems. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 168–178, August 1982.

[HO93] Kevin Hammond and John T. O'Donnell, editors. *Functional Programming, Glasgow 1993*, Workshops in Computing. Springer Verlag, 1993.

[Hug89] R. John M. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[ia600] Intel. *IA-64 Achitecture Software Developer's Manual*, 2000.

[int97] Intel. *Using the RDTSC instruction for Performance Monitoring*, 1997.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[Joh85] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA'85)*, volume 201 of *Lecture Notes in Computer Science*, Nancy, France, September 1985. Springer-Verlag.

[Jon92] Richard E. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, September 1998.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[Kri99] Venkata S. Krishnan. *Speculative Multithreading Architectures*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-CHampaign, 1999.

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, 1990.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In POPL93 [POP93].

[LCH+03] Jin Lin, Tong Chen, Wei-Chung Hsu, Roy Dz-Ching Ju, Ton-Fook Ngai, Pen-Chung Yew, and Sun Chan. A compiler framework for speculative analysis and optimizations. In *ACM Conference on Programming Languages Design and Implementation (PLDI'03)*, San Diego, California, June 2003. ACM.

[Lév78] Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université de Paris, 1978.

[Lév80] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 1980.

[LRVD98] Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. *The Objective Caml system, documentation and user's guide*. INRIA, 1998. Available at `http://pauillac.inria.fr/ocaml/htmlman/`.

[LS92] John Launchbury and Patrick Sansom, editors. *Functional Programming, Glasgow 1992*, Workshops in Computing. Springer Verlag, 1992.

[Mac92] Robert A. MacLachlan (ed). CMU Common Lisp users manual. Technical report, School of Computer Science, Carnegie Mellon University, 1992.

[Mae02a] Jan-Willem Maessen. Eager Haskell: Resource-bounded execution yields efficient iteration. In *The Haskell Workshop, Pittsburgh*, 2002.

[Mae02b] Jan-Willem Maessen. *Hybrid Eager and Lazy Evaluation for Efficient Compilation of Haskell*. PhD thesis, Massachusetts Institute of Technology, June 2002.

[Mai03] Harry Mairson. From Hilbert spaces to Dilbert spaces: context semantics made simple. In *Proceedings of the 23rd International Conference on Foundations of Software Technology and Theoretical Computer Science*, 2003.

[Mar02] Simon Marlow. State monads don't respect the monad laws in haskell. Message posted to the Haskell mailing list: http://haskell.org/pipermail/haskell/2002-May/009622.html, May 2002.

[Mat93] James S. Mattson. *An effective speculative evaluation technique for parallel supercombinator graph reduction*. Ph.D. thesis, Department of Computer Science and Engineering, University of California, San Diego, February 1993.

[Mau02] Luke Maurer. Isn't this tail recursive? Message posted to the Haskell mailing list: http://haskell.org/ pipermail/haskell/2002-March/009126.html, March 2002.

[MG99] Pedro Maruello and Antonio Gonzalez. Clustered speculative multithreaded processors. In *Proceedings of the International Conference on Supercomputing*, 1999.

[MGA93] Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 248–259, 1993.

[MJ98] Richard G. Morgan and Stephen A. Jarvis. Profiling large-scale lazy functional programs. *Journal of Functional Programming*, 8(3), May 1998.

[MJG93] Jim S. Mattson Jr and William G. Griswold. Local Speculative Evaluation for Distributed Graph Reduction. In Hammond and O'Donnell [HO93], pages 185–192.

[MLP99] Andy Moran, Soeren Lassen, and Simon Peyton Jones. Imprecise exceptions, coinductively. In *Proceedings of the 3rd International Workshop on Higher Order Operational Techniques in Semantics*, number 26 in Electronic Notes in Theoretical Computer Science, pages 137–156. Elsevier, 1999.

[MN92] Alan Mycroft and Arthur Norman. Optimising compilation — lazy functional languages. In *Proceedings of the 19th Software Seminar (SOFSEM)*, 1992.

[MP98] Simon Marlow and Simon Peyton Jones. The new ghc/hugs runtime system. Technical report, University of Glasgow, August 1998. From www.haskell.org.

[MP03] Simon Marlow and Simon Peyton Jones. Making a fast curry: Push/enter vs eval/apply for higher-order languages. Available at http://research.microsoft.com/users/simonpj/, 2003.

[MPMR01] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI'01)*, pages 274–285, Snowbird, Utah, June 2001. ACM.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming*, 1980.

[Myc81] Alan Mycroft. *Abstract Interpretation and optimising transformations of applicative programs*. PhD thesis, Edinburgh University, 1981.

[NA01] Rishiyur S Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufman, 2001.

[NF92] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. In Maurice Bruynooghe and Martin Wirsing, editors, *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming (PLILP '92)*, volume 631 of *Lecture Notes in Computer Science*, pages 385–399, Leuven, Belgium, August 1992.

[NF94] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.

[Nik91] Rishiyur Sivaswami Nikhil. Id (version 90.1) language reference manual. Technical Report CSG Memo 284-2, MIT Computation Structures Group, 1991.

[Nil01] Henrik Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, November 2001.

[NS96] Henrik Nilsson and Jan Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Technical report, Department of Computer and Information Science, Linkoping University, 1996.

[NS97] Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering: An International Journal*, 4(2):121–150, April 1997.

[NSvEP91] Eric Nocker, Sjaak Smetsers, Marko van Eekelen, and Rinus Plasmeijer. Concurrent clean. In Leeuwen Aarts and Rem, editors, *Proceedings of the International Symposium on Parallel Architectures and Languages Europe (PARLE '91)*, volume 505, pages 202–219. Springer-Verlag, 1991.

[Ong88] Chih-Hao Luke Ong. *The Lazy Lambda Calculus: An Investigation in the Foundations of Functional Programming*. PhD thesis, Imperial College, London, 1988.

[Osb89] Randy B. Osbourne. *Speculative computation in Multilisp*. PhD thesis, MIT Lab for Computer Science, December 1989.

[Par91] Andrew S. Partridge. *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages*. PhD thesis, University of Tasmania, October 1991.

[Par92] Will D. Partain. The `nofib` benchmark suite of Haskell programs. In Launchbury and Sansom [LS92], pages 195–202.

[PD89] Andrew S. Partridge and Anthony H. Dekker. Speculative parallelism in a distributed graph reduction machine. In *Proceedings of the Hawaii International Conference on System Sciences*, 1989.

[PEP97] *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, volume 32 of *SIGPLAN Notices*, Amsterdam, June 1997. ACM.

[Pey91] Simon Peyton Jones. The spineless tagless G-machine: a second attempt. Technical report, Department of Computing Science, University of Glasgow, February 1991.

[Pey92] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[Pey01] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In CAR Hoare, M Broy, and R Steinbrueggen, editors, *Engineering theories of software construction, Marktoberdorf Summer School 2000*, NATO ASI Series, pages 47–96. IOS Press, 2001.

[Pey03] SL Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, April 2003.

[PHH+93] Simon Peyton Jones, Cordelia Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell Compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, pages 249–257. DTI/SERC, March 1993.

[PL91] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, Boston, 1991. Springer Verlag.

[PL92] Simon Peyton Jones and David Lester. *Implementing functional languages: a tutorial*. Prentice Hall, 1992.

[PLD93] *ACM Conference on Programming Languages Design and Implementation (PLDI'93)*. ACM, June 1993.

[PMR99] Simon Peyton Jones, Simon Marlow, and Alastair Reid. The stg runtime system (revised). Technical report, Microsoft Research, February 1999. Part of the GHC source package.

[PO03] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the 9th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2003.

[POP93] *20th ACM Symposium on Principles of Programming Languages (POPL'93)*. ACM, January 1993.

[Pop98] Bernard Pope. Buddha: A declarative debugger for haskell. Technical report, Department of Computer Science, University of Melbourne, Australia, June 1998.

[PP93] Simon Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. In Hammond and O'Donnell [HO93], pages 201–220.

[PRH+99] Simon Peyton Jones, Alastair Reid, CAR Hoare, Simon Marlow, and Fergus Henderson. A semantics for imprecise exceptions. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI'99)*, pages 25–36, Atlanta, May 1999. ACM.

[PW93] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In POPL93 [POP93], pages 71–84.

[Rei01] Claus Reinke. GHood — graphical visualisation and animation of haskell object observations. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, 2001.

[Roe90] Paul Roe. Calculating lenient programs' performance. In *Proceedings of the Glasgow Worshop on Functional Programming*, 1990.

[Ros89] Mads Rosendahl. Automatic complexity analysis. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, September 1989.

[RPHL02] Álvaro J. Rebón Portillo, Kevin Hammand, and Hans-Wolfgang Loidl. Cost analysis using automatic size and time inference. In *Proceedings of the Workshop on Implementing Functional Languages*. Springer Verlag, 2002.

[RR96a] Niklas Röjemo and Colin Runciman. Lag, drag, void, and use: heap profiling and space-efficient compilation revisited. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 34–41. ACM, Philadelphia, May 1996.

[RR96b] Colin Runciman and Niklas Röjemo. New dimensions in heap profiling. *Journal of Functional Programming*, 6(4), September 1996.

[RV04] *4th Workshop on Runtime Verification*, Barcelona, Spain, 2004.

[RW92] Colin Runciman and David Wakeling. Heap profiling a lazy functional compiler. In Launchbury and Sansom [LS92], pages 203–214.

[San94] Patrick M. Sansom. *Execution profiling for non-strict functional languages*. Ph.D. thesis, University of Glasgow, September 1994.

[San95a] David Sands. A naive time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4), 1995.

[San95b] André Santos. *Compilation by transformation in non-strict functional languages*. Ph.D. thesis, Department of Computing Science, Glasgow University, September 1995.

[SCG95] Klaus E. Schauser, David E. Culler, and Seth C Goldstein. Separation constraint partitioning: a new algorithm for partitioning non-strict programs into sequential threads. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 259–271. ACM, January 1995.

[Ses97] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3), 1997.

[Smi81] James E. Smith. A study of branch prediction strategies. In *Proceedings of the International Symposium on Computer Architecture*, 1981.

[SP91] Richard M. Stallman and Roland H. Pesch. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, 1991.

[SP93] Patrick M. Sansom and Simon Peyton Jones. Generational garbage collection for haskell. In FPCA93 [FPC93], pages 106–116.

[Spa93] Jan Sparud. Fixing some space leaks without a garbage collector. In FPCA93 [FPC93], pages 117–124.

[SR97] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *PLILP*, pages 291–308, 1997.

[SR98] Jan Sparud and Colin Runciman. Complete and partial redex trails of functional computations. *Lecture Notes in Computer Science*, 1467:160–??, 1998.

[Sun01] Sun Microsystems. *The Java HotSpot Virtual machine, White Paper*, 2001.

[TA90] Andrew P. Tolmach and Andrew W. Appel. Debugging standard ML without reverse engineering. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 1–12, New York, NY, 1990. ACM.

[TA95] Andrew P. Tolmach and Andrew W. Appel. A debugger for standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[TEL95] Dean Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[Tra88] Ken Traub. *Sequential implementation of lenient programming languages*. Ph.D. thesis, MIT Lab for Computer Science, 1988.

[TtGT01] Andrew Tolmach and the GHC Team. An external representation for the ghc core language. Available at http://haskell.org, 2001.

[TWM95] David N Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture (FPCA'95)*, La Jolla, California, 1995. ACM.

[vD89] Christina von Dorrien. Stingy evaluation. Licentiate thesis, Chalmers University of Technology, May 1989.

[Voi02] Janis Voiglaender. ‘seq‘ breaks the foldr/buildr-rule. Message posted to the Haskell mailing list: http://haskell.org/pipermail/haskell/2002-May/009653.html, May 2002.

[Wad71] Christopher Wadsworth. *Semantics and Pragmatics of the lambda calculus*. PhD thesis, University of Oxford, 1971.

[Wad84] Philip Wadler. Listlessness is better than laziness. Technical report, Programming research group, Oxford University, January 1984.

[Wad87] Philip Wadler. Fixing a space leak with a garbage collector. *Software - Practice and Experience*, 17(9):595–608, 1987.

[Wad88] Philip Wadler. Strictness analysis aids time analysis. In *Proceedings of the ACM Sympossium on Principles of Programming Languages*, 1988.

[Wad90a] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[Wad90b] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Me thods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[Wad95] Philip Wadler. Monads for functional programming. In J Jeuring and E Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[Wad97] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3), 1997.

[Wad98] Philip Wadler. Why no one uses functional languages. *SIGPLAN Notices*, August 1998.

[Wan02] Keith Wansbrough. *Simple Polymorphic Usage Analysis*. PhD thesis, Computer Laboratory, University of Cambridge, March 2002.

[WCBR01] Malcom Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for haskell: a new hat. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, 2001.

[WH87] Philip Wadler and John Hughes. Projections for strictness analysis. In G Kahn, editor, *Proceedings of the ACM SIGPLAN Conference on Functional Programming Languages and Computer Architecture*. Springer Verlag LNCS 274, September 1987.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, Saint-Malo (France), 1992. Springer-Verlag.

[WP99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *26th ACM Symposium on Principles of Programming Languages (POPL'99)*, San Antonio, January 1999. ACM.

[ZS01] Craig B. Zilles and Gurindar S. Sohi. A programmable co-processor for profiling. In *Proceedings of the International Symosium on High Performance Computer Architecture*, January 2001.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W