Hybrid Eager and Lazy Evaluation for Efficient Compilation of Haskell

by

Jan-Willem Maessen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Author . . . . . .
Department of Electrical Engineering and Computer Science
17 May 2002

Certified by . . . . . .
Arvind
Charles and Jennifer Johnson Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Hybrid Eager and Lazy Evaluation for Efficient Compilation of Haskell

by

Jan-Willem Maessen

Submitted to the Department of Electrical Engineering and Computer Science
on 17 May 2002, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

The advantage of a non-strict, purely functional language such as Haskell lies in its clean equational semantics. However, lazy implementations of Haskell fall short: they cannot express tail recursion gracefully without annotation. We describe *resource-bounded hybrid evaluation*, a mixture of strict and lazy evaluation, and its realization in *Eager Haskell*. From the programmer's perspective, Eager Haskell is simply another implementation of Haskell with the same clean equational semantics. Iteration can be expressed using tail recursion, without the need to resort to program annotations. Under hybrid evaluation, computations are ordinarily executed in program order just as in a strict functional language. When particular stack, heap, or time bounds are exceeded, suspensions are generated for all outstanding computations. These suspensions are re-started in a demand-driven fashion from the root.

The Eager Haskell compiler translates λ_C, the compiler's intermediate representation, to efficient C code. We use an equational semantics for λ_C to develop simple correctness proofs for program transformations, and connect actions in the run-time system to steps in the hybrid evaluation strategy. The focus of compilation is efficiency in the common case of straight-line execution; the handling of non-strictness and suspension are left to the run-time system.

Several additional contributions have resulted from the implementation of hybrid evaluation. Eager Haskell is the first eager compiler to use a call stack. Our generational garbage collector uses this stack as an additional predictor of object lifetime. Objects above a stack watermark are assumed to be likely to die; we avoid promoting them. Those below are likely to remain untouched and therefore are good candidates for promotion. To avoid eagerly evaluating error checks, they are compiled into special *bottom thunks*, which are treated specially by the run-time system. The compiler identifies error handling code using a mixture of strictness and type information. This information is also used to avoid inlining error handlers, and to enable aggressive program transformation in the presence of error handling.

Thesis Supervisor: Arvind
Title: Charles and Jennifer Johnson Professor of Computer Science and Engineering

Acknowledgments

I would like to to those who have helped me along through the long haul. Most especial gratitude goes to Andrea Humez, who has supported me as friend and confidante over eight and a half years. On the technical I have been influenced by so many people an exhaustive list is out of the question. I would like to single out my *pH* co-conspirators, especially Alejandro Caro, Mieszko Lis, and Jacob Schwartz. I learned a tremendous amount in my collaborations with Lennart Augustsson, Rishiyur S. Nikhil, Joe Stoy, and Xiaowei Shen. Finally, Arvind's support and interest has made the *pH* and Eager Haskell projects possible and kept things on the straight and narrow.

Contents

1 Introduction **18**

1.1 Functional languages 18

1.1.1 Strict languages 19

1.1.2 Non-strict languages 20

1.1.3 The importance of purity 21

1.2 Evaluation Strategies 23

1.2.1 Lazy Evaluation 23

1.2.2 Eager Evaluation 24

1.2.3 Multithreaded strategies for parallelism 25

1.3 The advantages of eagerness over laziness 26

1.4 Contributions 28

1.5 Overview of this thesis 30

2 Representing Eager Programs: The λ_C Calculus **32**

2.1 Overview 32

2.2 Notation 33

2.3 Functions 34

2.4 Application 34

2.5 Blocks 35

2.6 Primitives 36

2.7 Algebraic Data Types 36

2.8 Case Expressions 37

2.9 Other syntax 39

3 The Semantics of λ_C 40

3.1 Extensionality . . . 40

3.2 Equivalence . . . 41

3.3 Conversion . . . 43

3.3.1 Functions . . . 43

3.3.2 Primitives . . . 44

3.3.3 Algebraic types . . . 44

3.3.4 Binding . . . 46

3.3.5 Structural rules . . . 47

3.4 λ_C is badly behaved . . . 49

3.5 Canonical forms of λ_C . . . 49

3.5.1 Full erasure . . . 49

3.5.2 Fully named form . . . 50

3.5.3 Binding contexts . . . 52

3.5.4 Named form . . . 52

3.5.5 Argument-named form . . . 53

3.5.6 Flattened form . . . 53

3.6 Reduction of λ_C . . . 53

4 Evaluation strategies for λ_C 56

4.1 Overview . . . 56

4.2 Evaluation mechanisms . . . 57

4.2.1 Term structure . . . 57

4.2.2 Starting the program . . . 58

4.2.3 Evaluation context . . . 58

4.2.4 Function calls: manipulating the stack . . . 59

4.2.5 Results . . . 59

4.2.6 Deadlock . . . 60

4.2.7 Storing and fetching values . . . 60

4.2.8 Placing non-values on the heap . . . 60

4.2.9 Placing computations on the heap . . . 61

4.2.10 Garbage collection . . . 62

4.3 Reduction strategies . . . 62
4.3.1 A lazy strategy . . . 62
4.3.2 A strict strategy . . . 63
4.4 Eagerness . . . 64
4.4.1 A fully eager strategy . . . 65
4.4.2 The hybrid strategy . . . 66
4.5 How strategies treat the heap . . . 67
4.6 Other Eager Strategies . . . 68
4.7 Resource-bounded Computation . . . 69

5 Run-time Structure **71**
5.1 Overview . . . 71
5.2 Driving assumptions . . . 72
5.2.1 Architectures reward locality . . . 72
5.2.2 Branches should be predictable . . . 72
5.2.3 Compiling to C will produce better code . . . 73
5.2.4 Non-strictness is rare . . . 73
5.2.5 Values are common . . . 74
5.3 Tagged data . . . 74
5.4 Function structure . . . 77
5.5 Currying . . . 79
5.5.1 The eval-apply approach . . . 80
5.5.2 The push-enter approach . . . 82
5.5.3 Analysis . . . 83
5.6 Suspensions . . . 84
5.7 Thunks . . . 87
5.8 Indirections . . . 88
5.9 Garbage Collection . . . 89
5.9.1 Multiprocessor collection constrains our design . . . 89
5.9.2 Write barrier . . . 90
5.9.3 Nursery management . . . 91
5.9.4 Fallback policy . . . 92

5.9.5 Promotion policy . . . 93
5.9.6 Tenured space management . . . 94
5.9.7 Problems with the tenured collector . . . 95
5.9.8 Towards better storage management . . . 96

6 Lowering Transformations 97

6.1 Optimizations . . . 98
6.2 Constant Hoisting . . . 99
6.3 Lambda lifting . . . 101
6.4 Splitting huge expressions . . . 103
6.5 Top-level common subexpression elimination . . . 104
6.6 Constant applicative forms . . . 105
6.7 Pseudo-constructors . . . 106
6.8 Back edge insertion . . . 106
6.9 Making synchronization explicit . . . 108
6.9.1 Introducing synchronization . . . 109
6.9.2 Eliminating excess synchronization . . . 109
6.10 Eliminating additional synchronization . . . 111
6.10.1 Hoisting to eliminate redundant synchronization . . . 111
6.10.2 Using Transitivity . . . 112
6.10.3 Partitioning versus explicit synchronization . . . 113
6.10.4 Interprocedural synchronization elimination . . . 114
6.11 Canonical lowered λ_C . . . 117

7 Eager Code Generation 118

7.1 Save points . . . 118
7.2 Frame structure . . . 121
7.3 Allocation . . . 122
7.4 Functions . . . 123
7.5 Constructors . . . 125
7.6 Function application . . . 126
7.7 Primitive expressions . . . 128
7.8 Case expressions . . . 129

7.9 Suspensive bindings . . . 131

8 Bottom Lifting: Handling Exceptional Behavior Eagerly 133

8.1 Semantics of divergence . . . 135

8.1.1 The meaning of a divergent expression . . . 136

8.2 Evaluation strategies for divergence . . . 137

8.3 Identifying bottom expressions . . . 138

8.3.1 Strictness information . . . 138

8.3.2 Type information . . . 138

8.3.3 Compiler assistance . . . 139

8.4 Enlarging the lifted region . . . 139

8.5 Lifting divergent terms . . . 140

8.6 Divergent computation at run time . . . 141

8.7 Related work . . . 141

9 Implementing Lazy Arrays 143

9.1 Signal Pools . . . 145

9.2 Using *seq* . . . 146

9.3 Fairness using *lastExp* . . . 147

10 Results 149

10.1 The benchmarks . . . 150

10.1.1 Fib . . . 151

10.1.2 Clausify . . . 151

10.1.3 fibheaps . . . 151

10.1.4 Queens . . . 151

10.1.5 Paraffins . . . 151

10.1.6 Primes . . . 152

10.1.7 Multiplier . . . 152

10.1.8 Wavefront . . . 153

10.1.9 Matrix Multiply . . . 153

10.1.10 Gamteb . . . 153

10.1.11 Symalg . . . 154

10.1.12 Anna . . . 154
10.2 Eager Haskell versus GHC . . . 154
10.3 Garbage collection . . . 158
10.4 Function Application . . . 163
10.5 Fallback . . . 166
10.6 Suspension . . . 167
10.7 Forcing variables . . . 169
10.8 Space-efficient recursion: the multiplier benchmark . . . 172

11 Scheduling Eager Haskell on a Multiprocessor 178
11.1 Indolent task creation . . . 179
11.2 Scheduling Strategy . . . 180
11.3 Scheduling in the presence of useless computation . . . 181
11.4 Memory Structure: The Principle of Monotonicity . . . 182

12 Compiling *pH* Programs Using the Eager Haskell Compiler 186
12.1 What is a barrier? . . . 186
12.2 Barriers in the *pH* compiler . . . 187
12.2.1 A lazier barrier . . . 187
12.2.2 Reducing state saving . . . 188
12.3 Barriers in Eager Haskell . . . 189
12.3.1 Tracking the work . . . 189
12.3.2 Run-time system changes . . . 190
12.3.3 Compiler changes . . . 190
12.3.4 A synchronization library . . . 191

13 Conclusion 193
13.1 Semantics . . . 193
13.2 Eagerness . . . 194
13.2.1 Fast stack unwinding . . . 194
13.2.2 Hybrid evaluation without fallback . . . 195
13.3 Improving the quality of generated code . . . 196
13.3.1 Garbage Collection . . . 196

13.3.2 Reducing synchronization . . . 197
13.3.3 Better representations for empty objects . . . 199
13.3.4 Object Tagging . . . 200
13.3.5 Unboxing . . . 200
13.4 Other compiler improvements . . . 202
13.4.1 Specialization . . . 202
13.4.2 Control-flow analysis . . . 203
13.4.3 Loop optimization . . . 203
13.4.4 Improving inlining . . . 204
13.4.5 Range-based optimizations . . . 205
13.4.6 Constructed products . . . 205
13.4.7 Better deforestation . . . 205
13.4.8 Escape analysis . . . 206
13.5 Envoi . . . 206

A The Defer List Strategy for λ_C **221**

List of Figures

1-1 Simple expression parser written using parsing combinators 22

2-1 Syntax of λ_C 33

3-1 Syntactic equivalences for terms in λ_C 41
3-2 Conversion in λ_C 42
3-3 Instantiation contexts in λ_C 46
3-4 Derivations for σ_m and τ_m 47
3-5 Strict contexts in λ_C 48
3-6 Restricted ν rules for full naming 50
3-7 Fully named form of λ_C 50
3-8 Order of floating matters 51
3-9 Binding contexts in λ_C 52
3-10 Argument-named form of λ_C 53
3-11 Argument-named λ_C during reduction 54
3-12 General dynamic reduction rules for λ_C 55

4-1 Structure of terms during evaluation 57
4-2 Reduction rules used by every strategy 58
4-3 Reduction rules for lazy strategy 63
4-4 Additional reduction rule for strict strategy 64
4-5 A fully eager strategy 65
4-6 Hybrid eager and lazy strategy 66
4-7 Reduction in the presence of exceptions 69

5-1 Boxed representation of numbers 75

5-2 Partial application of a simple closure. 81
5-3 Applying a partial application 83
5-4 Suspension structure 84
5-5 Updating transitive dependency fields. 86
5-6 Elision and shortcutting of indirections. 88

6-1 Correctness of full laziness 101
6-2 Reverse instantiation is CSE 104
6-3 Synchronization elimination for transitive dependencies 113
6-4 Worker/wrapper requires additional synchronization 115
6-5 Fully synchronized, lowered λ_C 116

7-1 Skeleton code for *fib* 124
7-2 Code for constructors 125
7-3 Three cases of function application 127
7-4 Code for primitive expressions 128
7-5 Code for **case** expressions 130
7-6 Spawn code 132

8-1 Semantics of divergence 135
8-2 Hybrid reduction with divergence 137
8-3 Hoisting divergence from a multi-disjunct case. 140

9-1 The `wavefront` benchmark 143
9-2 Implementing arrays using *SignalPool*s 145
9-3 Implementing *SignalPool*s 147

10-1 Run times of benchmarks 156
10-2 Slowdown of Eager Haskell compared to GHC. 157
10-3 Percentage of total run time spent in garbage collector. 159
10-4 Number of write barrier checks, normalized to mutator time. 161
10-5 Actual number of write barriers triggered, normalized to mutator time. 161
10-6 Barrier indirections per write barrier 162
10-7 Function entries, normalized to mutator time. 164

10-8 Entries to *GeneralApply*, as a percentage of all applications 164
10-9 Fallbacks per second 165
10-10 The consequences of fallback 166
10-11 Touch operations normalized to mutator time 167
10-12 Percentage of touches which force 168
10-13 Function entries upon resumption 168
10-14 Percentage of indirections which are followed 169
10-15 Variables forced, normalized to time 170
10-16 Variables forced, proportionally 170
10-17 Original multiplier code 172
10-18 Inlined multiplier code 173
10-19 Run times of different versions of multiplier 174
10-20 Slowdown of Eager Haskell compared to GHC on multiplier. 174
10-21 Speedup of re-annotated multiplier 176

11-1 Monotonic update of objects 183

12-1 Types and functions for representing barriers 190
12-2 Barrier translation 191

A-1 Eagerness using defer lists and work stealing 222

List of Tables

1.1 A taxonomy of languages, semantics, and strategies. 20

5.1 Different cases of curried function application and their presumed frequency 79
5.2 The eval-apply approach to partial application used in Eager Haskell 81
5.3 The push-enter approach to partial application used in GHC. 82

10.1 Benchmarks presented in this chapter 150
10.2 Run times of benchmarks 155
10.3 Write barrier behavior 160
10.4 Function entry behavior 163

Chapter 1

Introduction

The advantage of a non-strict, purely functional language such as Haskell lies in its clean equational semantics. This clean semantics permits the programmer to create high-level abstractions that would ordinarily require special-purpose tools. It should be possible to code using these high-level abstractions without any knowledge of the underlying execution model. However, lazy implementations of Haskell fall short: they cannot express tail recursion gracefully without annotation. These annotations change the semantics of programs, often destroying their equational semantics.

This thesis will describe *resource-bounded eager evaluation*, a hybrid of strict and lazy evaluation, and its realization in *Eager Haskell*. From the programmer's perspective, Eager Haskell is simply another implementation of Haskell [51, 46, 95, 96] with exactly the same semantics as the usual lazy implementations of the language. Hybrid evaluation provides efficient tail recursion without the need for program annotation. Internally, programs are ordinarily executed in a strict fashion. When resource bounds are exceeded, computation falls back and is restarted lazily. The fallback mechanism uses techniques from lazy language implementations to efficiently suspend and resume ongoing computations.

1.1 Functional languages

The idea of a *functional* language traces its origins to the λ-calculus [22]. Early efforts at turning the lambda calculus into a programming language establish the pattern for later work: A distinction is made between a *pure* language [108], in which variables can be freely instantiated with their value, and an *impure* language with side effects and a simpler implementation. In either case higher-order functions can be written by treating function values in the same way as any other program data.

Later efforts formalize the treatment of arbitrary data structures by introducing algebraic data types and strong static typing.

In this thesis, we refer to higher-order, polymorphically-typed programming languages [139] as functional programming languages. These languages—such as SML [78, 79], Caml and OCaml [66, 65], Id [87, 88], *pH* [85, 86], Clean [104], and Haskell [95]—share many common traits. All of them are rooted at some level in the λ-calculus; functions are lexically scoped, and can be created anonymously, passed as arguments, returned as results, or stored in data structures. All use an elaboration of polymorphic type inference [30] with algebraic data types. Storage is managed by the compiler and run-time system, using some combination of garbage collection and compiler-directed storage allocation.

There are three major differences which distinguish functional languages from one another: fine type structure, language semantics (strict versus non-strict), and execution strategy. For the purposes of this thesis, we will generally ignore typing considerations. Every functional language enriches basic Hindley-Milner polymorphic type inference [30] in a different way, and these enrichments leave a distinctive stamp on the structure and style of programs which are written; however, such differences ordinarily disappear after type checking and explicitly-typed intermediate representations are used.

More important for the purposes of this thesis are the differences in language semantics (strict versus non-strict) specified by a particular language, and in the execution strategy (eager versus lazy) chosen by the language implementation. Language semantics define what results a program must produce. Execution strategy influences every aspect of language implementation, dictating the ways in which programs are run and optimized, and consequently the style in which programs are written. Different execution strategies are suited to particular semantics. In an effort to make distinctions between semantics and strategies clear, Table 1.1 presents a taxonomy of the languages and concepts we discuss in this chapter.

1.1.1 Strict languages

Languages such as Standard ML and OCaml have strict semantics: subexpressions of a program (bindings and function arguments) are evaluated in the order in which they occur in the program. When a function in a strict language is called, every argument is a *value*: A particular integer, a pointer to a list, the closure of a particular function.

There are certain programs that cannot be expressed efficiently in a purely functional language

Languages	ML, OCaml	Id, *pH*	Haskell, Clean
Moniker	"Strict"	"Multithreaded"	"Pure"
Semantics	strict	— non-strict —	
Side effects	— impure —		pure
Strategy	call-by-value	multithreaded	lazy
	— eager —		*hybrid*

Table 1.1: A taxonomy of languages, semantics, and strategies.

with strict semantics [26], but can be expressed efficiently in a procedural language by using side effects, or in a non-strict language by using laziness. As a result, in practice all strict functional languages are *impure*: they include side effects and state. The difference between a strict functional language and a procedural language is therefore a difference of degree: functional languages offer a richer and stronger type system and carefully delimit the scope of side effects by segregating mutable types from pure (immutable) types.

The need for side effects combined with the ordering constraints imposed by strict semantics scuttles strong *equational reasoning*. In practice, this means that manipulating a strict program involves checking many side conditions. For example, we might like to rewrite *head* $[e_1,\ e_2]$ to e_1; however, in a strict language we must prove that e_2 terminates and is side-effect-free before such a transformation is legal. Conventionally, this means that a good deal of code motion (especially loop invariant hoisting and the like) is limited to primitive operations.

1.1.2 Non-strict languages

The Haskell programming language is *non-strict*. We need not compute function arguments before invoking a function; instead, we interleave the computation of arguments and results. This makes it possible to write arbitrary recursive bindings: the value returned by a computation may be used (possibly indirectly) in that computation. For example, the following binding creates a cyclic list:

$$\begin{aligned} &\mathit{oneTwos} && = 1 : \mathit{two} : \mathit{oneTwos} \\ &\quad \textbf{where } \mathit{two} && = \mathit{head}\ \mathit{oneTwos} + 1 \end{aligned}$$

In strict languages, only functions may be recursively defined, and cyclic structures such as *oneTwos* must be created using mutation. In a non-strict language we can write cyclic definitions directly. It falls to the language implementation to perform the necessary side effects to create a cyclic data structure. These side effects must be correctly interleaved with the computation of *two*; regardless of

the strategy used to perform this interleaving, the control structure of the underlying implementation must be sophisticated.

A pure language also permits infinite data structures to be expressed in a regular way. For example, we can construct the infinite list of fibonacci numbers using one line of code:

$$\begin{array}{ll} \textbf{let } \textit{fibList} & = \; 1 : 1 : \textit{zipWith } (+) \; \textit{fibList } (\textit{tail } \textit{fibList}) \\ \textbf{in } \textit{fibList} \; !! \; 100 & \end{array}$$

Note that a non-strict semantics must define carefully what is meant by an *answer*. In this example it should be sufficient to compute and return the 100^{th} element of *fibList* (which in turn requires the previous 99 elements to be evaluated). The expression *fibList* !! 100 is the *root* of computation; in non-strict programming languages it is customary to evaluate until the root is in Weak Head Normal Form (a simple syntactic constraint) [92].

It is particularly natural to realize definitions such as *fibList* using a lazy evaluation strategy, where list elements are computed only when they are needed. Reduction strategy is often closely bound to particular semantics in this way. An unfortunate consequence of this close association is that details of reduction strategy often impinge upon semantics. If this happens, even small changes to reduction strategy may have unintended consequences for the meaning of programs.

Historically, non-strict evaluation has been used in languages such as Id and *pH* to exploit the parallelism implicit in a functional language. Id and *pH* incorporate impure features such as barriers and side effects. In an impure language there may be outstanding side effects or barriers after a result is obtained. The semantics must therefore define a notion of *termination*, which is defined for the entire program rather than just the root. Consequently, even the side-effect-free subsets of these languages have a weak equational semantics (comparable to those of strict languages). For example, the *fibList* example returns a result but does not terminate. The semantics of non-strict languages with barriers have been explored extensively in the context of Id and *pH* [126, 3, 2, 17, 18]. These languages represent a middle ground between strict languages and purely functional non-strict languages.

1.1.3 The importance of purity

By contrast, Haskell has a strong equational semantics, made possible by the fact that it is pure (for convenience, we refer to non-strict, purely-functional languages simply as “pure” in the remainder of this thesis). This encourages a style where high-level program manipulation is possible, permitting a high degree of meta-linguistic abstraction [1] within Haskell itself. For example, in most

```
opGroup op term  =  exp
where exp        =  exp +.+ op +.+ term   >>-  Op
                 ||!  term

expr             =  opGroup (lit '+') term

term             =  opGroup (lit '*') factor

factor           =  lit '(' ..+ expr +.. lit ')'
                 ||!  var                 >>-  Var
                 ||!  const               >>-  Const
```

Figure 1-1: Simple expression parser written using parsing combinators. The result is both a grammar and a valid Haskell program.

languages we would generate a parser using a standalone tool which takes a grammar and generates code which parses that grammar. In Haskell, a grammar can be written directly as a program using any of a number of suites of parsing combinators. Figure 1-1 the grammar of a simple expression parser, written using the same parsing library used by the Eager Haskell compiler itself. Because this is full-fledged Haskell code, we can write functions such as *opGroup* which generates a grammar fragment; a standalone tool does not permit this kind of abstraction.

In an impure language we cannot express such computations with the same degree of abstraction. The expressive power of a pure language can be used in numerous ways. The Haskell language itself uses monads to encapsulate state and control flow in a purely functional setting. Monads obey a simple set of algebraic rules which can be used to reason about monadic programs [138, 125, 69, 98, 43, 91]. Hudak's textbook [50] is devoted to describing how graphics, animation, and sound can be encapsulated and manipulated naturally within Haskell.

The level of abstraction and expressiveness permitted by equational reasoning is Haskell's most important advantage. It is always possible to re-code a Haskell program in, say, C and usually we will realize a performance gain from doing so. However, the effort and complexity involved in such a project may be prohibitive, and in the worst case will result in marginal performance improvement. It is generally easier and more productive to make algorithmic improvements to the original Haskell program. This can reduce the asymptotic complexity of a program rather than simply speeding it up by a constant factor.

Equational reasoning is used extensively by compilers. Many equational optimizations are quite simple, but are considered critical to efficiency; Santos [110] examines these transformations and

their effects in his dissertation. Even complex and dramatic transformations such as deforestation [135, 137] can be captured using simple equational rules, as in the foldr/build optimization of Gill [38, 39, 40]. Complex equational transformations are often guided by static analysis. The oldest example of such a transformation is full laziness [53], which generalizes the loop invariant hoisting of procedural languages. Arbitrarily complex constant subexpressions (usually including function calls) are hoisted out of arbitrary recursive procedures; analysis identifies subexpressions for which such hoisting will be beneficial.

The root of equational reasoning is an equational semantics. In Haskell, it is *always* safe to replace a variable with its definition, or to name a subexpression and "lift" it out of its original context. Haskell can be specified using a simple order- and context-independent set of evaluation rules (equational rules). The λ-calculus is itself a particularly simple set of equational rules which is then used as the basis for the semantics of all functional programming languages. The λ_C calculus presented in Chapters 2 and 3 elaborates on the λ-calculus, capturing the constructs of desugared Haskell.

1.2 Evaluation Strategies

A program will generally have multiple reducible expressions (redexes) according to our equational semantics. An *evaluation strategy* is a method for identifying one or more of these redexes as candidates for reduction. A *normalizing strategy* guarantees that evaluation will never "get stuck" (signal an error or fail to terminate) if it is possible to produce an answer according to our equational rules. Pure languages require a normalizing strategy. There are a broad range of normalizing evaluation strategies for the λ-calculus [22]. The call-by-name (leftmost) strategy is the simplest, and serves as the inspiration for lazy evaluation. The Gross-Knuth strategy evaluates all the current redexes of a term in parallel; it serves as the inspiration for parallel evaluation strategies based on fair scheduling.

1.2.1 Lazy Evaluation

In principle, *any* normalizing strategy can be used as the basis for implementing a pure language. In practice, most normalizing strategies are inefficient; for example, the call-by-name strategy recomputes the value of an expression each time it is used. Existing Haskell implementations use call-by-need evaluation. An expression is computed only if its value is required; this computation is performed at most once, and the result is remembered for subsequent uses. A distinguished root

computation is used (often implicitly) to determine whether a particular expression is required; the goal of evaluation is to compute the value of this root (in Haskell, the root of computation is *main*; in general the root is comparable to the main function of a procedural language). Lazy evaluation has been formalized with call-by-need semantics [11, 14].

Lazy compilers use very different implementation techniques from those used in procedural languages. In a lazy compiler, particular attention is paid to the creation and invocation of *thunks*. A thunk contains the information required to represent a computation, which is saved until it can be proven that the computation is necessary. When its value is required, a thunk is *forced*. The thunk is then overwritten with its value, so that future attempts to force it simply retrieve the value rather than repeating the computation.

Thunks may have an indefinite lifespan, and the associated computation may require an arbitrary amount of data in order to complete. For this reason, thunks are typically created in the heap. Forcing a thunk, however, triggers computation much like a function call in a procedural language; this may in turn force other thunks and trigger further computation. A stack may therefore be used for forcing thunks just as it is used for nested function calls in a procedural language. In a lazy language, stack frames represent individual computations rather than procedure activations.

The lazy evaluation mechanism itself—creating thunks, testing them to see if they have been evaluated, fetching suspended computations from them, updating them, and so forth—introduces a tremendous amount of overhead which simply does not exist in a procedural language. If the value of a computation is not needed, its thunk is unnecessary; if the value is needed, it is generally cheaper to compute it immediately rather than creating a thunk. Compilers for lazy languages perform extensive analysis in order to eliminate one or more steps in thunk creation and update, and intermediate representations for lazy languages define a virtual machine whose operations represent those steps. Examples include the G-machine [54], TIM [99], and the spineless, tagless G-machine [93]. Compilation of lazy languages is treated extensively in several books [92, 99, 104], though all these treatments are somewhat out of date.

1.2.2 Eager Evaluation

In an eager evaluation strategy, expressions are evaluated as they are encountered (*i.e.* in the order in which they occur in the program text). Strict languages are implemented using call-by-value, an eager evaluation strategy. Call-by-value is not a normalizing strategy. However, under call-by-value there is a simple correspondence between a primitive operation (such as integer addition)

and its realization on the machine (load values into registers and add). We represent the state of a computation as a stack of nested activation frames, each containing the local data for a procedure call. As a result, it is relatively easy to produce an efficient implementation of a strict language using eager evaluation. Compilers for strict languages draw directly on the techniques used to compile procedural languages such as C, Java, and Scheme. Consequently, a program written in a strict language will nearly always run faster and more efficiently than an identical one written in a non-strict language. This efficiency advantage means that compilers for strict languages make universal use of the call-by-value strategy; when we refer to "a strict compiler" we mean "a compiler for a strict language using the eager call-by-value compilation strategy." Appel has written excellent overviews of the compilation of both procedural languages [8] and strict functional languages [7].

1.2.3 Multithreaded strategies for parallelism

Parallel execution strategies have evolved over time, but all are multithreaded eager strategies, that is, they focus on the management of very small threads of computation. Id implementations [133, 49, 42, 84] originally made use of multithreading to tolerate remote memory latencies. Memory accesses occur in two phases: each transaction is initiated, and other code is run until a response is received. Fine-grained threads are a particularly good match for this computation model: computations run freely in parallel when their inputs are available, and block when they are not. A long-latency memory operation and a non-strict dependency look much the same from the perspective of the compiled code.

Unfortunately, the Id evaluation model is not particularly suited to execution on ordinary uniprocessor or shared memory multiprocessor (SMP) machines. On a modern microprocessor, caching masks the latency of memory operations, rewarding applications which exhibit temporal and spatial locality. This is fundamentally at odds with the unstructured control flow of an application using fine-grained threads. Rather than providing a means to tolerate latencies, the mechanisms of multithreaded execution become a source of overhead and complexity on a machine with limited parallelism.

The *pH* compiler [16, 31] addresses these issues, with the goal of running *pH* [86, 85] programs efficiently on an SMP machine. Data is stored on a garbage-collected heap in shared memory; memory access is by ordinary loads and stores. Fine-grained partitions (equivalent to the fine-grained threads of Id implementations) are grouped together into suspensive threads. A suspensive thread allows the compiler to generate an efficient schedule for a group of partitions based on the

assumption that non-strictness will be rare in practice. A work-stealing execution model based on the one in Cilk [37] is used to ensure that parallel computations are as coarse-grained as possible.

The *pH* implementation still shares a number of important properties with its forebears. In *pH* the presence tests required by non-strictness are accomplished using a level of indirection. Every computation has a separately-allocated proxy; the proxy is initially empty, and is eventually updated with a pointer to the actual value it should contain. Parallel execution occurs using a cactus stack: every function call allocates a frame on the heap to contain its local variables (and various suspension-related data structures). By heap-allocating frames, multiple sibling calls can easily coexist in parallel, and computations can suspend and resume without copying execution state between stack and heap. There was some hope that sufficiently fast allocation and garbage collection could be as fast as stack allocation [6, 9]. In practice, however, the advantages of stack-based execution (fast allocation and deallocation of activation frames, good cache locality, ease of interoperability with other languages) are compelling [77], and Eager Haskell uses a stack for function calls.

Meanwhile, if we are to believe Schauser *et. al.* [116] then the picture for multithreading is very bleak indeed. The Id programs they survey are by and large easy to transform into strict code; those that are not amenable to this transformation can be transformed so that all non-strictness is captured in data structures. Conditionals and function calls can be strict. Indeed, Shaw [120] showed that such programs can be efficiently compiled using simple structured parallelism with no need for synchronizing memory (though a certain amount of parallelism is lost in his approach). The result is essentially a strict functional language with automatic parallelization.

1.3 The advantages of eagerness over laziness

In this thesis we deliberately take a very different tack: rather than compiling programs written for multithreaded execution, we attack the much harder problem of compiling a pure language using an eager evaluation strategy. Eager Haskell programs possess the clean equational semantics of a pure, non-strict language which are the root of Haskell's expressiveness; they are *semantically* indistinguishable from the same programs compiled with a lazy Haskell compiler. This is accomplished by using a *hybrid* evaluation strategy which evaluates eagerly by default, but uses laziness to bound resource usage. These resource bounds permit our implementation to handle code such as *fibList* which constructs infinite data structures. In Chapter 4 we characterize the space of hybrid evaluation strategies, including the resource-bounded strategy used in Eager Haskell.

Lazy evaluation performs very badly in the presence of *accumulating parameters*. Consider the following canonical example:

$$
\begin{array}{lcl}
\mathit{fact}\ n & = & \mathit{factLoop}\ n\ 1 \\
\quad \textbf{where}\ \mathit{factLoop}\ 0\ a & = & a \\
\qquad \mathit{factLoop}\ k\ a & = & \mathit{factLoop}\ (k-1)\ (a*k)
\end{array}
$$

In a strict functional language, code such as this is typical—we express the iterative computation *fact* as a tail-recursive function with the accumulating parameter *a*. Tail recursion does not consume stack space, and no allocation is required. When we use a lazy language implementation, the picture is dramatically different: *a* must not be computed unless it is used. For example, if we evaluate *fact* 5, we will eventually create five nested thunks representing the computation . When *factLoop* returns, the resulting chain of thunks will finally be forced and the multiply operations will actually be performed.

An an optimizing Haskell compiler such as hbc [20] or the Glasgow Haskell Compiler [130] can use strictness information to efficiently compile *factLoop*; it can be shown that *a* must eventually be used and can therefore be passed by value. However, accumulating parameters need not be used strictly, in which case no optimization is possible. Further, different Haskell systems will produce very different results for code such as this depending on the precision of strictness analysis (an unoptimizing system such as hugs [59] or nhc [106] will always build the chain of closures). In order to obtain consistent behavior, *factLoop* must be annotated using *seq*:

$$
\begin{array}{lcl}
\mathit{fact}\ n & = & \mathit{factLoop}\ n\ 1 \\
\quad \textbf{where}\ \mathit{factLoop}\ 0\ a & = & a \\
\qquad \mathit{factLoop}\ k\ a & = & \mathit{ak}\ \text{`seq`}\ \mathit{factLoop}\ (k-1)\ \mathit{ak} \\
\qquad\quad \textbf{where}\ \mathit{ak} & = & a*k
\end{array}
$$

This says "make sure *ak* is evaluated before calling *factLoop*". We can also use strict function application \$!, which is defined in terms of *seq*:

$$
\begin{array}{lcl}
\mathit{fact}\ n & = & \mathit{factLoop}\ n\ 1 \\
\quad \textbf{where}\ \mathit{factLoop}\ 0\ a & = & a \\
\qquad \mathit{factLoop}\ k\ a & = & \mathit{factLoop}\ (k-1)\ \$!\ a*k
\end{array}
$$

In either case, the annotation is consistent with program semantics; in general, however, adding annotations changes the semantic behavior of our programs, and can be a source of unintended deadlock.[1]

[1] An interesting example: as this dissertation was undergoing final revision, it was discovered by others that *seq* annotations actually destroy the equational semantics of monadic operations in the presence of divergence.

The need for annotation is Haskell's biggest performance pitfall. A glance at the Haskell and Haskell-cafe mailing lists in March 2002 (a fairly typical month in the author's experience) revealed 41 messages (out of a total of 398) discussing how to structure and annotate tail recursive code so that it behaves reliably and efficiently. Only two or three of the remaining messages in that time were performance-related. Even more messages are routed to compiler-specific mailing lists; programmers assume the stack and heap overflows they see are the fault of a compiler bug and not a simple consequence of constructing and forcing an overly-long chain of lazy computations.

In Eager Haskell we solve this problem. Hybrid evaluation means that accumulating parameters will ordinarily be eagerly evaluated as we compute. If, however, we encounter an expensive computation, we will fall back to lazy evaluation in the hopes that the result will be discarded. Under this regime, an iteration using accumulating parameters will never have more than a single associated thunk at a time.

1.4 Contributions

This thesis describes resource-bounded hybrid evaluation, a novel execution strategy for non-strict, purely functional programming languages, and its realization in the Eager Haskell compiler. Hybrid evaluation permits iterative algorithms to be expressed using tail recursion, without the need to resort to program annotations. For a list comprehension version of Queens, our most tail-recursion-intensive benchmark, Eager Haskell outperforms the Glasgow Haskell Compiler (GHC) by 28%. Adding annotations to this benchmark speeds up GHC by up to 8%; in order to match the performance of Eager Haskell, however, list comprehensions must be re-written as explicit recursion, which effectively means re-coding the benchmark. Similarly, on the multiplier benchmark strictness annotations speed GHC up by 20%. The same annotations have no significant effect in Eager Haskell. We believe that by eliminating the need for annotation hybrid evaluation represents the better choice of execution strategy for Haskell.

Under hybrid evaluation, code is ordinarily executed in program order just as in a strict functional language. When particular stack, heap, or time bounds are exceeded, an exception is signaled, and computation falls back. During fallback function calls are transformed into thunks and program execution gradually suspends. When fallback has completed, execution restarts with the root. New computations proceed eagerly as before; existing thunks, however, are only forced on demand. Unlike previous eager language implementations, Eager Haskell has exactly the same clean equational

semantics as lazy Haskell does. However, eager evaluation avoids one of the biggest pitfalls of lazy evaluation: the inability to express iteration in a clean and natural way.

Resource-bounded execution bounds the amount of computation which cannot be reached from the root. This effectively results in a fixed-size tile of computation being performed between each fallback step. Each fallback step results in progress on the root computation. We therefore conjecture that the worst-case runtime and space of an Eager Haskell program is a constant factor larger than the equivalent bounds for lazy execution. However, the reduction in space consumption of tail recursion are likely to reduce asymptotic space usage in many cases.

This thesis makes a number of smaller contributions which are relevant to other language implementations:

- In order to better understand the performance consequences of various reduction strategies and program transformations, we have developed a semantics for λ_C (a realization of our compiler's intermediate representation). Building on Ariola and Arvind [12] we associate particular implementation mechanisms with corresponding semantic actions. The use of a common equational framework to compare radically different language implementation strategies is novel; previous efforts in this direction have focused on drawing semantic distinctions or on modeling a particular implementation.

- Eager Haskell is (to our knowledge) the first eager, non-strict functional language implementation to make use of a call stack, which is far more efficient in the common case than placing frames on the heap.

- By offloading the overhead of non-strictness to the run-time system, dramatic changes in the execution strategy of Eager Haskell are possible by making changes to the run-time system without modifying the compiler itself.

- Functions in a memory-safe language such as Haskell contain many error checks. For example, the *head* operation on lists must check its argument and signal an error if it is the empty list. We refer to an expression which always signals an error as a *divergent* expression. Often (as in *head*) these expressions account for the majority of code in a particular function. This code is rarely executed, and should not be inlined. In addition, divergent expressions have a very strong equational semantics: no expression which depends upon a divergent expression can execute. A number of important program transformations are based upon this fact.

The compiler identifies divergent expressions statically using type and strictness information. Divergence information is used to segregate error handling code from regular control flow to avoid the cost of inlining error handlers, and to guide transformations around divergent expressions.

- A divergent expression must not be evaluated eagerly; if the answer does not depend on the divergent expression, doing so will terminate the program spuriously. Divergent expressions are compiled into special *bottom thunks*; these are always evaluated lazily, preserving normalization in the face of error handling.

- The object representation in Eager Haskell combines the idea of object update from lazy languages with the presence-checked I-structure memory conventionally used by multithreaded compilers.

- The Eager Haskell garbage collector uses the stack as an additional predictor of object lifetime: near the top of the stack, where old and new objects are intermixed, objects are assumed to be likely to be updated or to die and will not be promoted. Old objects lower down in the stack are likely to remain untouched and therefore are good candidates for promotion.

1.5 Overview of this thesis

We begin the thesis by presenting the Eager Haskell compiler's intermediate representation, λ_C. Chapter 2 describes the constructs in λ_C. Chapter 3 present an exhaustive semantics for λ_C, including a number of extensional rules crucial to equational reasoning but not required for an operational reading. Chapter 4 characterizes different execution strategies for λ_C. Between lazy and eager evaluation, there is an enormous space of possible eager execution strategies. A hybrid strategy combines execution rules from both strict and lazy strategies, resulting in a stack-based eager evaluation strategy. Resource-bounded execution is one point within a broad range of hybrid strategies.

Having described the resource-bounded strategy, we shift our focus to implementation. Chapter 5 describes the data structures and run-time mechanisms used by the Eager Haskell compiler. Chapter 6 describes the various program transformations, most notably the insertion of explicit synchronization operations, required to turn λ_C into a form suitable for code generation. Chapter 7 describes the final code generation step which maps canonical λ_C to C code.

Having described the basic hybrid implementation, the next two chapters fill in crucial detail.

Chapter 8 explains how type and strictness information are used to identify divergent expressions, and how these expressions are handled by the compiler and the run-time system. Chapter 9 describes the implementation of arrays in Eager Haskell. Preserving non-strict array semantics in the face of resource-bounded execution is a daunting task: when array initialization suspends, it is not clear which computations should resume and in what order.

Chapter 10 presents benchmark results for Eager Haskell. We compare Eager Haskell with the Glasgow Haskell Compiler, the only optimizing Haskell compiler currently under active development. The Eager Haskell implementation is able to beat GHC on several benchmarks, but is about 60% slower overall. Much of this figure is due to the relative maturity of the two compilers. We then examine various aspects of the compiled code and run-time system in detail. Those applications which perform poorly often stress particular aspects of the runtime; this code is relatively slow compared to compiled code. Finally, we conclude Chapter 10 with a case study of tail recursion. Annotations in the multiplier benchmark speed it up by 20% under GHC, but have no effect in Eager Haskell.

We conclude the thesis with several chapters on future work. In Chapter 11 we describe how the present Eager Haskell implementation can be made to run on a multiprocessor. We propose to schedule computation based on randomized work stealing. The fallback mechanism is a natural match for indolent task creation; work stacks contain thunks which are created during fallback. Our design avoids locking whenever possible; we articulate a general monotonicity principle, and use it to devise a protocol for lock-free update of shared data.

In Chapter 12, we describe how to add barriers to Eager Haskell, allowing *pH* programs to be run using the new compiler and run-time system. The record-keeping required to track suspended work closely parallels the mechanisms used to track work in our array implementation, and to track outstanding thunks in the multiprocessor version of Eager Haskell.

Finally, in Chapter 13 we conclude by looking at the future of uniprocessor Eager Haskell. A number of crucial compiler optimizations are either missing or severely crippled. There is ample potential for future work on problems such as unboxing in an Eager setting. In light of the results in Chapter 10 it should be possible to tune the run-time system to better handle the most common uncommon cases. Overall, however, the system is solid and the promise of hybrid evaluation is evident.

Chapter 2

Representing Eager Programs: The λ_C Calculus

This thesis assumes that the source syntax of Haskell has been compiled into a simpler intermediate representation, λ_C. The techniques for performing this translation—type checking, pattern-matching compilation, and desugaring of list comprehensions and monad syntax—are well-understood and widely used. They are covered exhaustively in other sources [99, 92, 95, 90]. This chapter gives a brief overview of the features of λ_C, and introduces the terminology and notation used in this thesis. A semantics for λ_C is given in the next chapter.

2.1 Overview

The syntax of λ_C is summarized in Figure 2-1. It has a great deal in common with other calculi used for reasoning about non-strict languages. The most important commonality is the use of a **let** construct to capture sharing. It has a few distinguishing features:

- Data structures are created using constructors and examined using case expressions.
- Case expressions are used to force or synchronize evaluation.
- Arity of functions and of applications is explicitly represented.
- Primitives applications are distinguished.
- Recursive (**letrec**) and non-recursive (**let**) bindings are distinguished.

$e \in E$	$::=$	x		Variable
	$\mid$	$E\ \vec{E}_k$	$k > 0$	Application
	$\mid$	$p_k\ \vec{E}_k$	$k > 0$	Primitive appl.
	$\mid$	$C_k\ \vec{E}_k$	$k \geqslant 0$	Constructor appl.
	$\mid$	$\lambda \vec{x}_k \rightarrow E$	$k \geqslant 0$	Function
	$\mid$	**case** $x = E$ **of** D		
	$\mid$	**let** $x = E_1$ **in** E_2	$x \notin FV[E_1]$	
	$\mid$	**letrec** B **in** E		
$b \in B$	$::=$	$x = E$		Binding
	$\mid$	$B\,;\ B$		Group
	$\mid$	ϵ		Empty binding
$d \in D$	$::=$	$C_k\ \vec{x}_k \rightarrow E$		Regular disjunct
	$\mid$	$\_ \rightarrow E$		Default
	$\mid$	$D\,;\ D$		Disjunct group
	$\mid$	ϵ		Empty disjunct
$v \in V$	$::=$	$C_k\ \vec{x}_k$	$k \geqslant 0$	Simple constructor
	$\mid$	$(\lambda \vec{x}_k \rightarrow E)\ \vec{y}_j$	$0 \leqslant j < k$	Closure

Figure 2-1: Syntax of λ_C

The λ_C calculus formalized here is an outgrowth of previous work on the λ_S calculus [17, 18]. The use of the **case** construct for strict binding (Section 2.8) was inspired by the Core syntax at the heart of the Glasgow Haskell Compiler [103]. Unlike Core, λ_C is not explicitly typed. In this sense λ_C is more similar to its precursor λ_S. However, λ_C eliminates impure constructs, a central theme of λ_S, and substitutes strict binding using **case**; in this respect is is equivalent in power to ordinary λ calculi. All of these calculi are close relatives of the call-by-need λ-calculus [14], the most widely-studied lambda calculus with explicit sharing. However, unlike the call-by-need calculus but like Core, λ_C is designed to cleanly represent actual Haskell programs during compilation.

2.2 Notation

In this and subsequent sections we will use a few convenient shorthands to simplify the notation. Eager Haskell is lexically scoped, and the usual assumptions about renaming of variables are made to prevent scope conflicts. The notation $\vec{x}_k$ stands for the k-ary vector of variables $x_1,\ x_2 \ldots x_k$. The ith element of such a vector is written x_i. We omit k and write $\vec{x}$ when arity is clear from context (for example, the arity of primitive and constructor applications is fixed by the syntax). The vector

notation extends to all syntactic elements in the obvious way. We will sometimes want to replace the ith element of such a vector; in this case we write the vector out longhand like so:

$$x_1 \ldots x_i \ldots x_k \longrightarrow x_1 \ldots E' \ldots x_k$$

or we can abbreviate the source vector, when x_i is arbitrary or is constrained elsewhere:

$$\vec{x}_k \longrightarrow x_1 \ldots E' \ldots x_k$$

In either case, we understand that x_1 and x_k are placeholders, and $1 \leqslant i \leqslant k$.

Variables scope straightforwardly. The arguments $\vec{x}$ of a λ-expression $\lambda\vec{x} \to e$ scope over the body e. The variable x in the binder of the **case** expression **case** $x = e$ **of** D scopes over the disjuncts D, and the variables $\vec{x}$ of the disjunct binder $C_k\ x_k \to e$ scope over the right-hand side e. Finally, the binding for a variable x in a binding block **letrec** $x = e_1\ ;\ B$ **in** e_2 extends over the entire **letrec** expression. Indeed, the order of bindings in a block does not matter (a notion which we formalize in Figure 3-1), and we will by convention place interesting bindings leftmost in our presentation.

2.3 Functions

We write the k combinator in λ_C as follows:

$$\lambda a\ b \to a$$

In general, functions use a version of the familiar lambda notation:

$$\lambda\vec{x}_k \to E$$

This notation indicates a function of *arity k*; the x_i are the *arguments* of the function and the expression E is the *body* of the function. Semantically, nested functions $\lambda\vec{x}_i \to \lambda\vec{y}_j \to E$ are equivalent to a single function with combined arity $i + j$, $\lambda\vec{x}_i\ \vec{y}_j \to E$. Operationally, however, we treat these two expressions very differently, and our compiler will produce different code for them. It is for this reason that λ_C represents function arity explicitly.

2.4 Application

Function application in λ_C is by juxtaposition as usual:

$$k\ a\ b$$

Here we apply the function *k* to two arguments *a* and *b*. Like functions, application have an associated arity:

$$E\ \vec{E}_k$$

Again, nested applications $((f\ \vec{x}_i)\ \vec{y}_j)$ are semantically equivalent to a single application $(f\ \vec{x}_i\ \vec{y}_j)$, but imply very different operational readings. In particular, note that a *k*-ary function may be *partially applied* to fewer than *k* arguments (say *i*) yielding a function of $k - i$ arguments.

2.5 Blocks

We can name the *k* combinator as follows:

$$k = \lambda a\ b \to a$$

In general, we can bind any expression:

$$x = E$$

We refer to this as a *binding for x*. *E* is the *definition* of *x*; *x* is *bound to E*.

Bindings are grouped together into *blocks*, or let-expressions. These are written as follows in Haskell:

let *two* = *g x x*
    four = *g two two*
in *g four four*

The λ_C notation is similar. However, we use **letrec** to indicate the possibility that the bindings may be mutually recursive:

letrec *t* = *g f f*
        f = *g t t*
in *t f*

The expression *t f* is the *result* of the block. Mutual recursion allows a definition to refer to any of the variables bound in the same block:

letrec *r* = *k x r* **in** *r*

For pragmatic reasons, it is frequently useful to distinguish non-recursive bindings. We use **let** to indicate a single, non-recursive binding:

let $t = g\ x\ x$
in let $f = g\ t\ t$
  in $g\ f\ f$

As with function arity, the distinction between **let** and **letrec** has no semantic impact, but affects details of code generation and manipulation.

2.6 Primitives

Primitive operations are not considered to be function symbols. Instead, λ_C has distinguished syntax for a *primitive application*:

intplus a b

All primitive applications are *saturated—i.e.* a primitive of arity *k* is always applied to exactly *k* arguments. We denote *k*-ary primitives with the notation p_k. Particular primitives are distinguished using superscripts, as in p_k^i:

$p_k^i\ \vec{E}_k$

The Haskell language allows primitives to be partially applied, passed as arguments, and so forth. Saturating primitive applications is simple: replace every occurrence of primitive p_k^i with a *k*-ary function.

$$p_k^i \qquad \longrightarrow (\lambda \vec{x}_k \rightarrow p_k^i\ \vec{x}_k)$$

This saturation operation is performed by the compiler as it transforms Haskell source into λ_C.

In practice, many of the primitive operations in Eager Haskell are familiar mathematical functions. We will use infix notation freely to keep sample code clear. Thus we write *fib* $(n-1) +$ *fib* $(n-2)$ rather than *intplus* (*fib* (*intminus n* 1)) (*fib* (*intminus n* 2)).

2.7 Algebraic Data Types

In Haskell, algebraic types neatly combine sum and product constructs into a single mechanism. For example:

data *Tree* α = *Node* (*Tree* α) α (*Tree* α)
  | *Leaf*
tree = *Node* (*Node Leaf* 2 *Leaf*) 5 (*Node Leaf* 7 *Leaf*)

This declaration states that an object of type *Tree* α is either a *Leaf* (which we refer to as a constant or a nullary constructor), or it is a *Node* with a left subtree, a datum (of type α) and a right subtree. In λ_C we omit **data** declarations, and distinguish constructors by using uppercase (a convention which is observed in Haskell as well):

$$t = N\ (N\ L\ 2\ L)\ 5\ (N\ L\ 2\ L)$$

Instead of explicit data type declarations, we define the functions $\tau[C_k]$ which returns the type of a constructor and $\kappa[\tau]$ which yields the set of constructors of a particular type. We read $\kappa[C_k]$ as $\kappa[\tau[C_k]]$ in the obvious way. Thus $\kappa[N] = \{N, L\}$.

We use the notation C_0 to indicate an arbitrary constant. These include the integers, floating-point numbers, unary algebraic constructors, and the like. Similarly, *k*-ary constructors are notated as C_k, with particular constants or constructors using superscripts, as in C_k^i:

$$C_k^i\ \vec{E}_k$$

Like primitive applications, constructor applications are saturated.

We use Haskell's notation for lists and tuples. The empty list is written $[\,]$ and cons is written as infix colon $(:)$. Square braces make longer lists more readable:

$$1 : 1 : 2 : 3 : 5 : [\,] \equiv [1,\ 1,\ 2,\ 3,\ 5]$$

2.8 Case Expressions

The **case** construct is the most complex part of λ_C. Consider the following **case** expression:

case *cs* = *xs* **of**
  a : *as* → *revApp as* (*a* : *ys*)
  _ → *ys*

This **case** expression *scrutinizes* the *discriminant xs*. It first ensures *xs* is evaluated (it is a value, *V* in Figure 2-1). Second, the resulting value is bound to the name *cs*. Third, one of the two *disjuncts* is selected based on the value of *xs*. Disjuncts are labeled with a single constructor; the final disjunct may instead be labeled as the *default* disjunct using _ (as it is here). If *xs* is a cons cell, it will match the first disjunct *a* : *as*; it is then *projected*: *a* is bound to the head of *xs* and *as* is bound to its tail. Finally, the *body* of the disjunct, *revApp as* (*a* : *ys*), is executed.

A case expression thus serves four different purposes in the language:

- Ensuring its discriminant has been evaluated.
- Naming the discriminant, with the guarantee that this name is always bound to a value.
- Selecting an execution path based on the value of the discriminant.
- Fetching fields from objects with algebraic type.

Note that the typing rules of Haskell guarantee that all the constructors which appear in a particular **case** expression belong to the same type. Only one disjunct may be labeled with a given constructor. For clarity, we prefer to avoid default disjuncts when a single constructor is missing from a case expression—thus, we should replace the default disjunct in the above expression with the empty list:

case $cs = xs$ **of**
  $a : as \rightarrow \mathit{revApp}\ as\ (a : ys)$
  $[\,] \rightarrow ys$

In practice, a **case** expression may not perform *all* of the above functions. Ordinarily we do not name the discriminant, and thus omit this notation from the case expression:

case xs **of**
  $a : as \rightarrow \mathit{revApp}\ as\ (a : ys)$
  $[\,] \rightarrow ys$

When a datatype has a single disjunct, as is the case with tuple types, there is no need to select a disjunct for execution.

case *pair* **of**
  $(a,\ \_) \rightarrow a$

Naturally, not every constructor has fields, and thus fetching is not always necessary (it does not occur in the $[\,]$ disjunct in *revApp*). Similarly, if a field of a constructor is not required (as in the above definition), it is replaced with the wildcard pattern $\_$ and is not fetched.

We consider an expression to have terminated when it is a value. Sometimes the compiler must ensure that a particular expression has been computed, but its actual value does not matter. Every expression in λ_C is lifted—it admits the possibility of non-termination. We refer to non-terminating expressions as *bottoms*, notated $\perp$. A **case** expression with only a default disjunct checks that its discriminant has terminated:

$$\textbf{case } a \textbf{ of}$$
$$\quad \_ \rightarrow a + 1$$

We call the resulting expression a *touch* (by analogy with the touch operation in the SMT eager abstract machine [16]). Unlike other **case** expressions, the discriminant of a touch may have any type. The *seq* operator in Haskell is represented by a touch operation:

$$seq\ a\ b = \textbf{case } a \textbf{ of}$$
$$\qquad\qquad\qquad \_ \rightarrow b$$

Touches are inserted by the compiler in order to control program evaluation.

Not every type in Haskell is lifted. The **newtype** declaration creates an unlifted "wrapper type"; this type has a single constructor which accepts a single (lifted) component as an argument. As a result, **case** expressions and constructors for such types can be eliminated from the program. They are erased after type checking, and are not represented in λ_C.

2.9 Other syntax

The syntax for λ_C in Figure 2-1 includes a few constructs that have not been described thus far. First, it defines the notion of a *value*. Values are simple constructor expressions and unsaturated function applications (closures). Second, our notation separates bindings and disjuncts with a semicolon. As in Haskell, we omit these semicolons when indentation makes the grouping clear. Finally, it is convenient to add an empty grouping construct ϵ—for example, this allows us to treat a singleton binding as a binding group. This eliminates a special case from many rules.

Chapter 3

The Semantics of λ_C

In the Chapter 2 we presented the syntax of λ_C, the calculus we use as an intermediate representation for Eager Haskell programs. In this chapter we present a semantics for λ_C. The semantics of λ_C are equivalent in power to the λ-calculus, the call-by-need λ-calculus, or the core calculus of GHC [103]. Our presentation is unique in several ways. Unlike core, we give small-step reduction semantics for the **case** construct. We also include a limited set of extensional conversions. Such conversions are not included in the call-by-need λ-calculus [14], and do not appear to have been widely studied for **let**-based calculi in general.

3.1 Extensionality

Our presentation of λ_C begins by focusing on *conversion* rather than *reduction*. We examine conversion because the **case** construct simultaneously expresses three semantic properties: Projection of products, extraction of sums (coproducts), and unlifting of lifted data [80, Chap 2]. The equational theory of λ_C therefore includes extensional expansion rules for these three uses of case.

The purpose of extensionality is to capture observational equivalence: we say two terms M and N are observationally equivalent when they behave the same way in all contexts. A calculus is extensional when convertibility $M = N$ is congruent to observational equivalence for all terms [22]. We express extensional rules as expansions; the virtues (such as confluence) of expansion rather than contraction for extensionality have been revisited in recent years [52].[1]

[1]Note that in a calculus with constants, extensional equivalence usually requires a typed calculus. For example, η-expansion must guarantee that the expression being expanded is of function type. The limited expansions permitted in λ_C are type-independent; rules such as η_a will be meaningless (but harmless) if performed at the wrong type.

Identities on bindings			
$\epsilon\ ;\ B$	$\equiv$	B	unit
$B_0\ ;\ B_1$	$\equiv$	$B_1\ ;\ B_0$	commutative
$B_0\ ;\ (B_1\ ;\ B_2)$	$\equiv$	$(B_0\ ;\ B_1)\ ;\ B_2$	associative
Identities on disjuncts			
$C_k^i\ \vec{x} \rightarrow E_0\ ;\ C_k^i\ \vec{y} \rightarrow E_1$	$\equiv$	$C_k^i\ \vec{x} \rightarrow E_0$	redundant
$\_ \rightarrow E_0\ \ ;\ C_k^i\ \vec{y} \rightarrow E_1$	$\equiv$	$\_ \rightarrow E_0$	default
$C_k^i\ \vec{x} \rightarrow E_0\ ;\ C_l^j\ \vec{y} \rightarrow E_1$	$\equiv$	$C_l^j\ \vec{y} \rightarrow E_1\ ;\ C_k^i\ \vec{x} \rightarrow E_0$	independent

Figure 3-1: Syntactic equivalences for terms in λ_C. The usual α renaming equivalences are assumed. Reduction rules are obtained by reading conversions left to right.

The conversions given in this section do not make λ_C as a whole extensional; there are equivalent terms such as $\lambda x \rightarrow f\ x$ and f which are not convertible. Instead, the new rules simplify the correctness proofs for many widely-used program transformations. They also serve to reinforce the parallels between the use of **case** and constructors in λ_C and the use of constructs such as pairing and projection in other λ-calculi.

3.2 Equivalence

We consider terms to be equivalent (written $\equiv$) when they differ in unimportant ways. In Chapter 2 we noted that the order of bindings in **letrec** expressions does not matter; we formalize this with the first three rules in Figure 3-1. Similarly, our disjunct syntax is very permissive. The order of independent disjuncts is unimportant. Unusually, we permit multiple disjuncts with the same constructor. In this case the first disjunct takes precedence. Similarly, a default disjunct supersedes the disjuncts which follow it. These equivalences simplify our rules for merging and splitting **case** expressions (particularly η_a and χ_p, see Section 3.3.3). A compiler, of course, simply erases useless disjuncts.

We omit α-renaming rules from the figure. *Throughout this thesis we assume the usual α-equivalences hold, and all conversions hold modulo the usual rules prohibiting name capture.* In the Eager Haskell compiler, all program identifiers are distinct; renaming occurs when a λ_C term is duplicated, for example when a function is instantiated or a new arm is added to a **case**. Fresh names are used when new variables are introduced by naming or expansion.

$(\lambda x_1\ \vec{x}_k \to e)\ y$	$=$	$(\lambda \vec{x}_k \to e\ [y\ /\ x_1])$	β_{var}
$p^i_k\ \vec{v}_k$	$=$	$\underline{p^i_k\ \vec{v}_k}$	δ axiom schema
case $y = C^i_k\ \vec{y}$ **of** $C^i_k\ \vec{x} \to e\ ;\ d$	$=$	**case** $y = C^i_k\ \vec{y}$ **of** $\_ \to e\ [\vec{y}\ /\ \vec{x}]$	χ_c (constructor)
case $y = C^i_k\ \vec{e}$ **of** $C^j_l\ \vec{x} \to e\ ;\ d$	$=$	**case** $y = C^i_k\ \vec{e}$ **of** d	χ_p (mismatch)
case $x = v$ **of** $\_ \to e$	$=$	**let** $x = v$ **in** e	χ_d (discharge)
$\_ \to e$	$=$	$C^j_l\ \vec{x} \to e\ ;\ \_ \to e$	η_a (new arm)
e	$=$	**case** $x = e$ **of** $\_ \to x$	η_l (lift)
letrec $x = e\ ;\ b$ **in** $I_E[x]$	$=$	**letrec** $x = e\ ;\ b$ **in** $I_E[e]$	ι_e (instantiate)
$x = e\ ;\ I_B[x]$	$=$	$x = e\ ;\ I_B[e]$	ι_b (inst. binding)
$x = I_E[x]$	$=$	$x = I_E[I_E[x]]$	ι_r (inst. rec.)
$C^i_k\ \vec{x} \to I_E[x]$	$=$	$C^i_k\ \vec{x} \to I_E[C^i_k\ \vec{x}]$	ι_d (inst. disj.)
case $x = e$ **of** $I_D[x]$	$=$	**case** $x = e$ **of** $I_D[e]$	ι_c (inst. case)
case $x = e$ **of** $I_D[$ **case** $y = x$ **of** $\_ \to e]$	$=$	**case** $x = e$ **of** $I_D[$ **let** $y = x$ **in** $e]$	ι_l (unlift)
e	$=$	**let** $x = e$ **in** x	ν (name)
let $x = e_1$ **in** e_2	$=$	**letrec** $x = e_1$ **in** e_2 $x \notin FV[e_1]$	ρ (let conv.)
$I_E[$ **letrec** b **in** $e]$	$=$	**letrec** b **in** $I_E[e]$ $FV[I_E[\,]] \cap BV[b] = \emptyset$	σ_m (hoist)
letrec b **in** $($**case** $x = e_0$ **of** $\_ \to e_1)$	$=$	**case** $x = e_0$ **of** $\_ \to$ **letrec** b **in** e_1 $FV[e_0] \cap BV[b] = \emptyset$	σ_l (case hoist)
$S[$ **case** $x = e_0$ **of** $\_ \to e_1]$	$=$	**case** $x = e_0$ **of** $\_ \to S[e_1]$ $x \notin FV[S[\,]]$	σ_s (strict hoist)
case $x = e_0$ **of** $\_ \to$ **case** $y = e_1$ **of** $\_ \to e_2$	$=$	**case** $y = e_1$ **of** $\_ \to$ **case** $x = e_0$ **of** $\_ \to e_2$ $x \notin FV[e_1] \wedge y \notin FV[e_0]$	σ_t (swap touch)
$\lambda \vec{x}_k \to \lambda \vec{y}_j \to e$	$=$	$\lambda \vec{x}_k\ \vec{y}_j \to e$	τ_l (merge λ)
$(e\ e_1 \ldots e_j)\ e_{j+1} \ldots e_k$	$=$	$e\ e_1 \ldots e_k$	τ_a (merge app)
$x =$ **letrec** b **in** e	$=$	$b\ ;\ x = e$	τ_f (flatten)
letrec b_1 **in letrec** b_2 **in** e	$=$	**letrec** $b_1\ ;\ b_2$ **in** e $FV[b_1] \cap BV[b_2] = \emptyset$	τ_m (merge let)
letrec ϵ **in** e	$=$	e	ϵ_d (drop)
letrec $b_1\ ;\ b_2$ **in** e	$=$	**letrec** b_2 **in** e $BV[b_1] \cap FV[$ **letrec** b_2 **in** $e] = \emptyset$	ϵ_e (erase)

Figure 3-2: Conversion in λ_C. All conversions hold modulo the usual rules prohibiting name capture.

3.3 Conversion

We initially view the λ_C calculus as a purely mathematical system. In Figure 3-2 we list the conversions among terms, notated $=$. When it comes time to actually *evaluate* a λ_C program, rather than manipulate it equationally, we require only a limited subset of the conversions presented here. We present these rules in Figure 3-12; Chapter 4 is devoted exclusively to various reduction strategies for λ_C. When speaking of evaluation, we view the conversions as left-to-right reduction rules.

3.3.1 Functions

The most fundamental rule in any λ-calculus is β-reduction. In λ_C we make use of a very minimalist form, β_{var}. Only arguments which are variables can be reduced, and reduction occurs by simply substituting the parameter variable for the argument variable. Most **let**-based λ-calculi (such as the call-by-need calculus, λ_S, and Core) make use of the $\beta_{\textbf{let}}$ rule instead. We can derive $\beta_{\textbf{let}}$ in λ_C as follows:

$$\begin{array}{lcll} (\lambda x \to e_1)\ e_2 & = & (\lambda x \to e_1)\ (\textbf{let}\ y = e_2\ \textbf{in}\ y) & (\nu) \\ & = & \textbf{let}\ y = e_2\ \textbf{in}\ (\lambda x \to e_1)\ y & (\sigma_m) \\ & = & \textbf{let}\ y = e_2\ \textbf{in}\ e_1\ [y\ /\ x] & (\beta_{\text{var}}) \\ & \equiv & \textbf{let}\ x = e_2\ \textbf{in}\ e_1 & (\alpha) \end{array}$$

As we shall see, the β_{var} rule arises naturally out of the desire to separate the static portions of reduction (ν and σ) from dynamic β reduction itself.

The β_{var} rule is *unary*—we disregard the arity of functions and applications. We include merge rules (τ_l, τ_a), but it should be clear that for the purposes of β reduction they are unimportant. When we read λ_C operationally, we will restrict β_{var} to full arity, and τ_a will be used in taking apart closures of partial applications. Lambda merging (τ_l) optimizes the process of reducing function applications.

Note that there is no η rule for function application. In an untyped calculus with constants, there are only two ways to determine that a term must represent a function: either it is actually a lambda expression, or it is always applied to some arguments. If we η-abstract a function, we obtain an immediate β redex:

$$\begin{array}{lcll} (\lambda \vec{x}_k \to e) & = & (\lambda x \to (\vec{x}_k \to e)\ x) & (\eta) \\ & = & \lambda x\ x_2 \ldots x_k \to e\ [x\ /\ x_1] & (\beta_{\text{var}}) \\ & \equiv & \lambda \vec{x}_k \to e & (\alpha) \end{array}$$

Similarly, η-abstracting an application also yields a (nearly) immediate β redex:

$$
\begin{array}{lcll}
e_0\ e_1 & = & (\lambda x \to e_0\ x)\ e_1 & (\eta) \\
& = & (\lambda x \to e_0\ x)\ (\textbf{let}\ y = e_1\ \textbf{in}\ y) & (\nu) \\
& = & \textbf{let}\ y = e_1\ \textbf{in}\ (\lambda x \to e_0\ x)\ y & (\sigma_m) \\
& = & \textbf{let}\ y = e_1\ \textbf{in}\ e_0\ y & (\beta_{\text{var}}) \\
& = & e_0\ (\textbf{let}\ y = e_1\ \textbf{in}\ y) & (\sigma_m) \\
& = & e_0\ e_1 & (\nu)
\end{array}
$$

These limited readings of η therefore have no practical application on their own. They can, however, be useful when combined with instantiation:

$$
\begin{array}{lcll}
\textbf{let}\ f = e\ \textbf{in}\ f\ e_1 & = & \textbf{let}\ f = e\ \textbf{in}\ (\lambda x \to f\ x)\ e_1 & (\sigma_m,\ \nu,\ \beta_{\text{var}}) \\
& = & \textbf{let}\ f = e\ \textbf{in}\ (\lambda x \to e\ x)\ e_1 & (\iota_e) \\
& = & (\lambda x \to e\ x)\ e_1 & (\epsilon) \\
& = & (\textbf{let}\ f = \lambda x \to e\ x\ \textbf{in}\ f)\ e_1 & (\nu) \\
& = & \textbf{let}\ f = \lambda x \to e\ x\ \textbf{in}\ (f\ e_1) & (\sigma)
\end{array}
$$

In general, we can η-expand a binding if every occurrence of the bound variable is a function application.

3.3.2 Primitives

By their nature, primitives require special reduction rules. We leave the set of primitives open-ended. The main thing to note about the δ axiom schema is that primitives take values as arguments, and in a single step produce a value as a result.

3.3.3 Algebraic types

The **case** construct and constructor application are the fundamental operations for manipulating algebraic types. There is no specific constructor application rule. Calculi such as λ_S distinguish between unevaluated and evaluated constructor applications. In λ_C any constructor application can be treated as "evaluated" if its arguments have been named. At that point, **case** expressions which depend upon it may be discharged (χ_d). When viewing λ_C operationally, we will distinguish evaluated constructor applications by their position in the term.

As noted in Chapter 2, the algebraic types in λ_C actually encompass three concepts at once: Lifting (distinguishing evaluated and unevaluated terms), pairing and projection, and sum types. This results in a large number of conversions on **case** expressions. We examine the conversions for each category separately.

Lifting

A **case** expression can be discharged when its discriminant is a value (χ_d). This is the only rule which erases a **case** expression; consequently, termination of a **case** expression is contingent upon termination of the discriminant, as we would expect. Discharge χ_d represents elimination of lifting—the lifted expression is determined to be non-bottom and thus can be erased.

Any expression may safely be touched in place. The lifting rule (η_l) describes this possibility. If the expression is $\perp$ then the resulting expression will be $\perp$ as well. If the expression can be converted into a value, then the introduced **case** may be discharged.

The bound variable of a **case** expression can be considered to be *unlifted* within the case body. This is not strictly true: the case instantiation rule ι_c permits the discriminant to be instantiated before it has terminated. However, the **case** expression as a whole terminates only if the discriminant does—in which case a copy of the discriminant will terminate as well. This means we can safely discharge inner touches of the bound variable. We state this as a very restricted form of instantiation (ι_l). We imagine *some* value being instantiated, causing the discharge of the inner **case**.

Pairing

The tuple $(x_1,\ x_2)$ corresponds to pairing. Projection is represented by the substitutions performed in the constructor rule χ_c. The corresponding disjunct instantiation rule ι_d represents surjective pairing, the extensional rule for products (akin to η for function spaces). In the **case** framework it is natural to write this rule as an expansion—ι_d does the work of ι_c in advance, in the sense that a disjunct $C^i_k\ \vec{y} \to e$ will only be kept if x has the form $C^i_k\ \vec{e}$.

Sum types

Similarly, we can represent sum types by using the terms *Left* x_1 and *Right* x_2 to represent injection. The rules χ_c and χ_p perform selection. The rule η_a is the extensional axiom for sum, permitting us to expand a **case** expression to explicitly handle all the constructors of a particular type. In this case extensionality effectively *reverses* the action of selection, rather than anticipating it. This reflects the categorical duality of sum and product—the extensional axioms are oriented in a direction where they are naturally used for expansion, rather being oriented according to our concept of **case** reduction, and the direction of expansion for sum is opposite that of product.

$$
\begin{array}{lcll}
I_E[\,] & ::= & \Box & \mid \ \lambda\vec{x}_k \rightarrow I_E[\,] \\
 & \mid & p_k\ E_1 \ldots I_E[\,] \ldots E_k & \mid \ I_E[\,]\ \vec{E} \\
 & \mid & C_k\ E_1 \ldots I_E[\,] \ldots E_k & \mid \ E\ E_1 \ldots I_E[\,] \ldots E_k \\
 & \mid & \textbf{letrec}\ I_B[\,]\ \textbf{in}\ E & \mid \ \textbf{case}\ x = I_E[\,]\ \textbf{of}\ D \\
 & \mid & \textbf{letrec}\ B\ \textbf{in}\ I_E[\,] & \mid \ \textbf{case}\ x = E\ \textbf{of}\ I_D[\,] \\
 & & & \\
I_B[\,] & ::= & x = I_E[\,] & \mid \ I_B[\,];\ B \\
 & & & \\
I_D[\,] & ::= & C_k\ \vec{x} \rightarrow I_E[\,] & \mid \ I_D[\,];\ D \\
 & \mid & \_ \rightarrow I_E[\,] & \mid \ D;\ I_D[\,]
\end{array}
$$

Figure 3-3: Instantiation contexts in λ_C

3.3.4 Binding

Binding is crucial to evaluation in λ_C. Both β_{var} and χ_d require arguments to be named. As reduction proceeds nested block structures must be flattened in order to expose new opportunities for reduction. Most important, variable instances must be instantiated with their definitions.

We noted in Section 2.5 that non-recursive binding blocks (**let**) are distinguished from recursive blocks (**letrec**) primarily for pragmatic reasons. The let conversion rule (ρ) states that **let** can be converted freely into **letrec**; the reverse conversion can be applied if the binding is non-recursive.

Instantiation

All the instantiation rules for λ_C make use of the *instantiation contexts* described in Figure 3-3. An instantiation context describes which occurrences of a variable can be replaced by that variable's definition. In λ_C any variable occurrence which is in scope is a candidate for instantiation. Operational strategies will substantially restrict the scope of instantiation by restricting the instantiation context.

The three instantiation rules for **case** expressions $(\iota_d,\ \iota_c,\ \iota_l)$ have already been discussed in Section 3.3.3. There are also three instantiation rules for **letrec** bindings. This is because there are three parts of a **letrec** expression which might be candidates for instantiation. Occurrences in the result expression may be instantiated by (ι_e). Occurrences in other bindings may be instantiated by (ι_b). Finally, recursive occurrences may be instantiated using (ι_r), in which a definition is substituted within its own body.

Note no instantiation rule is given for **let** expressions. Instantiation of the result is trivial to derive as follows:

$$
\begin{array}{lcll}
\textbf{let } x = \textbf{letrec } b \textbf{ in } e \textbf{ in } I_E[x] & = & \textbf{let } x = \textbf{letrec } b \textbf{ in } e \textbf{ in } & \\
 & & I_E[\textbf{ letrec } b \textbf{ in } e] & \iota_e \\
 & = & I_E[\textbf{ letrec } b \textbf{ in } e] & \epsilon \\
\textbf{let } x = \textbf{letrec } b \textbf{ in } e \textbf{ in } I_E[x] & = & \textbf{letrec } b\,;\ x = e \textbf{ in } I_E[x] & \tau_f \\
 & = & \textbf{letrec } b\,;\ x = e \textbf{ in } I_E[e] & \iota_e \\
 & = & \textbf{letrec } b \textbf{ in } I_E[e] & \epsilon_e \\
\textbf{letrec } b_1\,;\ x = \textbf{letrec } b_2 \textbf{ in } e \textbf{ in } x & = & \textbf{letrec } b_1 \textbf{ in letrec } b_2 \textbf{ in } e & \iota_e \\
\textbf{letrec } b_1\,;\ x = \textbf{letrec } b_2 \textbf{ in } e \textbf{ in } x & = & \textbf{letrec } b_1\,;\ b_2\,;\ x = e \textbf{ in } x & \tau_f \\
 & = & \textbf{letrec } b_1\,;\ b_2\,;\ x = e \textbf{ in } e & \iota_e \\
 & = & \textbf{letrec } b_1\,;\ b_2 \textbf{ in } e & \epsilon_e
\end{array}
$$

Figure 3-4: Derivations for σ_m and τ_m

$$
\begin{array}{lcll}
\textbf{let } x = e \textbf{ in } I_E[x] & = & \textbf{letrec } x = e \textbf{ in } I_E[x] & (\rho) \\
 & = & \textbf{letrec } x = e \textbf{ in } I_E[e] & (\iota_e) \\
 & = & \textbf{let } x = e \textbf{ in } I_E[e] & (\rho)
\end{array}
$$

Naming

Naming is accomplished using the naming rule ν. In Figure 3-2, we give a completely unrestricted form of naming. Naming can be derived by running other conversions backwards:

$$
\begin{array}{lcll}
e & = & \textbf{letrec } \epsilon \textbf{ in } e & (\epsilon_d) \\
 & = & \textbf{letrec } x = e \textbf{ in } e & (\epsilon_e) \\
 & = & \textbf{letrec } x = e \textbf{ in } x & (\iota_e) \\
 & = & \textbf{let } x = e \textbf{ in } x & (\rho)
\end{array}
$$

There are a number of reasons to favor the explicit inclusion of ν in λ_C. If we view the rules as left-to-right reductions, then we must either include ν or permit reversed instantiation (ι_e) and erasure (ϵ). Reverse instantiation permits us to invent terms from thin air—hardly a model of evaluation. Instead, we allow naming of preexisting expressions.

Inclusion of a naming axiom also frees us to place particular operational interpretations on naming, instantiation, and erasure. We would like to view naming as a static process—subexpressions need only be named once—while instantiation is dynamic. Erasure corresponds to Garbage Collection and memory management; reduction should make progress even in its absence.

3.3.5 Structural rules

Repeated instantiation and reduction often gives rise to deeply-nested bindings. These bindings can in turn interfere with evaluation. For this reason, there are a number of *structural rules* which ma-

$$
\begin{array}{rcll}
S[\,] & ::= & \Box & \mid \; S[\,]\,\vec{E} \\
 & \mid & \textbf{let } x = E \textbf{ in } S[\,] & \mid \; p_k\, E_1 \ldots S[\,] \ldots E_k \\
 & \mid & \textbf{letrec } b \textbf{ in } S[\,] & \mid \; \textbf{case } x = S[\,] \textbf{ of } D
\end{array}
$$

Figure 3-5: Strict contexts in λ_C

nipulate binding constructs. The simplest structural rules are the rules for merging nested functions and applications $(\tau_l,\ \tau_a)$ which were discussed in Section 3.3.1.

It must be possible to erase bindings which are no longer being used. We state binding erasure (ϵ_e) as a rule on binding blocks; this permits multiple mutually-recursive bindings to be discarded in a single pass. If every binding in a block can be erased, then the block itself can be eliminated as well (ϵ_d). Erasure allows the elimination of "semantic noise"—bindings left behind by repeated naming and instantiation steps. As noted in Section 3.3.4, erasure can also be used to model the actions of a garbage collector.

In order to perform β_{var} and χ_d reductions, nested bindings must be hoisted out of applications and constructor arguments. The general hoisting rule (σ_m) states that a **letrec** contained in an instantiation context may be hoisted outside that context. Note that σ_m is actually a *derived* rule; it can be described using a mix of instantiation and erasure as shown in Figure 3-4. We include σ_m in our conversion rules for two reasons. First, flattening is a necessary part of straightforward left-to-right reduction. Second, we are uncomfortable with a semantics that gives a central semantic role to erasure, and particularly any reduction strategy that requires erasure. Erasure played just such a central role in some early call-by-need calculi [73].

The **case** construct also introduces bindings. Because **case** requires its discriminant to terminate, these bindings cannot be hoisted freely. They can, however, be hoisted from *strict contexts* (σ_s): the arguments of primitives and the discriminants of **case** expressions (see Figure 3-5). Note that σ_s also allows **case** to be hoisted from the result part of a block; this simply inverts the σ_m rule. Finally, nested touch operations may be freely interchanged (σ_t).

In any **let**-based calculus, **let**-blocks themselves can become highly nested. Such nested blocks can be flattened out; indeed, terms in a **let**-calculus can be viewed as graphs, in which case the exact binding structure does not matter at all [15, 13, 10]. We therefore provide rules $(\tau_f,\ \tau_m)$ to flatten nested blocks. Again, the merge rule τ_m can be derived as shown in Figure 3-4; again, the proof relies on reverse erasure.

3.4 λ_C is badly behaved

The conversions presented in figure 3-2 can be used to justify many common transformations of Haskell programs. However, when viewed as reductions they are not confluent, nor does λ_C possess normal forms. Non-confluence is a common property of **let**-based calculi with the rules ι_b and ι_r—mutually recursive bindings can be instantiated such that they cannot then be brought back together. It was problems with non-confluence that led to graph-based views of **let** expressions; non-confluent reductions change expression structure, but do not change the unfoldings an expression generates.

Absence of normal forms is a consequence of the generality of the ν and η_l rules. It is possible to name or lift any expression:

$$\begin{array}{rcll} e & = & \textbf{let } x = e \textbf{ in } x & (\nu) \\ & = & \textbf{let } x = \textbf{let } y = e \textbf{ in } y \textbf{ in } x & (\nu) \\ & \ldots & & \end{array}$$

These rules can be restricted syntactically, restoring normal forms; we will examine such a restriction in the next section. However, the more general versions are useful for proofs (see Chapter 6).

3.5 Canonical forms of λ_C

The front end of the Eager Haskell compiler translates programs directly into the full λ_C calculus, but in practice it is simpler to manipulate program code if the syntax of the language is restricted in various ways. In this section we examine various canonical program forms and how programs may be transformed into those forms.

3.5.1 Full erasure

The erasure rules $(\epsilon_e,\ \epsilon_d)$, when used in isolation, form a strongly-normalizing reduction system. It is easy to see that erasure terminates: Every erasure step makes a term strictly smaller. To see that erasure is confluent, note erasure cannot make a dead binding live again. Thus, any pair of erasure reductions can be brought back together in a single step.

Note that the erasure rules for λ_C are not safe for a language with side effects or termination detection such as λ_S (and thus *pH*). The rule ϵ_e can freely erase non-terminating computations. In Eager Haskell these non-terminating computations will be garbage collected dynamically if they are left unerased by the compiler.

$$
\begin{array}{lcl}
n\ \vec{E} & \longrightarrow & (\textbf{let}\ x = n\ \textbf{in}\ x)\ \vec{E} \\
E\ E_1 \ldots n \ldots E_k & \longrightarrow & E\ E_1 \ldots (\textbf{let}\ x = n\ \textbf{in}\ x) \ldots E_k \\
p_k\ E_1 \ldots n \ldots E_k & \longrightarrow & p_k\ E_1 \ldots (\textbf{let}\ x = n\ \textbf{in}\ x) \ldots E_k \\
\textbf{case}\ x = n\ \textbf{of}\ D & \longrightarrow & \textbf{case}\ x = (\textbf{let}\ y = n\ \textbf{in}\ y)\ \textbf{of}\ D \\
\textbf{let}\ y = e\ \textbf{in}\ n & \longrightarrow & \textbf{let}\ y = e\ \textbf{in let}\ x = n\ \textbf{in}\ x \\
\textbf{letrec}\ b\ \textbf{in}\ n & \longrightarrow & \textbf{letrec}\ b\ \textbf{in let}\ x = n\ \textbf{in}\ x \\
C_k^i\ \vec{x} \to n & \longrightarrow & C_k^i\ \vec{x} \to \textbf{let}\ x = n\ \textbf{in}\ x \\
\_ \to n & \longrightarrow & \_ \to \textbf{let}\ x = n\ \textbf{in}\ x
\end{array}
$$

$$
\begin{array}{lcllll}
n \in N & ::= & E\ \vec{E}_k & | & \lambda \vec{x}_k \to E \\
 & | & p_k\ \vec{E}_k & | & \textbf{case}\ x = E\ \textbf{of}\ D \\
 & | & C_k\ \vec{E}_k
\end{array}
$$

Figure 3-6: Restricted ν rules for full naming

$$
\begin{array}{lcllll}
L & ::= & \textbf{let}\ x = E\ \textbf{in}\ L & | & x \\
 & | & \textbf{letrec}\ B\ \textbf{in}\ L \\
 & & & & \\
E & ::= & x & | & x\ \vec{x}_k \\
 & | & \lambda \vec{x}_k \to L & | & p_k\ \vec{x}_k \\
 & | & \textbf{case}\ x\ \textbf{of}\ D & | & C_k\ \vec{x}_k \\
 & & & & \\
B & ::= & x = E & | & B\,;\ B \\
 & & & & \\
D & ::= & C_k\ \vec{x}_k \to L & | & D\,;\ D \\
 & | & \_ \to L
\end{array}
$$

Figure 3-7: Fully named form of λ_C

3.5.2 Fully named form

Many compiler phases—particularly those which perform static analysis or code motion—can be expressed more smoothly when every program expression is given a name. For this we can use the naming axiom ν. Because naming is non-normalizing (Section 3.4) we restrict naming contexts as shown in Figure 3-6. We also prohibit the naming of identifiers (where it would be redundant) and **let** and **letrec** expressions. Together these restrictions prevent the ν rule from being applied to any portion of its own right-hand side, thus guaranteeing that naming will terminate. This syntactic restriction also means that naming redexes are disjoint, insuring confluence.

In conjunction with naming, it is useful to move **let** and **letrec** expressions outwards using the σ_m rules. This creates a clear separation between ordinary expressions E and binding blocks L. We can also flatten blocks $(\tau_f,\ \tau_m)$ to simplify block structure. However, when λ_C is used

$$
\begin{array}{lcl}
f\ (\textbf{let}\ x = e_1\ \textbf{in}\ x) & \longrightarrow & \textbf{let}\ x = e_1\ \textbf{in} \\
\quad (\textbf{let}\ y = e_2\ \textbf{in}\ y) & & \quad f\ x\ (\textbf{let}\ y = e_2\ \textbf{in}\ y) \\
 & & \\
 & & \textbf{let}\ x = e_1\ \textbf{in} \\
 & \longrightarrow & \textbf{let}\ y = e_2\ \textbf{in} \\
 & & \quad f\ x\ y \\
 & & \\
f\ (\textbf{let}\ x = e_1\ \textbf{in}\ x) & \longrightarrow & \textbf{let}\ y = e_2\ \textbf{in} \\
\quad (\textbf{let}\ y = e_2\ \textbf{in}\ y) & & \quad f\ (\textbf{let}\ x = e_1\ \textbf{in}\ x)\ y \\
 & & \\
 & & \textbf{let}\ y = e_2\ \textbf{in} \\
 & \longrightarrow & \textbf{let}\ x = e_1\ \textbf{in} \\
 & & \quad f\ x\ y
\end{array}
$$

Figure 3-8: Order of floating affects order of nesting. Here two possible reduction sequences give rise to different results.

as an intermediate representation it is useful to preserve the distinction between **let** and **letrec**. Consequently, we only flatten blocks on the right-hand sides of bindings (τ_f).

We can derive a flattening rule for **let** blocks as follows:

$$
\begin{array}{lcll}
\textbf{let}\ x = (\textbf{let}\ y = e_0\ \textbf{in}\ e_1)\ \textbf{in}\ e_2 & = & \textbf{letrec}\ y = e_0\ ;\ x = e_1\ \textbf{in}\ e_2 & (\rho,\ \tau_f) \\
 & = & \textbf{let}\ y = e_0\ \textbf{in let}\ x = e_1\ \textbf{in}\ e_2 & (\rho,\ \tau_m)
\end{array}
$$

This derived flattening rule has the advantage it does not create a **letrec** from the nested **let** expressions. The syntax after naming, lifting, and flattening can be found in Figure 3-7. We call this *fully named* λ_C.

Full code motion using σ_m is guaranteed to terminate. Every σ_m reduction moves a **let** or **letrec** expression further out; eventually every such block is the result of a **case** disjunct, function, or another block, or is the value of a binding. However, the order in which bindings are floated outwards affects the order in which they end up nested, as shown in Figure 3-8. Floating order is semantically irrelevant, but will affect which expressions are evaluated in an eager strategy; we discuss this further in Chapters 4 and 6. Once code motion is complete, flattening using τ_f (and the derived rule given above) is very simple; each use of τ_f moves exactly one block.

For simplicity, we have described the translation to fully named form a rule at a time: first name (ν), then perform code motion (σ_m), then flatten (τ_f). If the rules are applied in exactly this order, "identity bindings" can be created:

$$
\begin{array}{lcll}
x = \textbf{let}\ y = e_0\ \textbf{in}\ n & \longrightarrow & x = \textbf{let}\ y = e_0\ \textbf{in let}\ z = n\ \textbf{in}\ z & (\nu) \\
 & \longrightarrow & y = e_0\ ;\ x = \textbf{let}\ z = n\ \textbf{in}\ z & (\tau_f) \\
 & \longrightarrow & y = e_0\ ;\ z = n\ ;\ x = z & (\tau_f)
\end{array}
$$

$$
\begin{array}{lcll}
B_E[\,] & ::= & \lambda\vec{x}_k \to B_E[\,] & \\
 & | & p_k\ E_1 \ldots B_E[\,] \ldots E_k & | \quad B_E[\,]\ \vec{E} \\
 & | & C_k\ E_1 \ldots B_E[\,] \ldots E_k & | \quad E\ E_1 \ldots B_E[\,] \ldots E_k \\
 & | & \textbf{letrec}\ B[\,]\ \textbf{in}\ E & | \quad \textbf{case}\ x = B_E[\,]\ \textbf{of}\ D \\
 & | & \textbf{letrec}\ B\ \textbf{in}\ B_E[\,] & | \quad \textbf{case}\ x = E\ \textbf{of}\ B_D[\,] \\
 & & & \\
B[\,] & ::= & \Box & \\
 & | & x = B_E[\,] & | \quad B[\,];\ B \\
 & & & \\
B_D[\,] & ::= & C_k\ \vec{x} \to B_E[\,] & | \quad B_D[\,];\ D \\
 & | & \_ \to B_E[\,] & | \quad D;\ B_D[\,]
\end{array}
$$

Figure 3-9: Binding contexts in λ_C

Here the binding $x = z$ could have been avoided by flattening first:

$$x = \textbf{let}\ y = e_0\ \textbf{in}\ n \quad \longrightarrow \quad y = e_0\,;\ x = n \quad (\tau_f)$$

To avoid introducing identity bindings, we must flatten using τ_f before naming and before each code motion σ_m.

3.5.3 Binding contexts

All the named forms of λ_C have a common feature: They are *binding-centered* rather than *expression-centered*. By naming subexpressions we can manipulate bindings $x = E$ rather than expressions themselves. This observation is useful in proving the correctness of some of the transformations outlined in Chapter 6. When manipulating terms in named form, it is frequently useful to refer to a *binding context* rather than an *expression context*. Binding contexts for the λ_C calculus as a whole are given in Figure 3-9.

3.5.4 Named form

Fully named form gives a name to *every* program expression. This obscures the fact that certain expressions occur in *tail position*: as the result of a function or block, or in a case disjunct. If we restrict naming to instantiation contexts $I_E[n]$ and flatten as before we obtain the *named* form of λ_C. Here arbitrary expressions E are allowed in the result position L described in Figure 3-7. The net result is that only *nested* subexpressions are named and hoisted. Named form is useful for program transformations which do not need to tabulate all the expressions in the program.

$$
\begin{array}{lcllll}
L & ::= & E & | & \textbf{let } x = E \textbf{ in } L & | \quad \textbf{letrec } B \textbf{ in } L \\
E & ::= & C_k\ \vec{x}_k & | & \textbf{case } x = P \textbf{ of } D & \\
 & | & x\ \vec{x}_k & | & \lambda \vec{x}_k \rightarrow L & | \quad P \\
P & ::= & x & | & C_0 & | \quad p_k\ \vec{P}_k \\
B & ::= & x = E & | & B\,;\ B & \\
D & ::= & C_k\ \vec{x}_k \rightarrow L & | & \_ \rightarrow L & | \quad D\,;\ D
\end{array}
$$

Figure 3-10: Argument-named form of λ_C

3.5.5 Argument-named form

An even weaker form of naming is *argument-named form*, shown in Figure 3-10. Unlike named form, argument-named form does not name primitive expressions or constants C_0 in strict contexts $S[p]$. Thus, primitive expressions may occur as arguments to other primitive expressions and in **case** discriminants. To convert to argument-named form we simply restrict naming to instantiation contexts which are not of the form $S[p]$. Motion of the resulting blocks remains unaffected. Argument-named form is used as the basis for code generation in the Eager Haskell compiler.

3.5.6 Flattened form

In Section 3.5.2 it was noted that **let**-motion is not confluent—lifting bindings in a different order results in a different nesting of the binding structure. By merging nested blocks using τ_m confluence can be restored at the cost of losing information about the recursive structure of the bindings. This leads to a fully-flattened form of λ_C. The fully flattened syntax seen during reduction is shown in Figure 3-11

3.6 Reduction of λ_C

Not all of the conversions listed in Figure 3-2 are necessary for reduction of λ_C programs. In the next chapter we will restrict our attention to a greatly restricted subset of λ_C; we call this subset the *dynamic reduction rules*. These rules, shown in Figure 3-12, assume that programs have been converted to argument-named form (as given in Section 3.5.5), and that they have been fully flattened (Section 3.5.6). The reductions in Figure 3-12 differ from the conversions in Figure 3-2 in

$$
\begin{array}{lcllll}
L & ::= & E & | & \textbf{letrec } B \textbf{ in } E & \\
E & ::= & \textbf{case } P \textbf{ of } D & | & F & | \quad P \\
F & ::= & F\ \vec{x}_k & | & \lambda \vec{x}_k \rightarrow L & \\
P & ::= & x & | & V & | \quad p_k\ \vec{P}_k \\
B & ::= & x = L & | & B\,;\ B & \\
D & ::= & C_k\ \vec{x}_k \rightarrow L & | & \_ \rightarrow L & | \quad D\,;\ D
\end{array}
$$

Figure 3-11: Argument-named λ_C during reduction

a few important ways. The remainder of this section examines those differences.

Instantiation is limited to values and variables which reside on the heap; we do not instantiate an expression until it has been completely evaluated. The syntax for terms undergoing evaluation, shown in Figure 3-11, reflects this need and therefore differs slightly from the argument-named syntax of Figure 3-10. Similarly, instantiation only occurs when a variable's value is required for further computation; we therefore restrict ι_b to strict contexts $S[x]$.

The β_{var} rule is restricted to *full arity* applications. This reflects an *eval/apply* approach to curried function application: functions expect a particular number of arguments, and the caller must provide them (possibly with the assistance of the run-time system). We combine naming and lifting in order to split applications at greater than full arity:

$$
\begin{array}{rcll}
(\lambda \vec{y}_k \rightarrow e)\ \vec{x}_k\ x_{k+1} \ldots x_i & = & ((\lambda \vec{y}_k \rightarrow e)\ \vec{x}_k)\ x_{k+1} \ldots x_i & (\tau_a) \\
& = & (\textbf{let } x = (\lambda \vec{y}_k \rightarrow e)\ \vec{x}_k \textbf{ in } x)\ x_{k+1} \ldots x_i & (\nu) \\
& = & \textbf{let } x = (\lambda \vec{y}_k \rightarrow e)\ \vec{x}_k \textbf{ in } x\ x_{k+1} \ldots x_i & (\sigma_m)
\end{array}
$$

When a closure is instantiated into an application, the resulting application must be flattened (τ_a) before it can be further reduced (either by β_{var} or by ν). This corresponds to fetching the arguments from the closure in preparation for performing a call or allocating a larger closure.

The rules for **case** reduction are largely unchanged. The exception is a dynamic optimization of the discharge rule χ_d. When the bound variable is not used, the binding need not be created, as it is immediately subject to erasure:

$$
\begin{array}{rcll}
\textbf{case } \_ = v \textbf{ of } \_ \rightarrow e & = & \textbf{let } \_ = v \textbf{ in } e & (\chi_d) \\
& = & e & (\epsilon)
\end{array}
$$

$x = v\,;\ y = S[x]$	$\longrightarrow$	$x = v\,;\ y = S[v]$	ι_b (instantiate value)
$x = z\,;\ y = S[x]$	$\longrightarrow$	$x = z\,;\ y = S[z]$	ι_b (instantiate variable)
$(\lambda \vec{x}_k \rightarrow e)\ \vec{y}_k$	$\longrightarrow$	$x = e\ [\vec{y}\ /\ \vec{x}]$	β_{var}
$(f\ \vec{x})\ \vec{y}$	$\longrightarrow$	$f\ \vec{x}\ \vec{y}$	τ_a (merge app)
$(\lambda \vec{y}_k \rightarrow e)\ \vec{x}_k\ x_{k+1} \ldots x_i$	$\longrightarrow$	**let** $x = (\lambda \vec{y}_k \rightarrow e)\ \vec{x}_k$ **in** $x\ x_{k+1} \ldots x_i$	ν (split app)
$S[p^i_k\ \vec{v}_k]$	$\longrightarrow$	$S[\underline{p^i_k\ \vec{v}_k}]$	δ axiom schema
case $y = C^i_k\ \vec{e}$ **of** $C^j_l\ \vec{x} \rightarrow e\,;\ d$	$\longrightarrow$	**case** $y = C^i_k\ \vec{e}$ **of** d	χ_p (mismatch)
case $y = C^i_k\ \vec{y}$ **of** $C^i_k\ \vec{x} \rightarrow e\,;\ d$	$\longrightarrow$	**case** $y = C^i_k\ \vec{y}$ **of** $\_ \rightarrow e\ [\vec{y}\ /\ \vec{x}]$	χ_c (constructor)
case $x = v$ **of** $\_ \rightarrow e$	$\longrightarrow$	**let** $x = v$ **in** e	χ_d (discharge)
case $\_ = v$ **of** $\_ \rightarrow e$	$\longrightarrow$	e	
$x =$ **letrec** b **in** e	$\longrightarrow$	$b\,;\ x = e$	τ_f (flatten)
letrec $b\,;\ h\,;\ t$ **in** *main*	$\longrightarrow$	**letrec** $h\,;\ t$ **in** *main* $BV[b] \cap FV[$ **letrec** $h\,;\ t$ **in** *main*$] = \emptyset$	ϵ_e (garbage coll)

Figure 3-12: General dynamic reduction rules for λ_C

This reflects the actual behavior of an implementation. A previously-created value is not copied when scrutinized by a **case** expression. A **case** expression whose discriminant is, say, a boolean value may not create that value explicitly; it may instead be manifest in the control flow of the program itself.

Because function bodies and case disjuncts may contain nested **letrec** blocks, we must still use the flattening rule τ_f. Again, the syntax in Figure 3-11 indicates that the right-hand side of a binding may be a **letrec** during reduction. As we shall see in the next chapter, the flattening rule is the linchpin of our evaluation strategies. The difference between strict, lazy, and eager evaluation is determined to a great extent by how **letrec** blocks are treated by the implementation.

Chapter 4

Evaluation strategies for λ_C

Between call-by-value and lazy strategies there is an enormous space of possible eager strategies. The Eager Haskell implementation is simply one point in this space. In this chapter we characterize the space of eager languages by describing a series of evaluation strategies for the λ_C calculus. The strategies we use are defined by imposing additional structure on λ_C terms. This additional structure evokes the structure of a real machine; the strategies we present here are designed to reflect actual language implementations.

4.1 Overview

A term in the λ_C calculus usually has many possible redexes. A reduction strategy narrows the choice of redexes. The strategies we present in this chapter are under-specified: the strategy may consider more than one redex to be a candidate for reduction. For example, none of the strategies we present completely specifies the order of evaluation for primitive arguments. Ambiguities of this form in a strategy represent places where the implementation has a choice; a particular language implementation resolves these ambiguities in particular ways. We can study the differences between particular implementations by examining how they resolve these ambiguities. In this thesis we refine one strategy—the hybrid lazy/eager strategy—by eliminating ambiguities. The result will be a semantics for Eager Haskell which reflects the choices made in the language implementation.

We begin our presentation by describing some of the evaluation mechanisms used in functional languages, and presenting the corresponding notation we will use in our reduction strategies (Section 4.2). We then present lazy (Section 4.3.1) and strict (Section 4.3.2) strategies for λ_C. We use these well-known strategies to define the notion of an *eager* strategy (Section 4.4), which per-

C	$::=$	**letrec** H $\bullet$ T **in** *main*				Program
H	$::=$	$x = E$	$\mid$	$H,\ H$	$\mid$ ϵ	Heap
T	$::=$	F	$\mid$	$T \parallel T$	$\mid$ ϵ	Threads
R	$::=$	$\langle K \rangle\ R$	$\mid$	$\langle K \rangle$		Stack
K	$::=$	$x = L\,;\ K$			$\mid$ ϵ	Frame

Figure 4-1: Structure of terms during evaluation

forms additional reductions beyond those required by the lazy strategy. A fully eager strategy (Section 4.4.1) captures the entire space of non-strict strategies, but does not lend itself to an efficient implementation.

As a result, we narrow our search to hybrid strategies that mix strict and lazy execution (Section 4.4.2). Ordinarily, execution proceeds just as in a strict language. However, the implementation contains a fallback mechanism to suspend computations in the presence of recursive dependencies. By measuring the resource consumption of the program and initiating fallback when the stack or heap become too full, we achieve normalization 4.7.

4.2 Evaluation mechanisms

In order to capture the evaluation mechanisms used by various strategies, we impose additional structure on the syntax in Figure 3-11. This structure, shown in Figure 4-1, allows us to express our strategies as simple virtual machines. At any time, we can replace the separators , , $\bullet$, $\parallel$ with semicolons ; and rewrite the stack $\langle k \rangle\ r$ to $k\,;\ r$, resulting in an ordinary λ_C term.

4.2.1 Term structure

The main program C is a binding block **letrec** h $\bullet$ t **in** *main* which returns a distinguished value *main*. The special treatment of *main* is implicit in the Haskell language; the top level of a Haskell program is a collection of modules, each of which is a collection of bindings. It is only within this topmost binding block that we use special notation to organize the bindings. It's important to note that the basic structure of λ_C terms has not otherwise changed.

We divide the program bindings into two parts: the heap H and the threads T. The separator $\bullet$ between them was chosen simply to make it easy to distinguish the heap from the threads. The heap H can be thought of as the memory of our abstract machine. It is an unordered collection of bindings separated by commas.

$h \bullet \langle x = v\,;\, k\rangle\ r \parallel t$	$\equiv$	$x = v,\ h \bullet \langle k\rangle\ r \parallel t$	(store)
$x = v,\ h \bullet \langle y = S[x]\,;\, k\rangle\ r \parallel t$	$\longrightarrow$	$x = v,\ h \bullet \langle y = S[v]\,;\, k\rangle\ r \parallel t$	ι_b (fetch)
$x = z,\ h \bullet \langle y = S[x]\,;\, k\rangle\ r \parallel t$	$\longrightarrow$	$x = z,\ h \bullet \langle y = S[z]\,;\, k\rangle\ r \parallel t$	ι_b (indirect)
$\langle x = (\lambda \vec{x}_k \to e)\ \vec{y}_k\,;\, k\rangle\ r$	$\longrightarrow$	$\langle x = e\ [\vec{y}\,/\,\vec{x}]\rangle\ \langle k\rangle\ r$	β_{var} (call)
$\langle x = (\lambda \vec{x}_k \to e)\ \vec{y}_k\rangle\ r$	$\longrightarrow$	$\langle x = e\ [\vec{y}\,/\,\vec{x}]\rangle\ r$	β_{var} (tail call)
$\langle\,\rangle\ r$	$\equiv$	r	(return)
letrec $b\,;\ h \bullet t$ **in** *main*	$\longrightarrow$	**letrec** $h \bullet t$ **in** *main* $BV[b] \cap FV[$ **letrec** $h \bullet t$ **in** *main*$] = \emptyset$	ϵ_e (garbage coll)

Figure 4-2: Structured reduction rules used by every strategy; remaining rules (τ_a, ν, δ, χ) are purely local and identical to those in Figure 3-12.

The threads are an unordered collection of stacks separated by the parallel marker $\parallel$; only one of strategies, the fully eager strategy (Section 4.4.1), actually makes use of multiple threads, but the notation will be used again in Chapter 11 to describe multiprocessor hybrid strategies.

A stack R consists of an ordered list of frames $\langle k\rangle$. The leftmost frame in a stack is the *topmost frame*. A frame is an ordered collection of bindings delimited with angle brackets $\langle\rangle$. The leftmost binding in the topmost frame is the *working term*. In the fully eager strategy (Section 4.4.1), the stack always has a single entry. Similarly, in the lazy strategy (Strategy 4.3.1) there is always exactly one binding per frame.

4.2.2 Starting the program

Initially, the heap H contains all the top level bindings of the program (except the binding for *main*). There is a single thread whose working term is the top-level binding for *main*:

$$\textbf{letrec}\ h \bullet \langle \mathit{main} = e\rangle\ \textbf{in}\ \mathit{main}$$

4.2.3 Evaluation context

To understand how the term structure works, we must examine the mechanisms used in our reduction strategies. All our strategies restrict reduction to the working terms in the program. Only the rules in Figure 4-2 manipulate the structure of the stack and heap. Thus, the local evaluation rules which carry over from Figure 3-12—τ_a, ν, δ, χ—are wrapped in the following context:

$$\textbf{letrec}\ h \bullet \langle x = \Box\,;\, k\rangle\ r \parallel t\ \textbf{in}\ \mathit{main}$$

These rules correspond to local evaluation and local control flow; they will be implemented in a very similar manner regardless of reduction strategy.

4.2.4 Function calls: manipulating the stack

The β_{var} rules and the return rule manipulate the stack, and therefore operate on the stack as a whole:

$$\textbf{letrec } h \ \bullet \ \Box \ \parallel \ t \textbf{ in } main$$

An ordinary full-arity function call pushes a frame:

$$\begin{array}{rl} & \textbf{letrec } h \ \bullet \ \langle x = (\lambda \vec{x}_k \rightarrow e)\ \vec{y}_k\ ;\ k \rangle\ r \ \parallel \ t \textbf{ in } main \\ \longrightarrow & \textbf{letrec } h \ \bullet \ \langle x = e\ [\vec{y}\ /\ \vec{x}] \rangle\ \langle k \rangle\ r \ \parallel \ t \textbf{ in } main \qquad \text{(call)} \end{array}$$

The body of the function is instantiated and pushed onto the stack. When the topmost frame becomes empty, it is popped (erased) and control *returns*:

$$\begin{array}{rl} & \textbf{letrec } h \ \bullet \ \langle\ \rangle\ \langle k \rangle\ r \ \parallel \ t \textbf{ in } main \\ \equiv & \textbf{letrec } h \ \bullet \ \ \langle k \rangle\ r \ \parallel \ t \textbf{ in } main \qquad \text{(return)} \end{array}$$

The bindings in a frame $\langle k \rangle$ represent the flow of control in a function. When we call, the frame is left on the stack, and evaluation resumes with the next binding when we return. Section 5.4 describes the realization of function calls in Eager Haskell.

When a function call is the last binding in a frame, we can perform tail-call optimization, transferring control directly from the current function to the new function without enlarging the stack. This is the purpose of the tail call rule:

$$\begin{array}{rl} & \textbf{letrec } h \ \bullet \ \langle x = (\lambda \vec{x}_k \rightarrow e)\ \vec{y}_k \rangle\ r \ \parallel \ t \textbf{ in } main \\ \longrightarrow & \textbf{letrec } h \ \bullet \ \langle x = e\ [\vec{y}\ /\ \vec{x}] \rangle\ r \ \parallel \ t \textbf{ in } main \qquad \text{(tail call)} \end{array}$$

As noted in Section 3.6, β_{var} is restricted (dynamically) to full arity applications. Partial application of a function will perform allocation (see Section 4.2.7); oversaturated application will invoke the run-time system, which can split the function application in two (split app). Implementation techniques for curried function application are described in Section 5.5.

4.2.5 Results

We do not permit the stack to become completely empty. Instead, the lazy and hybrid strategies provide a special rule (*outermost*) for handling this last stack frame. This rule represents the outermost level of control in the program, which is usually mediated by the run-time system. We consider execution to be complete when *main* has been bound to a value:

$$\textbf{letrec}\ h\ \bullet\ \langle main = v\rangle\ \ \|\ \ t\ \textbf{in}\ main$$

This is the simplest rule governing the outermost frame, and is shared by all the strategies in this chapter. In Section 4.6 we discuss termination rules for multithreaded evaluation.

4.2.6 Deadlock

Under any strategy, if a program reaches a state where there are no redexes, but where *main* has not been bound to a value, then the program has *deadlocked*. We assume programs are well-typed, and deadlock therefore cannot be caused by *e.g.* applying a number to arguments. In the presentation of each strategy we will describe how deadlock can occur and how it can be detected by an implementation. Deadlock is equivalent to divergence ($\perp$).

4.2.7 Storing and fetching values

When the active term binds a value, it cannot be reduced any further. It is removed from the frame and stored onto the heap:

$$\begin{array}{rl} & \textbf{letrec}\ h\ \bullet\ \langle x = v\,;\ k\rangle\ r\ \ \|\ \ t\ \textbf{in}\ main \\ \equiv & \textbf{letrec}\ x = v,\ h\ \bullet\ \langle k\rangle\ r\ \ \|\ \ t\ \textbf{in}\ main \quad \text{(store)} \end{array}$$

Recall that values include constants and constructor applications (both of which are represented in Eager Haskell by tagged heap objects; see Section 5.3) and *closures* (function applications at less than full arity; the Eager Haskell realization of closures is described in Section 5.5 and Section 6.3). In each case the store rule corresponds to allocating space for x and storing v into that space.

Instantiation corresponds to fetching values previously stored to the heap. An actual implementation fetches a value only when it is needed. We therefore restrict fetches to strict contexts in the active term:

$$\begin{array}{rl} & \textbf{letrec}\ x = v,\ h\ \bullet\ \langle y = S[x]\,;\ k\rangle\ r\ \ \|\ \ t\ \textbf{in}\ main \\ \longrightarrow & \textbf{letrec}\ x = v,\ h\ \bullet\ \langle y = S[v]\,;\ k\rangle\ r\ \ \|\ \ t\ \textbf{in}\ main \quad \text{(fetch)} \end{array}$$

In the Eager Haskell implementation, a fetch operation corresponds to fetching the tag of an object on the heap, or fetching the contents of a boxed number; see Section 5.3.

4.2.8 Placing non-values on the heap

Every strategy presented in this chapter except for call-by-value permits non-values to reside on the heap. This has substantial impact on the language implementation: values and non-values must be

represented in a way that allows them to be distinguished. As we will see in Section 4.5, this richer heap structure adds power to the language. It is important to remember that that power has a cost: extra tagging and boxing of values may be required when they are stored, and extra checking will be required before a variable can be fetched. Different tagging methods are compared in Section 5.3.

Bindings on the heap of the form $y = z$ are *indirections*. Indirections are created when a term contains a variable in result position:

$$\begin{array}{rll} & b = C_2\ 7\ b,\ \ i = (\lambda x \to x),\ \ h \bullet \langle y = i\ b \rangle\ r & \\ \longrightarrow & b = C_2\ 7\ b,\ \ i = (\lambda x \to x),\ \ h \bullet \langle y = b \rangle\ r & \text{(tail call)} \\ \equiv & b = C_2\ 7\ b,\ \ i = (\lambda x \to x),\ \ y = b,\ \ h \bullet r & \text{(store indirection)} \\ \longrightarrow & b = C_2\ 7\ b,\ \ i = (\lambda x \to x),\ \ h \bullet \langle y = C_2\ 7\ b \rangle\ r & \text{(fetch)} \\ \equiv & b = C_2\ 7\ b,\ \ i = (\lambda x \to x),\ \ y = C_2\ 7\ b,\ \ h \bullet r & \text{(store)} \end{array}$$

In the first pair of reductions, we allocate an indirection from y to b; in the second we must instead copy the entire structure of b. Indirections represent a substantial space savings when many reductions of this sort occur. Moreover, it is possible for the garbage collector to remove indirections; they therefore represent only a transient space overhead. However, the cost is increased implementation complexity: heap accesses must detect and handle indirections correctly (Section 5.8 explains the handling of indirections in Eager Haskell).

4.2.9 Placing computations on the heap

The lazy and hybrid strategies also place computations in the heap. For example, the hybrid strategies contain the following rule:

$$\begin{array}{rll} & h \bullet \langle x = e\,;\ k \rangle\ r & \\ \equiv & x = e,\ h \bullet \langle k \rangle\ r & \text{(suspend)} \end{array}$$

In practice, e is represented by a data structure containing (at minimum) a code pointer and references to the free variables of e. A computation cannot be loaded; instead, its value must be computed. This is done by pushing it onto the stack:

$$\begin{array}{rll} & x = e,\ h \bullet \langle y = S[x]\,;\ k \rangle\ r & \\ \equiv & h \bullet \langle x = e \rangle\ \langle y = S[x]\,;\ k \rangle\ r & \text{(force)} \end{array}$$

An actual implementation will restore the state saved during suspension, then execute the code to compute e. The binding is not actually *removed* from the heap. Instead we indicate x is being evaluated by overwriting the heap binding with a special "empty" value. When $x = e$ has been computed, the heap binding is overwritten with the resulting value. Control then returns to the

forcing computation; in this respect, forcing $x = e$ is similar to calling the function $x = f\ \vec{x}$. The implementation details of creating and forcing suspensions and thunks are described in Sections 5.6 and 5.7.

4.2.10 Garbage collection

The garbage collection rule ϵ_e deserves a quick mention. We only garbage collect bindings which are found in the heap. One strategy presented in this chapter, the fully eager strategy of Section 4.4.1, keeps unevaluated bindings outside the heap; this specifically prevents them from being garbage collected. Any dead binding in the heap may be subject to garbage collection. The actual garbage collection techniques used in Eager Haskell are described in Section 5.9.

4.3 Reduction strategies

All the strategies we describe have a few common features. First, all reduction occurs in the working term of some thread. Second, no reduction will be permitted inside λ expressions or case disjuncts. Our strategies will be distinguished by a few main features:

- How a nested **letrec** block is flattened.
- When or if a computation can be suspended and copied to the heap.
- When or if a computation can be forced, moving it from the heap back to the stack.
- When or if a new thread must be created.

As we shall see, these elements capture the behavior of strict, lazy, and eager functional language implementations.

4.3.1 A lazy strategy

Any lazy strategy must have two important properties:

- Only bindings whose value is required to obtain the value of the root should be reduced.
- It must *preserve sharing*—bindings outside λ expressions must be reduced at most once (note that this reduction is not *optimal*; redexes inside λ are duplicated freely when instantiation occurs).

$h \bullet \langle x = \textbf{letrec } b \textbf{ in } e_1 \rangle\ r$	$\longrightarrow$	$b,\ h \bullet \langle x = e_1 \rangle\ r$	τ_f (allocate)
$x = e,\ h \bullet \langle y = S[x] \rangle\ r$	$\equiv$	$h \bullet \langle x = e \rangle\ \langle y = S[x] \rangle\ r$ $e \notin var \wedge e \notin V$	(force)
$h \bullet \langle x = y \rangle\ r$	$\equiv$	$x = y,\ h \bullet r$	(store indirection)
$y = e,\ h \bullet \langle x = y \rangle$	$\equiv$	$x = y,\ h \bullet \langle y = e \rangle$	(outermost)

Figure 4-3: Reduction rules for lazy strategy (See also Figures 4-2 and 3-12)

Rules for a lazy strategy are given in Figure 4-3. When a **letrec** block is encountered, the bindings are allocated (τ_f) as suspended computations (thunks) on the heap (in our notation the semicolons separating bindings must also be replaced by commas). When a variable is required, its value is *demanded*; if the value is not yet available, it must be forced. Every frame contains a single binding; as a result all function calls are tail calls, and only forcing a variable will cause the stack to grow.

A special case occurs when the outermost binding is rewritten to an indirection. In this case, the rewritten binding replaces the indirection on the stack (store and force). This guarantees that reduction will continue until the outermost binding has actually been fully evaluated.

Deadlock occurs in the lazy strategy when a variable which resides on the stack is a candidate for instantiation. For example:

$$\begin{array}{rl} & y = 4 * x,\ h \bullet \langle x = y + 1 \rangle\ r \\ \longrightarrow & h \bullet \langle y = 4 * x \rangle\ \langle x = y + 1 \rangle\ r \quad \text{(force)} \\ & \textit{deadlock} \end{array}$$

We noted in Section 4.2.9 that a binding is overwritten with a special value when it is forced. In lazy languages, this special value is traditionally known as a "black hole". The implementation detects deadlock by noticing that x is a black hole as we attempt to evaluate y.

4.3.2 A strict strategy

The strict strategy has three important characteristics:

- Only values may be stored in the heap.
- Bindings are executed in the order in which they occur in the program text.
- All variable references in the active term must refer to values on the heap.

$h \bullet \langle x = \textbf{letrec } b \textbf{ in } e_1 \,;\, k\rangle\, r$	$\longrightarrow$	$h \bullet \langle b \,;\, x = e_1 \,;\, k\rangle\, r$	τ_f (enter block)

Figure 4-4: Additional reduction rule for strict strategy (See also Figures 4-2 and 3-12)

These restrictions on evaluation permit the strict strategy to be implemented very efficiently. A variable reference is assumed to refer to a value—no checking is required to verify that this is the case. Contrast this with the lazy strategy in the previous section: a variable may refer to a value, a thunk, an indirection, or a black hole. None of the extra mechanisms for suspending and resuming computations need to exist in a strict language.

As a result, the strict strategy for λ_C given in Figure 4-4 is very simple—just one rule for flattening **letrec** blocks so that bindings are evaluated in program order. The drawback to this strategy is that we must restrict λ_C itself: The only recursive references permitted in **letrec** blocks are between functions. It is invalid to reference a binding which has not yet been evaluated.

Note that while the strict strategy executes a given program in order, the compiler still has a great deal of liberty in ordering program bindings. The process of canonicalization itself, which transformed an arbitrary λ_C program into argument-named form, is not fully specified. By varying the order in which function arguments are named and lifted, for example, different execution orders will result. Furthermore, the compiler is always free to reorder bindings when the results of doing so cannot be observed.

4.4 Eagerness

We call a strategy *eager* if there are infinitely many programs for which the strategy performs β_{var}, δ, or χ_d reductions which would not have been performed by the lazy strategy. Some eager strategies are trivial: the compiler for a lazy language may introduce eagerness statically when analysis shows that it is safe to do so [36]. In this chapter, we focus on strategies which permit *unlimited* eagerness—that is, the use of eager evaluation is not constrained to expressions with particular static properties. The call-by-value strategy is clearly eager: a **letrec**-bound variable is evaluated even if it is never referenced. In the remainder of this chapter we examine several eager strategies for non-strict languages. In Chapter 8, we will introduce simple static constraints to handle dynamic error conditions. By default, however, expressions will continue to be eagerly evaluated.

$h \bullet x = \textbf{letrec } b \textbf{ in } e_1 \parallel t$	$\longrightarrow$	$h \bullet x = e_1 \parallel b \parallel t$	τ_f (spawn)
$h \bullet x = v \parallel t$	$\equiv$	$x = v,\ h \bullet t$	(store value)
$h \bullet x = y \parallel t$	$\equiv$	$x = y,\ h \bullet t$	(store indirection)

Figure 4-5: A fully eager strategy (See also Figures 4-2 and 3-12)

4.4.1 A fully eager strategy

In order to construct a maximally eager strategy, we should attempt to compute every single binding in the program eagerly. We can do so by running these bindings in parallel. Every binding is evaluated in a separate thread. Every call will therefore be a tail call, and there is consequently no need for a stack or continuations. This *fully eager* strategy is captured in Figure 4-5. The eagerness of the model can be seen by comparing the spawn rule to the enter block rule in the strict strategy (Figure 4-4). Both rules immediately begin evaluating the bindings of the block; the strict strategy does so one at a time, whereas the lazy strategy evaluates every binding in parallel.

The fully eager strategy is extremely general—we can at any time choose to evaluate any thread or threads. This freedom of choice means that we can, for example, simulate lazy evaluation simply by keeping track of the "currently needed" thread and focusing all effort on evaluating that thread. Indeed, we can simulate any non-strict strategy by choosing which bindings are actually evaluated.

The generality of the fully eager strategy is also its downfall. Note that any thread in the system may become *blocked*. This happens when a variable requires instantiation, but the binding for that variable is still being computed in another thread. No reductions on that thread can be performed. An actual implementation could quickly become swamped by partially-completed computations which are awaiting results. A reasonable implementation must distinguish between threads which are blocked and threads which are not blocked.

The fully eager strategy has another failing: it breaks computations up into tiny, short-lived threads. Thread state must be allocated and tracked dynamically. Switching to an arbitrary thread requires arbitrary control flow. The combination of dynamic resource usage and unconstrained control flow are a disaster for a modern architecture, where temporal and spatial locality are vital to efficient execution. An efficient eager strategy must impose structure on the threads so that there is a clear way to manage control flow and resource allocation.

$h \bullet \langle x = \textbf{letrec } b \textbf{ in } e_1 \,;\, k\rangle\ r$	$\longrightarrow$	$h \bullet \langle b \,;\, x = e_1 \,;\, k\rangle\ r$	τ_f (enter block)
$h \bullet \langle x = e \,;\, k\rangle\ r$	$\equiv$	$x = e,\ h \bullet \langle k\rangle\ r$	(suspend)
$h \bullet \langle x = y \,;\, k\rangle\ r$	$\equiv$	$x = y,\ h \bullet \langle k\rangle\ r$	(store indirection)
$x = e,\ h \bullet \langle y = S[x] \,;\, k\rangle\ r$	$\equiv$	$h \bullet \langle x = e\rangle\ \langle y = S[x] \,;\, k\rangle\ r$ $e \notin \mathit{var} \wedge e \notin V$	(force)
$y = e,\ h \bullet \langle x = y\rangle$	$\equiv$	$x = y,\ h \bullet \langle y = e\rangle$	(outermost)

Figure 4-6: Hybrid eager and lazy strategy (Compare Figures 4-3 and 4-4)

4.4.2 The hybrid strategy

Both strict and lazy languages solve this problem in a similar fashion: A stack of frames is used to group together related bindings. A stack provides a systematic and well-understood way to manage both control flow and local storage. It is therefore worthwhile to seek a stack-based eager strategy. (The actual implementation of the stack is described in Section 5.2.3.)

One natural course is to combine elements of the strategies we understand well—the lazy strategy and the strict strategy. This results in the hybrid strategy found in Figure 4-6. Most of the rules are identical to rules for either the lazy or the strict calculus. Bindings are started in program order, so the enter rule is identical to the rule in the strict strategy (Figure 4-4). Demand-driven evaluation (evaluate) works just as in the lazy strategy (Figure 4-3). However, if no suspension exists on the heap for variable y in $x = S[y]$ then y is pending and resides somewhere in the stack. In this case we must create a suspension for x on the heap and continue executing the work on the stack (the implementation of suspension is described in Section 5.6).

We might expect the suspension rule to require that the variable y occur in a strict context $S[y]$. Doing so would yield a "minimally lazy" hybrid strategy—suspension would occur only when absolutely required by non-strictness. Unfortunately, this makes our strategy sensitive to the order of bindings in the program text; we examine this inconsistency in the next section.

The suspend rule for Eager Haskell contains no such restriction. Indeed, the active term may be suspended at *any* time for *any* reason, even if it is possible to evaluate it immediately. This extra flexibility captures a large and interesting range of implementation choices. For example, by immediately suspending the bindings b upon entering a block, we obtain exactly the effect of the allocate rule in the lazy strategy.

4.5 How strategies treat the heap

One crucial difference between the strategies discussed thus far is their treatment of the heap. Consider an instantiation context $x = S[y]$. Under the strict strategy, y has been computed and *must* reside somewhere on the heap (see Section 4.3.2). Under the fully eager strategy, uncomputed values lie *outside* the heap in the thread pool (Figure 4-5). Under the hybrid and lazy strategies, uncomputed values may reside on the stack (completely empty data structures; see Section 5.3) *or* in the heap.

The ability to place uncomputed bindings on the heap adds power to the language; the hybrid and lazy strategies will successfully execute some programs which do not terminate under the fully eager strategy. Consider the following example:

```
letrec forever x     = case x of _ → forever x
       const x y     = x
       y             = forever z
       z             = const 5 y
in z
```

In the strict strategy, we cannot execute this program at all—it contains the mutually-recursive value bindings y and z. In the lazy strategy, we evaluate z, which calls *const*. Since *const* ignores its second argument, it does not matter that y refers to a non-terminating computation—z is reduced to 5 and the binding for y can be garbage collected.

In the fully eager strategy, we attempt to evaluate the binding for y. This evaluation blocks or suspends because *forever* requires the value of z. The binding z is computed and discards y. At this point, the binding for y still exists as an independent thread. This thread can be run forever; more important, its execution resources can never be reclaimed. In contrast, the hybrid strategy creates a suspension for y on the heap. This suspension is ignored by the call to *const*, and can simply be garbage collected.

Under the hybrid strategy, there is a choice when evaluating this example. When the binding for y is encountered, it may be suspended; in this case the binding for z will eventually be evaluated and y discarded exactly as with the lazy strategy. If, however, the binding for y is run eagerly, then execution will suspend because the value for z is required. Again, z will be run, this time discarding the newly-created suspension. In either case, the net effect is the same: Execution of y is abandoned, and the storage required for the suspension can be reclaimed once z has discarded it.

However, consider what happens under the hybrid strategy if we reverse the order of y and

z in the above example. If *z* is evaluated first, it yields 5. The strategy may then choose to run *y* forever without suspending: simply reversing the order of two bindings has changed the termination behavior of the program. The general suspension rule permits us to suspend and eventually garbage collect the already-discarded computation of *y*. However, the strategy of Figure 4-6 does not give a *policy* for applying this rule.

4.6 Other Eager Strategies

The eager strategies implemented in Id and *pH* do not display the sensitivity to evaluation order which is possible in the hybrid strategy. This is because they define program termination in a stronger way (see Section 4.2.5). We consider a hybrid program to have terminated when we obtain a value for *main*. Both Id and *pH* require that *every* computation terminate. This condition is easy to express in the fully eager strategy: A program terminates when there are no longer any threads to be run. In the example, the binding for *y must* eventually be run, and as a result the program will *never* terminate.

Naturally, the evaluation strategies used in Id and *pH* make their own set of tradeoffs in the name of efficiency. Briefly, every program binding has an associated *location*, which is either empty or full. Empty locations correspond to the bindings in the thread pool under the fully eager strategy. An empty location has an associated *defer list* which lists bindings which have suspended awaiting the location's value. When a location becomes full, these bindings are re-started. In this way, threads which certainly cannot make progress are distinguished from those which may potentially contain useful work. A formalization of the defer-list strategy can be found in Appendix A; it requires additional notation to describe the structure of defer lists.

The hybrid approach is demand-driven, whereas the defer list approach is producer-driven. Both techniques have advantages and drawbacks. The defer list approach keeps computations alive (on defer lists) even if their results are never required. The problem of scheduling in the presence of defer lists is in general a murky one; when a defer list is re-started, it is unclear when it is appropriate to run the newly re-started computations. However, there will never be any attempt to run a suspended computation.

The hybrid strategy, on the other hand, can immediately discard the resources associated with useless computations. There is no question when to schedule formerly suspended computations—they should be run when their values are demanded. However, it is possible to re-start a suspension

$h \bullet \langle x = S[y]\,;\ k_0\,;\ y = e\,;\ k_1\rangle\ r$	$\equiv$	$x = S[y],\ h \bullet \langle k_0\,;\ y = e\,;\ k_1\rangle\ r$	(suspend)
$h \bullet \langle x = S[y]\,;\ k_0\rangle\ r_0\ \langle b\,;\ y = e\,;\ k_1\rangle\ r_1$	$\equiv$	$x = S[y],\ h \bullet \langle k_0\rangle\ r_0\ \langle b\,;\ y = e\,;\ k_1\rangle\ r_1$	(suspend)
$h \bullet \underline{\langle x = f\ \vec{x};\ k\rangle\ r}$	$\equiv$	$x = f\ \vec{x},\ h \bullet \underline{\langle k\rangle\ r}$	(thunk)
$h \bullet \underline{\langle x = f\ \vec{x}\rangle\ r}$	$\equiv$	$x = f\ \vec{x},\ h \bullet \underline{r}$	(tail thunk)
$y = e,\ h \bullet \underline{\langle x = S[y]\,;\ k\rangle\ r}$	$\equiv$	$y = e,\ x = S[y],\ h \bullet \underline{\langle k\rangle\ r}$ $e \notin \mathit{var} \wedge e \notin V$	(no-force)
$h \bullet \langle x = f\ \vec{x};\ k\rangle\ r$	$\equiv$	$h \bullet \underline{\langle x = f\ \vec{x};\ k\rangle\ r}$ $\lvert r\rvert > \mathit{stack}_{max}$	(call exception)
$y = e,\ h \bullet \langle x = S[y]\,;\ k\rangle\ r$	$\equiv$	$y = e,\ h \bullet \underline{\langle x = S[y]\,;\ k\rangle\ r}$ $\lvert r\rvert > \mathit{stack}_{max} \wedge e \notin \mathit{var} \wedge e \notin V$	(force ex.)
$h \bullet \langle x = v\,;\ k\rangle\ r$	$\equiv$	$x = v,\ h \bullet \underline{\langle k\rangle\ r}$ $e(\lvert h\rvert, \lvert r\rvert) > \mathit{resource}_{max}$	(heap ex.)
$h \bullet \underline{\langle x = e\rangle}$	$\equiv$	$h \bullet \langle x = e\rangle$ $\mathit{resource}_{max} \leftarrow f(\lvert h\rvert, \mathit{resource}_{max})$	(outer end)
letrec $b,\ h \bullet r$ **in** *main*	$\longrightarrow$	**letrec** $h \bullet r$ **in** *main* $BV[b] \cap FV[$ **letrec** $h \bullet r$ **in** *main*$] = \emptyset$ $\mathit{resource}_{max} \leftarrow g(\lvert b\rvert, \lvert h\rvert, \mathit{resource}_{max})$	ϵ_e (gc)

Figure 4-7: Reduction in the presence of exceptions. Underlines indicate fallback is in progress. Compare to Figure 4-6.

(on demand) only to immediately discover that a value upon which it depends remains unavailable.

4.7 Resource-bounded Computation

Any eager strategy presenting the same language semantics as lazy evaluation must stop the execution of runaway computations such as *forever* from Section 4.5. Efficiency demands that these computations be cut short before they use an inordinate amount of time and space. This leads us naturally to the idea of *resource-bounded computation*: limit the amount of time and space which can be used by a computation, and use suspension to fall back and shut down computations when those resource bounds are exceeded.

We view the fallback process as a form of exception mechanism. This idea is formalized in Figure 4-7. When multiple rules apply to a given term, the rules in Figure 4-7 take precedence over those in Figure 4-2. Ordinarily, computation proceeds eagerly; computations suspend only

when a required variable resides on the heap (due to non-strictness). This is expressed as two rules (suspend)—one when the relevant variable is bound in the same stack frame, the second when the variable is bound deeper in the stack.

When resource bounds are reached, an exception is signaled, and *fallback* begins, indicated by underlining the stack. During fallback, we disallow stack growth. This means that subsequent function calls must suspend (the thunk rules) and that computations which would usually require evaluation of a heap value must themselves suspend rather than forcing the computation on the heap (the no-force rule).

In order to bound the time and space used by a computation, we must check resource bounds in three places (the actual implementation of this policy is described in Section 5.9.4). First, the amount of stack which is used must be bounded; thus, we check stack usage at every function call and every evaluate. If usage exceeds the bound on stack growth $stack_{max}$ then an exception is signaled. Second, the total space usage must be bounded. This is checked at every allocation point; the monotonic function e adjusts for the fact that heap and stack usage may be accounted for in different ways.

Finally, we must bound the total time used by any computation. These rules do not measure time directly. Instead, we note that the heap grows steadily as evaluation progresses. Thus, space usage and time usage are closely correlated. The only exception to this is when garbage collection occurs: here the heap may shrink once again. Thus, we compute a new resource bound $resource_{max}$ each time we garbage collect. The bound is a function g of the current bound, the current space usage, and the amount of garbage collected.

Exceptional execution guarantees that the stack must shrink. When it is empty exceptional execution ends, and the program resumes ordinary eager execution once more. At this point the resource bounds are reset based on the resources currently in use by the suspended program. Note that suspended computations are re-started in a demand-driven fashion: the exception ends with a single computation on the stack, and the forcing mechanisms is used to evaluate suspensions as they are required.

Chapter 5

Run-time Structure

Data representation is vital to efficient compilation of a non-strict language. This chapter examines some of the tradeoffs inherent in various data representation choices. The tagged data representation used by the Eager Haskell compiler is described, and its advantages and potential drawbacks are noted. The chosen structure leads to a natural division of responsibility between compiled code and the Eager Haskell language runtime. This division is outlined, and the structure of the run-time system is described.

5.1 Overview

Our choice of data representation is affected by several factors. Recall from Section 4.2.9 that non-strict languages must distinguish *values* from *computations*. Polymorphically-typed languages must represent data in a uniform manner when it is used in a polymorphic context [64] unless they can eliminate polymorphism statically [129, 82]. In a higher-order language we must also represent function closures and curried partial applications in some fashion. In addition, any language which supports *precise* garbage collection must provide a way for the garbage collector to identify heap references, and must establish some invariants on heap usage. Finally, if we wish to run our programs in parallel we must establish the conventions by which multiple threads may access the heap. All these choices constrain our data representation and thus influence the structure of the code we generate.

In this chapter we review data representation strategies used elsewhere and present the strategy used by the Eager Haskell compiler. We use a tagged, boxed representation for all program data. Tags are represented as simple integer values rather than pointers to descriptors in order to simplify

the primary control flow of the program.

5.2 Driving assumptions

An modern architecture is optimized to make particular coding idioms run extremely fast. A language implementation should, whenever possible, generate code with these idioms in mind. Similarly, compilers are often designed to ensure that particular source-language idioms produce efficient code. We begin by outlining the assumptions we make about modern architectures and about the behavior of Eager Haskell programs; the remainder of the chapter describes in detail how these assumptions will guide our implementation.

5.2.1 Architectures reward locality

Modern architectures provide a multilevel memory hierarchy which rewards applications which exhibit temporal and spatial locality. We assume the existence of multiple levels of caches, including separate instruction and data caches for the most recently-accessed portions of memory. At higher levels of the memory hierarchy, a program with a large address space will incur overhead for misses in the translation lookaside buffer (TLB) when translating virtual addresses to physical addresses. When address spaces become very large, portions of memory will be paged to disk. For such large address spaces, a program with poor locality will quickly slow to a crawl.

5.2.2 Branches should be predictable

In order to optimize control flow for the common case, we take advantage of the features of modern processors, particularly branch prediction. We assume that indirect branches to unknown code are unpredictable, and thus expensive. For this reason we avoid using function pointers as part of ordinary control flow.

Similarly, we use branch prediction hints (provided by recent versions of gcc) to indicate that particular execution paths are common or rare. This also causes the C compiler to segregate rarely-executed basic blocks from the main control flow of a function. This improves the instruction cache performance of our programs.

5.2.3 Compiling to C will produce better code

The Eager Haskell compiler generates gcc code. The techniques and tradeoffs involved in compiling high-level languages to C have been examined in great detail elsewhere [31, 93, 42, 48, 24, 65]. The most compelling argument for compiling via gcc is the maturity and portability of the compiler. In order to generate high-quality machine code, a compiler must incorporate an instruction scheduler, a register allocator, and a good-quality code generator and peephole optimizer. It requires an enormous amount of effort to match gcc even on a single architecture. For example, the most mature optimizing compiler for Haskell, GHC, includes a native code generator for Intel machines. However, GHC generates faster code (at the cost of extra compile time) by compiling via gcc.

We choose gcc rather than another C compiler for two reasons. First, gcc is available on every popular machine architecture. Second, gcc provides support for a number of language extensions which make the task of mapping Haskell to C much easier. Many compilers make use of gcc's ability to map global variables to machine registers [31, 93, 48]. Recent versions of gcc can compile arbitrary tail recursion; previous compilers often resorted to complex trickery to avoid running out of stack [93, 31]. Finally, the provision of branch prediction annotations makes it much easier for the Eager Haskell compiler to express and exploit the assumptions made in the previous section.

Compiling to C rather than generating machine code does place a burden on the run-time system. An allocation-intensive language such as Eager Haskell requires a precise garbage collector, and must therefore maintain a *shadow stack* containing the live pointers in a computation. This shadow stack is treated as a root by the garbage collector. This increases overhead in two ways. First, an additional machine register is required to maintain the shadow stack pointer. Second, the compiler must explicitly generate code to save and restore shadow stack entries. The C register allocator is also making decisions about which variables will reside in registers and which must be kept on the stack. Inevitably these decisions do not coincide and extra memory operations result.

5.2.4 Non-strictness is rare

The most important assumption made by the Eager Haskell compiler is that non-strictness is rarely used even in non-strict programs. Most of the time programs can be run eagerly, in exactly the order given, and without requiring suspension. This assumption is certainly true of Id programs [116], and motivated Shaw's stripped down parallel implementation of a subset of Id [120]. Occasionally non-strict semantics will actually require suspension; occasionally resource bounds will be reached

and execution will suspend. However, we focus our energies on making ordinary, non-suspensive execution as fast as possible while still permitting non-strictness and suspension to occur.

5.2.5 Values are common

A corollary to the presumed rarity of suspension is a second assumption: most of the time an Eager Haskell program manipulates values. It must be easy and fast to distinguish a value from a non-value, and tests which make this distinction should be biased to favor values. When a suspended computation or an indirection is encountered, the implementation should take pains to make sure future computations can use the desired value directly.

In order to optimize control flow for the common case, we use branch prediction hints to indicate that suspension is rare. We avoid the use of indirect branches when checking whether a particular object has been evaluated. This stands in stark contrast to both GHC and hbc; in both cases objects are tagged with a function pointer which, when called, evaluates the object. As a result, every single **case** statement (at least in GHC) involves an indirect branch—even if the data involved has already been evaluated.

The GRIN project makes the same set of architectural assumptions, but takes a very different approach to compiling lazy languages [55]. Constructors and thunks are treated uniformly as integer tags. Whole-program control flow analysis reveals which tags reach particular *eval* instructions (equivalent to **case** expressions in Eager Haskell), and these are checked for explicitly. Boquist then uses the same whole-program analysis to perform interprocedural register allocation [29]. However, the techniques used in GRIN require whole-program compilation and a native code generator; we ruled out both approaches in designing the Eager Haskell compiler.

5.3 Tagged data

In order to support non-strict execution, all data in Eager Haskell is *tagged* and *boxed*. Boxing means that all data—even primitive values such as characters and floating-point numbers—is allocated on the heap. This has a measurable runtime cost—the cost of allocating additional storage for primitive data, initializing it, and garbage collecting it, and the cost of fetching primitive data from memory when it is required. Tagging means that all heap data is prefixed by a tag, which in Eager Haskell is a pair *MkTag* (*ident*, *size*). The size is used by the garbage collector. The *ident* distinguishes the following things:

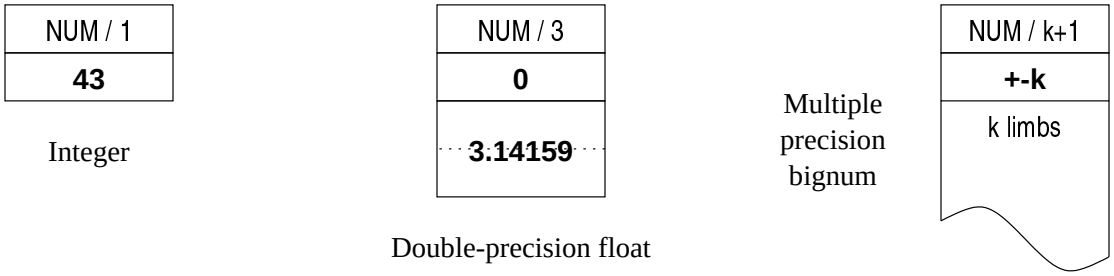


Figure 5-1: Boxed representation of numbers in Eager Haskell. Note that *Double*s have a word of padding after the tag so that the data will be doubleword-aligned.

- The various disjuncts of an algebraic data type (which contain pointers).
- Function closures and partial applications (Section 5.4).
- Values containing non-pointer data (*Int*, *Double*, *Integer*), shown in Figure 5-1.
- Indirections (Section 5.8).
- Barrier indirections used for enforcing heap invariants (Section 5.9.2).
- Completely empty data structures, which are currently under computation; they will eventually be filled in with a value or a suspension. This is the representation used for computations which reside in the stack in our strategies (Section 4.2.9).
- Thunks: suspended function calls due to an exception (Section 5.7).
- Suspensions: **case** expressions whose data was unavailable (Section 5.6).

We order the identifiers so that values are positive and non-values (including indirections) are negative. The compiled code deals only with values; all non-values are handled by the run-time system.

Tagged, boxed memory is not the only possible data representation in a non-strict language. Some hardware, such as the Monsoon dataflow machine (the primary target of the Id compiler), provides support for type-tagged memory. This can be exploited to distinguish pointers from non-pointers, or to distinguish empty objects and values. A closely related technique is to tag the data itself. This technique has long been a staple of Lisp systems, and is used by emacs [68], Caml Light [65, 118], and gofer and hugs [58]. Typically, particular settings of the low-order bits of a machine word incorporate limited type information, at the minimum indicating whether the word should be interpreted as an integer or a pointer. The current implementation of *pH* uses an interesting

variation of this technique, in which pointers are encoded as valid double-precision IEEE NaNs (not a numbers) [31]. This provides a contiguous integer range and allows the use of unboxed floating-point numbers; it also yields a large enough address range to allow pointers to be accompanied by type information. However, the *pH* object representation requires a 64-bit machine architecture in order realize full performance, and the memory consumption of pointer-intensive programs is doubled.

Using tagged values rather than tagged memory has certain attractions. Both tagging techniques impose overhead to tag and untag data; however, shifting and masking value tags can be done in a machine register; tagged memory requires additional memory operations. If we use tagged values, small integers and nullary constructors—both very common—need not be stored in separate, tagged memory locations. The Eager Haskell garbage collector mitigates this cost by redirecting references to nullary constructor and small integers so that they point to a fixed table.

The biggest drawback to using tagged values is that they usually require tagged memory as well. There simply aren't enough free bits in a pointer word to distinguish all possible object sizes and constructor tags that might exist within a single algebraic datatype. As a result, most pointers refer to tagged memory (Lisp systems typically reserve a special pointer tag for cons cells so that this common case does not require tagging). At the same time, tagging techniques impose additional constraints on the ranges of values. This can prove especially difficult when interacting with libraries that assume (for example) that 32-bit integers are available.

In a system using integer tags, object sizes and layouts are limited by the way information is encoded in the tag. Instead of tagging objects with a simple integer, we can tag them with a pointer to a *descriptor*. A descriptor can be shared by many objects, and is usually generated statically. As a result, the descriptor can be much larger than one or two machine words. Using a descriptor permits pointer and non-pointer data to be commingled in essentially arbitrary fashion.

Some descriptors—most notably those used in GHC and in hbc—can be thought of as *active tags*. One entry of the descriptor table is a function; in GHC and hbc this is the function which forces a thunk. Thus, forcing a thunk is a matter of entering the code stored in the descriptor. This technique can be extended to other portions of the system. For example, including a pointer to a garbage collection routine in every descriptor makes it easy to use unusual garbage collection techniques for particular heap objects [93, 140].

In a strongly typed language, it is possible to dispense with tagging entirely [5, 4]; if the garbage collector knows the type of every root, then it is simple to determine the types of all reachable

objects. However, callers must pass type information whenever a polymorphic function is invoked in order to determine the types of objects referenced from the stack. Moreover, in a language with algebraic data types tags are still required to distinguish the different constructors in a type. As a result, type-based garbage collection is seldom worthwhile.

On a system with a large address space, it is often possible to use a BiBoP (big bag of pages) allocator to segregate objects with different tags [144]. Memory is divided into fixed-size chunks (often one or more virtual memory pages in size); each chunk contains objects of a single size or with a single tag. The data structure used to manage chunks contains appropriate tagging information. We reject this approach in Eager Haskell for several reasons. First, an object can have many tags over its lifespan due to suspension; this would require a cheap method for migrating objects between pages. Second, Eager Haskell is extremely allocation-intensive; as a result, allocation must be cheap. The BiBoP technique requires separate allocation state for every possible tag used by the program.

The BiBoP technique *is* a good method for structuring a high-level allocator such as the shared multigenerational heap used in Eager Haskell. Here object tags are preserved, and the allocator segregates *some* objects simply for convenience. For example, by segregating objects of similar size a mark-sweep allocator can use a single bitmap for marking and allocation [28]. Similarly, pointer-free objects can be segregated from objects which must be traced by the collector, reducing page faults during collection [60].

5.4 Function structure

Having decided to compile Eager Haskell programs to C, another fundamental decision must be made: How to map Haskell functions to C functions. This is a tricky decision for any language with a substantially different control structure from C itself.

By choosing to compile Eager Haskell to C, we are obliged to have idiomatic Haskell programs compile to idiomatic C whenever possible. For example, we treat nested primitive expressions as single units for the purpose of code generation (see Section 3.5.5), allowing the C compiler to generate the best possible code for them. We therefore map each Eager Haskell function to a single C function. We *avoid* turning individual Haskell bindings into functions because we subvert the C compiler's ability to do register allocation, branch prediction and the like.

We also assume that larger functions are (within reason) better. Haskell functions tend to be very

small; C functions are generally much larger. As we note in Section 6.4, truly enormous functions strain the resources of the C compiler, and the Eager Haskell compiler takes steps to break such functions into smaller pieces at logical boundaries in the control flow.

There are a number of calling conventions that may be adopted. The shadow stack can be maintained either as a separate data structure, or each function can have a local array of shadow stack entries which are then linked together, embedding the shadow stack in the C stack. Eager Haskell function arguments can be passed either as C arguments or they can be pushed on the shadow stack. Simple tests confirm that maintaining a separate shadow stack is substantially more efficient. Surprisingly, even in the absence of a garbage collector using the shadow stack for parameter passing was approximately as efficient as using the C calling conventions. Garbage collection requires spilling arguments to the shadow stack and further shifts the balance.

When code suspends, we must somehow package up the point in the function body where execution should resume. We take our cue from Cilk [37], and give every function a set of numbered *entry points*. The entry point is passed as an argument to the function. Entry point zero is the distinguished entry point indicating the beginning of the function. If the entry point is nonzero, we perform an indexed jump to the resumption point in the function. Simple functions will have only one or two entry points, and their control flow is simplified to reflect that fact.

The Cilk implementation contains an additional refinement of the entrypoint technique: two copies of every parallel function are generated. The *slow clone* passes arguments on a shadow stack (the *steal stack*) and uses entrypoints. The *fast clone* uses the ordinary C calling conventions and is specialized with respect to the initial entry point. Ordinary function calls use the fast clone; the slow clone is used only after work stealing or suspension. A quick off-the-cuff experiment with this technique in Eager Haskell revealed that the resulting code ran substantially slower. Again, allocation and nested function call require arguments to be spilled to the shadow stack where they can be found by the garbage collector. By placing resumption points at existing control flow boundaries the cost of checking the entry point can be minimized, and increased instruction cache miss rates in the 2-clone code appear to dominate.

There are numerous other techniques for mapping source functions to C procedures. Scheme 48 [61] generates its interpreter by partially evaluating a simple scheme-like language called pre-scheme. Multiple scheme procedures are coalesced into a single C procedure; much of the interpreter collapses into a single function. Grouping functions in this way allows tail-recursive functions to be transformed naturally into loops, and permits calling conventions to be tailored to the context

	Function		
Application	Partial application	Known non-closure	Statically unknown
Partial	Rare	Uncommon	Uncommon
Full Arity	Common	Most common	Common
Oversaturated	Least common	Uncommon	Uncommon

Table 5.1: Different cases of curried function application and their presumed frequency. Here Least common $<$ Rare $<$ Uncommon $<$ Common $<$ Most common.

in which functions are actually used. However, Eager Haskell code is currently rather bulky, and this technique would subvert function splitting and result in unmanageably large C functions.

GHC chooses to place each thunk in a separate C function [93]. This fits in naturally with the lazy execution model: a thunk is entered and executed independently of the containing function. The resulting functions are knitted together by post-processing the assembly output of the C compiler.

The *pH* [31] and Mercury [48] compilers had a notion of entrypoints similar to the Eager Haskell compiler; however, they rely on a non-portable feature of gcc (taking the address of labels). This results in slightly faster code, but modern versions of the trick are not sufficiently robust for production use.

5.5 Currying

Currying is popular in Haskell, and curried functions must be represented in an efficient manner. However, C is *not* curried. Many ML implementations make tradeoffs in the efficiency of curried function application in favor of speeding up tupled application [7]; in Haskell such a tradeoff would generally be unacceptable. We make two key assumptions about the function calls in Eager Haskell programs: Most function calls invoke a known function, and most function calls occur at (exactly) full arity. Thus currying and higher-order function calls, while frequently used, still only account for a small proportion of calls overall. We also assume that a function is ordinarily only curried once; the resulting partial application is likely to be applied to all of its missing arguments. Semantically (see Figure 3-12), the merge app rule will usually be followed by an immediate β_{var}, and we will rarely need the split app rule.

The simplest implementation of currying statically transforms a function with n arguments into n functions each taking a single argument and returning a function. We would write this in λ_C as follows:

$$\lambda x_0\ x_1\ x_2\ x_3 \rightarrow e \quad = \quad \lambda x_0 \rightarrow \lambda x_1 \rightarrow \lambda x_2 \rightarrow \lambda x_3 \rightarrow e$$

This can be mapped very naturally to C, as every function call occurs at full arity. However, it does the worst possible job of the common case: a full-arity application of an n-argument function generates $n-1$ closures. The Id compiler [133] uses a similar technique, but adds a second "full arity" entrypoint to every function. This makes full-arity applications of uncurried functions efficient; full-arity applications of existing partial applications are still inefficient.

Techniques exist to statically eliminate currying [47]. In practice, these techniques require whole-program analysis. The analyses also tend to err on the side of the "most-curried" version of each function. Additional closures result, and full-arity application suffers.

Instead of statically compiling away currying, extant Haskell compilers generate code designed to deal with curried function application while making full-arity application as fast as possible. There are two basic approaches to compiling currying, the *eval-apply* approach and the *push-enter* approach. In examining these two techniques, we identify the function being applied as a partial application, a statically known function which has not been partially applied, or a statically unknown function. We identify a call site (dynamically) as a full-arity application, a partial application, or an over-saturated (more than full arity) application (note that we cannot know statically whether an application of an unknown function will be at partial arity, full arity, or oversaturated, so this distinction will need to be made at run time in these cases). Together, these give rise to nine different cases of function application, summarized in Table 5.1. Both techniques will group some of the nine cases together, and will move functionality required for the less common cases into the run-time system.

5.5.1 The eval-apply approach

In Eager Haskell we use the eval-apply approach to compile partial application. The compiler generates code for a function assuming it is being invoked at full arity. It is the caller's responsibility to create and unpack partial applications. This means that full arity application of a known function can use a simple, fast calling convention (such as the regular C convention). Because there are generally many more call sites than functions in a program, all calls to statically unknown functions are handled in the run-time system. The eval-apply technique is summarized in Table 5.2. In particular, no special treatment is given to oversaturated function applications. An oversaturated application is treated as a full arity application yielding a partial application or an unknown function.

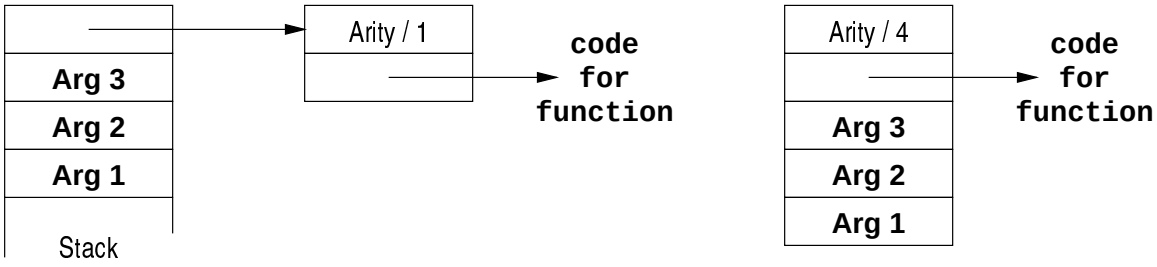
Application	Function: Partial application	Function: Known non-closure	Function: Statically unknown
Partial	merge (τ_a), store	static store	store stack
Full Arity	copy closure (ι_b), β_{var}	direct call (β_{var})	β_{var}
Oversaturated	merge (τ_a), split (ν), β_{var}, apply	split statically (ν)	split (ν), β_{var}, apply

Table 5.2: The eval-apply approach to partial application used in Eager Haskell. Rule names refer to rules in Figures 3-12 and 4-2.

Partial applications build a *closure*. In Eager Haskell a closure looks just like an ordinary data structure—see Figure 5-2. The tag indicates the object size (as usual) and the remaining arity of the closure. The first field is the function pointer. The remaining fields (if any) are the arguments to which the function has been applied. This means that closures are *flat*—if we apply a function of n arguments to i arguments, then apply the resulting closure to j arguments, we will construct two closures—one of size i and the second of size $i + j$. Closures are also required for functions with free variables; this is discussed in more detail in Section 6.3.

All three cases of known function application can be handled at compile time. Oversaturated applications of known functions are split into a full-arity application and an unknown application. Full arity applications result in a simple function call. Partial applications allocate and fill in the closure directly.

Any function call involving a closure is handled by a run-time system function called *General-Apply*. We represent unknown functions as closures, so *GeneralApply* handles both the "Partial Ap" and "Unknown" cases in Table 5.2. Finally, *GeneralApply* ensures that the closure has been computed. If this check were not done in *GeneralApply* it would need to be done at the site of every unknown function application.



State on entering the Apply function

Resulting partial application

Figure 5-2: Partial application of a simple closure.

	Function		
Application	Partial application	Known non-closure	Statically unknown
Partial	Copy, call, revert	Direct call, revert	Call, revert
Full Arity	Copy, call	Direct call	Call
Oversaturated	Copy, call, unwind	Direct call, unwind	Call, unwind

Table 5.3: The push-enter approach to partial application used in GHC.

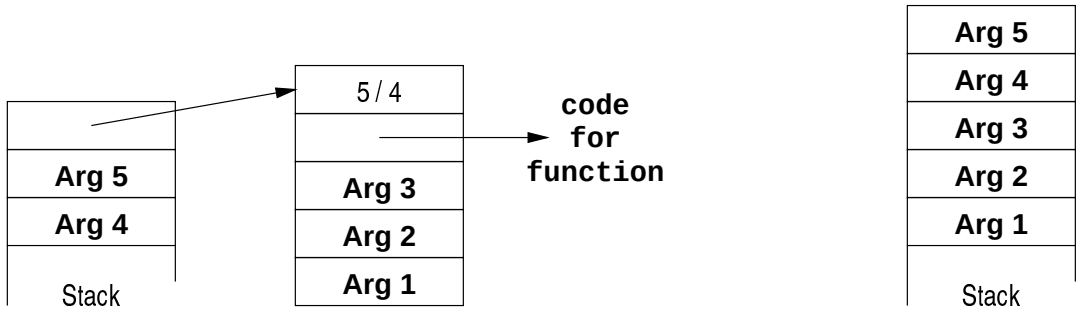
Compiled code invokes *GeneralApply* by pushing function arguments onto the shadow stack as usual. The closure is then pushed on top of them. The entrypoint passed to *GeneralApply* indicates the number of arguments which have been pushed; *GeneralApply* is therefore the only function which is called with a nonzero entry point from user code. This allows a single function to handle every possible case of function application.

5.5.2 The push-enter approach

The push-enter approach is used in GHC and hbc, and is described in detail in books on functional programming implementation [92, 99]. It differs from the eval-apply approach in two important respects. First, oversaturated applications are handled by a special return convention which avoids creating an intermediate closure for partial applications in tail position. Second, as a result of this return convention a function can be invoked at any arity; the burden of arity checking is shifted from the caller to the callee. This requires the use of a contiguous stack for argument passing (in practice the shadow stack is used). The push-enter convention used in GHC is summarized in Table 5.3.

The basic push-enter approach is very simple: the arguments (regardless of number) are pushed on the stack from right to left, and the function is called. Upon entry the function performs an *argument satisfaction* check; if not enough arguments were supplied, then the run-time system is invoked to create and return a partial application. Otherwise execution continues normally. Note that if a function is called with too many arguments, the arguments required for execution will reside at the top of the stack; no special code is required for this case.

By shifting the task of arity checking from caller to callee, we can make a special optimization for tail calls. In the eval-apply approach, if a partial application of g is in tail position in a function f then the compiler will generate and return a closure for the function. However, f's caller might immediately apply the resulting closure to additional arguments. In the eval-apply approach, the excess arguments are pushed before calling f. Now when the tail call to g occurs, the extra arguments



State on entering the Apply function

Stack on function entry

Figure 5-3: Applying a partial application to its remaining two arguments. Arguments must be shuffled to the top of the stack to make room for the arguments in the closure.

already reside on the stack, and *g* can run directly.

As with the eval-apply approach, the calling convention can be simplified when a function with known arity is called. Partial applications of known functions can branch directly to the run-time system. Full-arity and oversaturated applications of known functions can skip the argument satisfaction check. As a result, every function has two entry points: one for known applications, and a second for unknown applications which performs the argument satisfaction check and branches to the first.

A partial application is represented much as in the eval-apply approach, as an ordinary object which contains a pointer to the function and a copy of the arguments which must be pushed. The apparent entrypoint of a partial application is a run-time routine which checks for arguments on the stack; if any have been provided, the partially applied arguments are pushed and the partially applied function is entered. If enough arguments were provided, the function will run; otherwise a flattened closure will result.

5.5.3 Analysis

If the shadow stack is to be used for argument passing, it would seem that the push-enter approach has a compelling advantage over the eval-apply approach: it handles partial application and full-arity application just as gracefully, and avoids closure creation for oversaturated applications. However, the push-enter approach requires additional state: we must keep track of the base of the pushed arguments. This frame pointer must be saved and restored across non-tail calls.

The eval-apply approach also has a compelling advantage: flexibility. For example, function

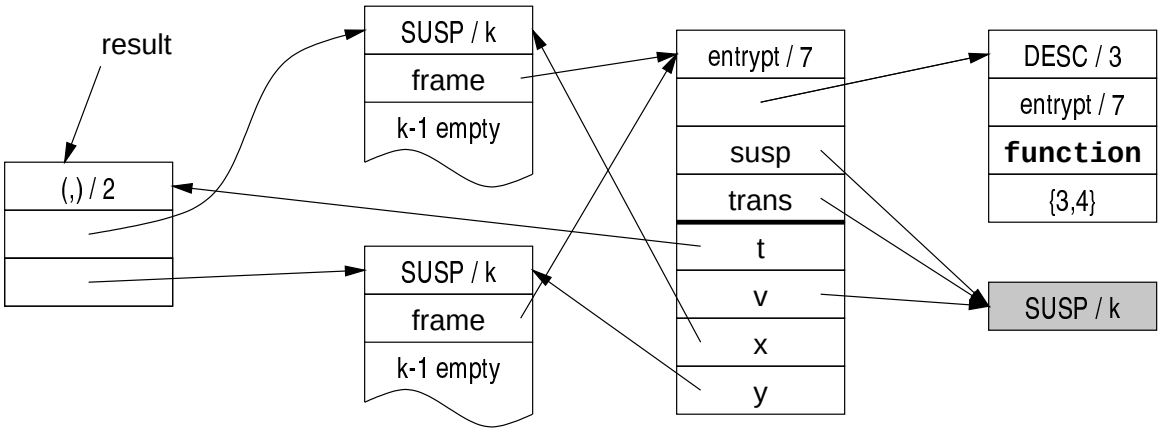


Figure 5-4: The computation of *x* and *y* suspended because *v* itself (in gray) suspended. These suspensions share a single synchronization point, and therefore share a single frame. The compiler-generated frame descriptor (top right) indicates that frame entries 3 and 4 should be resumed together. The suspended-upon field and the transitive suspension field both point to *v*.

arguments can be passed in either left-to-right or right-to-left order. Pushing arguments from right to left makes applying a closure simpler: the partially-applied arguments can simply be pushed onto the stack as in the push-enter approach.

Pushing arguments from left to right requires sliding the stack to make room for closed-over arguments (Figure 5-3). However, lambda lifting (Section 6.3) works by abstracting functions with respect to their free variables. These are therefore the first arguments in any function application. In a tail-recursive loop, the free variables will be preserved. Using left-to-right argument order, arguments and local variables can be pushed and popped without affecting the free variables. For example, the first three arguments to the function in Figure 5-3 might be its free variables; computation can leave these elements of the stack in place across tail calls. In practice we expect tail-recursive loops to be much more common than partial applications, and we therefore choose to push arguments from left to right.

5.6 Suspensions

The semantics of suspensions were discussed in Section 4.2.9. A function may have several associated suspensions. We make no attempt to share state between suspensions, or between suspensions and the currently-running function activation. We suspend simply by copying chunks of the stack. If a function has *n* independent suspension points, we may end up with *n* copies of its frame. Nonethe-

less n is statically bounded albeit potentially large.

Sharing suspension state places several severe constraints on the compiler and run-time system. First, extant suspension state (if any) must be tracked. Second, the structure of a function's frame must be carefully chosen to permit sharing. In practice this means that variables must be assigned fixed, non-overlapping frame slots. This scuttles attempts to represent the frame compactly, and requires a liveness map for each suspension point to prevent the garbage collector from retaining dead frame entries. Because most function calls will never suspend, we reject suspension techniques which bottleneck access to the frame.

A suspension may, when executed, produce values for several bindings. Consider a **letrec** with two recursively-produced values x and y which are fed back:

$$\begin{array}{lll} \textbf{letrec } t & = & (x,\ y) \\ \quad v & = & \mathit{const}\ 7\ t \\ \quad x & = & v + 2 \\ \quad y & = & v * x \\ \textbf{in } t & & \end{array}$$

This leads to a problem with sharing. We must allocate a location for x and a location for y in order to construct t. If an exception is signaled while we are running *const*, v will not be a value. We synchronize once on v before computing x and y. Both x and y must become valid suspensions. Moreover, if x is forced this fact must be reflected in y and *vice versa* or we will duplicate computations. Thus, a suspension is a two-level structure, as shown in Figure 5-4. The complete contents of the suspended frame are shared. Suspended locations like x and y contain a pointer to this shared frame, and have their tags changed to "suspended".

In addition to the copied stack frame, the shared part of a suspension includes two additional pieces of data. The first of these is a compiler-generated suspension descriptor. When user code suspends, it ensures its frame is up to date on the shadow stack and calls the run-time system. The suspension descriptor is passed as an argument. This descriptor gives the function and entrypoint where resumption must occur and indicates where suspended variables such as x and y reside in the frame. This descriptor is used to construct the suspension and to run it when it is resumed.

A suspension will not be resumed until the variable which has been suspended upon is fully evaluated. This avoids performing a force action immediately followed by a suspend (Figure 4-6), whose net effect would be to copy data from a suspension to the stack and then back to the heap again. The run-time system stores a pointer to the suspended-upon data; if this is itself a suspension, that suspension is evaluated.

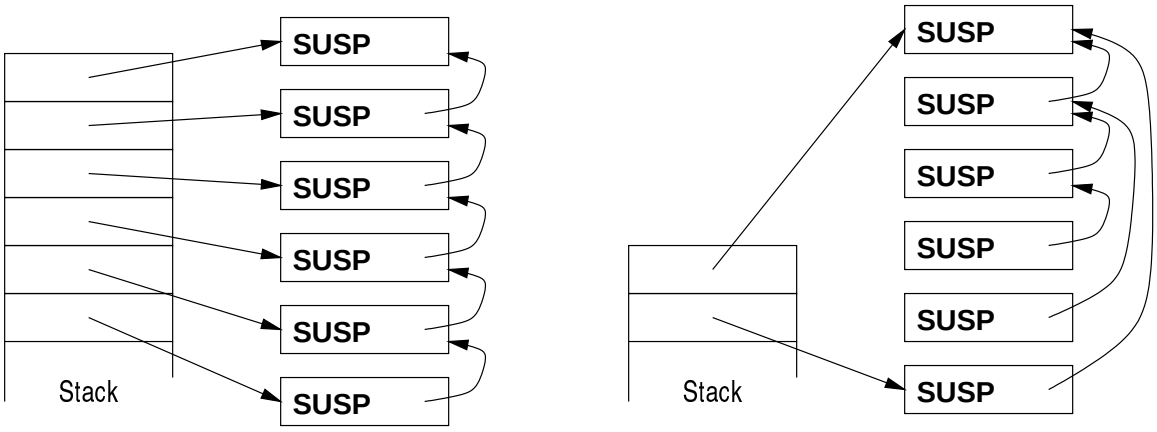


Figure 5-5: Updating transitive dependency fields.

Fallback creates long chains of dependent closures; indeed, there may be more closures than can be made to fit on the stack, since fallback occurs when the stack grows full. To limit stack growth we add a *transitive dependence* field. This initially points to the suspended-upon variable. It can later be adjusted by the run-time system to point to the variable responsible for a particular suspension. This variable is checked and forced first before the suspended-upon variable. Using this technique allows us to place a constant bound on the stack consumption of the run-time system.

Because fallback prevents stack overflow, arbitrary programs will make progress in bounded stack space. In principle, there is thus no need for constant-space tail recursion in Eager Haskell. In practice, of course, there are compelling reasons to preserve tail recursion wherever possible; the exception mechanism is comparatively expensive, and frequent stack growth results in poor cache performance.

Maintaining the transitive dependence field at first blush seems simple: simply traverse the chain of suspensions, and update the transitive dependence fields of objects on the chain to point to the first non-suspension. This can be done simply by keeping track of the beginning and end of the suspension chain. This is the technique used for shortcutting indirections, as shown in Figure 5-6. Commonly, however, the transitive dependence is a value, or can immediately be forced resulting in a value. Once the transitive dependence field points to a value, it is effectively useless, and we must re-traverse the chain of dependencies from the beginning.

The Eager Haskell compiler instead uses an algorithm which requires $2k$ stack slots, for some fixed $k > 1$. Figure 5-5 shows the technique for $k = 3$. As the chain of suspensions is traversed,

entries are pushed on the stack. When $2k$ entries have been pushed, the transitive fields of the k oldest entries are made to point to the k entries above them, in reverse order. The interior $2k-2$ entries are popped, leaving the initial and final entry on the stack. The traversal continues from these two entries.

To understand the source of efficiency in this technique, consider a chain of length exactly $2k$. We must force the suspensions in the chain starting from the end and working back to the beginning. After collapsing the chain, only entries 1 and $2k$ are on the stack. We force suspension $2k$ and pop it. We now need to force entry $2k-1$, but only entry 1 resides on the stack. Entry 1 is transitively dependent on entry $2k$, which has been evaluated, so its direct dependency is followed instead and 2 is pushed. The transitive dependency of entry 2 is $2k-1$. Once entry $2k-1$ has been forced, the direct dependency of entry 2, entry 3, will be followed. This will cause entry $2k-2$ to be forced, and so on. Thus, after collapsing the transitive dependency chain we can find the next suspension to be forced by chasing two pointers at a time.

5.7 Thunks

A thunk represents the suspended application of a function to some arguments. Thunks are introduced when a function call is encountered during fallback (Figure 4-7). An argument can be made for two different approaches to thunk representation—it is similar to both a closure and to a suspension.

A thunk is effectively a suspension of the *GeneralApply* function described in Section 5.5. In order to force a thunk, we must evaluate a closure and apply it to some arguments. However, the structure of a thunk is particularly simple. A function returns a single value, so there is no need to worry about producing multiple values when it is forced. The size of the frame uniquely determines the number of arguments involved in the application. The closure (which may be the suspended-upon variable) is always the first slot of the thunk.

Like a closure, a thunk represents a function applied to some arguments. There are two crucial differences between them. First and most important, a closure is a fully-computed value. A thunk is not—it requires forcing. It must therefore be clearly distinguished from a closure. Second, we represent closures in flattened form, as a code pointer and a group of arguments. Many thunks are created specifically because the apply function cannot obtain the value of its closure, so creating a flattened thunk is impossible in general.

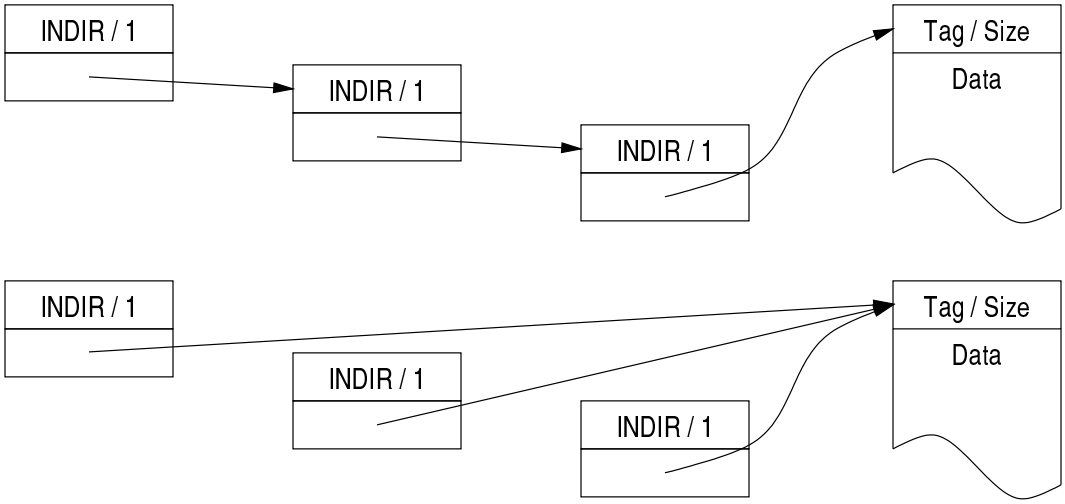


Figure 5-6: Elision and shortcutting of indirections.

We adopt a simple closure-like structure for thunks. However, we use a special "thunk" descriptor which is distinct from the descriptor for a suspension. The fields of a thunk are the function closure followed by its arguments. As with a closure, the arguments are stored in reverse order so that they can be copied directly to the stack. A thunk is forced simply by copying its contents verbatim to the stack and invoking *GeneralApply*.

5.8 Indirections

In common with lazy function language implementations, Eager Haskell requires the use of *indirections* [92, 99]. Problems with sharing occur whenever a computation returns an already-computed value (see also Section 4.2.8):

$$head\ xs@(x : \_) \quad = \quad x$$

This example seems simple enough: when we eagerly evaluate, we can simply fetch the value of x from xs and return it. However, consider what happens if xs is a suspension, and an exception occurs while it is being forced. In that case we must construct and return a suspension for *head xs*. When the suspension is successfully forced, it must be *updated* to refer to x; if this update does not occur, the suspension will need to be forced again every time it is used. If x is already a value, we might perform this update simply by copying x into the suspension for *head xs*. However, in general x may itself be a suspension. If we duplicate a suspension each copy will need to be forced

separately. We therefore replace *head xs* with an indirection pointing to *x*.

Compiled code treats indirections as uncomputed locations: attempting to evaluate an indirection causes a call to the run-time system. The run-time system handles indirections in much the same way as other implementations do [93, 31]—it *elides* them. Chains of indirections are followed until a non-indirection is reached. All the indirections in a chain are made to point to the non-indirection, as shown in Figure 5-6. If it is a value, it is returned to the compiled code; otherwise it is treated just as any other empty object would be. The garbage collector also elides indirections as they are traced; this allows indirections to be eliminated immediately. Indirections therefore cannot increase the total space usage of a program, though they will increase the rate of allocation.

The presence of indirections in the run-time machinery for Eager Haskell can have surprising effects on code generation. These effects are discussed in Section 6.10.2.

5.9 Garbage Collection

The need to box the results of every computations means that Eager Haskell programs are very allocation-intensive. Much of this data is short-lived. As a consequence, garbage collector performance has a first-order effect on the performance of Eager Haskell programs. However, the garbage collector is a separable piece of the Eager Haskell implementation, and its inner workings need not be understood in detail in order to follow the presentation in the rest of this thesis.

The Eager Haskell compiler uses a hybrid generational garbage collector. To keep object allocation fast, the nursery is a series of fixed-size *chunks* of memory. Allocating an object is a simple matter of incrementing the heap pointer. As each nursery chunk is exhausted, the heap pointer is reset to point to the next chunk. When the nursery is full, a nursery collection is initiated. Live nursery objects are copied, either into the new nursery or into tenured space. Tenured space is collected using a non-moving mark-sweep algorithm.

5.9.1 Multiprocessor collection constrains our design

The Eager Haskell garbage collector is designed to run efficiently on a uniprocessor, while still permitting multiprocessor coherence. Our experience with *pH* indicates that the scalability of multiprocessor garbage collection affects the scalability of the language implementation as a whole [31]. It is surprisingly difficult to retrofit a uniprocessor language implementation to permit multiprocessor garbage collection unless multiprocessor operation is part of the original design.

The need for efficient allocation and the need for efficient multiprocessor collection impose conflicting constraints on memory use: allocation must be fast, demanding a simple bump-a-pointer allocator and copying collection, and yet memory coherence must be as simple as possible, meaning that objects should not move around memory or otherwise change in ways that would require inter-processor synchronization.

Generational collection imposes a natural division upon the memory hierarchy: the nursery is purely local, providing fast allocation and access; tenured space uses a slower allocator, but its contents will not move and can be shared freely. This division has several important consequences for the present uniprocessor allocator.

5.9.2 Write barrier

Any generational garbage collector must track references from tenured objects to nursery objects. There are a number of algorithms for doing so, each of which enforces slightly different invariants on memory [60]. In Eager Haskell we guarantee that objects are tenured *en masse*—that is, if an object is promoted during nursery collection, then the objects it points to will be promoted as well.

On occasion, objects which are promoted by the collector are *updated* to contain pointers to nursery objects. For example, a long-lived suspension might be promoted and later forced; the suspension must then be updated with the forced result. In Eager Haskell we accomplish this by tracking updates to shared objects using a *write barrier*. In compiled code, any update which might possibly refer to a tenured object must check if a write barrier is necessary and if so call the run-time system. The impact of write barriers on compiled code is discussed further in Section 7.3.

In Eager Haskell the nursery is purely local. As a result, we must clearly distinguish pointers within shared memory (which may be dereferenced by any processor) from pointers which point from shared memory into the nursery (which may only be dereferenced by the owning processor). Checking every single pointer dereference would clearly add unacceptable overhead to our programs. Instead, Eager Haskell uses a special kind of indirection—the *barrier indirection*—to represent references from shared memory into the nursery. In addition to a pointer to the nursery object, a barrier indirection includes the identifier of the processor which created the indirection and a link field. The indirection may be followed only by the creating processor. The link field is used by the garbage collector to find all the barrier indirections created by a particular processor. When a nursery collection occurs, the objects referenced by barrier indirections are moved to tenured space, and the barrier indirections are changed into ordinary indirections.

5.9.3 Nursery management

By guaranteeing that nursery data will be purely local, we can allocate and collect each nursery independently and in parallel with the nurseries of other processors. There are a few aspects of nursery management which are worth highlighting.

As noted in Section 5.3, the Eager Haskell run-time system incorporates static tables of nullary constructors and small integers. When the collector finds such an object in the nursery, the reference is redirected to the static table. It is important to note that dynamic program operations such as *Int* arithmetic will *not* generate references to the static constructor table; there would be considerable overhead in checking each result to see if it lies within the table bounds. By relying on the garbage collector, only the small fraction of objects which survive garbage collection need be checked.

During a nursery collection, every single nursery indirection is removed. In order to remove an indirection, all locations which refer to that indirection must be changed to refer directly to its destination. This is easy during garbage collection, but frequently impossible in compiled code; by the time a pointer is discovered to refer to an indirection, the object from which that pointer was fetched may no longer be in scope. Thus, the indirection elimination performed by the garbage collector is strictly more powerful than that performed by the suspension mechanism in the run-time system (Section 5.8).

While the nursery is composed of fixed-size chunks of memory (currently 32 kilobytes, or 4K words), the number of chunks in the nursery is not fixed. The nursery size is set based on a target object retention rate R:

$$\text{chunks}_{\text{next}} = \frac{\text{words}_{\text{retain}} R}{\text{words}_{\text{chunk}}}$$

The goal is to allocate R words for each word which is copied by the garbage collector. In this way we hope to bound collection time with respect to allocation time while keeping the memory footprint small.

The estimate in the above equation makes a few sloppy assumptions. Retention rate is assumed to scale linearly with nursery size. Actual retention rates behave exponentially when retention is large. At the desired retention ($R = 53$) the behavior is close enough to linear for our purposes.

Number of words copied isn't necessarily an accurate estimate of collection time. There is a hard upper limit on nursery size of slightly more than 4MB; very large nurseries cause paging and TLB thrashing during collection. Similarly, a nursery which fits into the processor cache can be collected quite a bit faster than one which is just slightly larger; discontinuities in collection times

can therefore be observed when nursery size hovers around cache size. We have access to actual collection time, and could use that number instead of retention in setting nursery size. However, doing so would not address these discontinuities.

Current retention may not be a good predictor of future retention. When a computation changes phases, this assumption is usually violated. In particular, the fallback mechanism often induces large changes in allocation behavior even in otherwise uniform computations. Unfortunately, the run-time system has no simple way to predict the effect of phase changes. In practice, the assumption does not appear to lead to performance problems when R is set to acceptable values. It is possible to use other techniques to estimate future retention; for example, we can estimate the slope of the retention curve and use that to size the nursery. However, there is no general technique to predict sudden changes in allocation behavior.

5.9.4 Fallback policy

Control passes to the garbage collector code each time a chunk is exhausted; by measuring heap consumption in chunk-sized units, we shift heap resource checks to the garbage collector. The routine which parcels out nursery chunks is therefore also responsible for checking resource bounds and initiating fallback when necessary. This avoids the cost of checking resource bounds at every single allocation point.

Fallback can also be initiated when the C stack becomes too full; this condition is checked in the compiled code at every function entry point by comparing the C stack pointer to an overflow threshold. During fallback, this threshold is set so that the stack check always fails; the run-time system code for stack overflow creates a thunk for the called function.

In Figure 4-7, the stack bound $stack_{max}$ is assumed to be fixed. However, the total resource bound $resource_{max}$ is reset after fallback and after every garbage collection. The run-time system must therefore perform three computations specified in Figure 4-7: $e(|h|, |r|)$ computes the current resource usage of the program. $g(|b|, |h|, resource_{max})$ resets $resource_{max}$ after garbage collection. Finally, $f(|h|, resource_{max})$ resets $resource_{max}$ after fallback.

Both e and g are computed by the nursery chunk exhaustion code. At present, the stack of an Eager Haskell program is limited to approximately one chunk in size. Fallback due to stack overflow is nonetheless virtually unheard of; resource bounds are usually reached first. As a result, the amount of live stack is not significant and is ignored in computing e and g. Current resource usage e is computed based on *estimated* heap usage. Nursery usage is estimated by multiplying

the retention rate of the previous nursery collection by the current live nursery. Tenure usage is estimated in a similar manner—past retention multiplied by current usage. These quantities are computed precisely when garbage collection occurs.

At the moment the actual resource cutoff $resource_{max}$ is only reset when fallback occurs. It is set based on the previous two values of $resource_{max}$. The value of $resource_{max}$ shrinks slowly over time until a lower bound is reached; fallback becomes slightly more frequent as time passes.

There is a good deal more room for experimentation with fallback parameters. For example, a program with a large amount of persistent data which isn't changing over time will nonetheless have a higher measured heap usage; this means that such a program will fall back more often. Similarly, a larger nursery will (for any particular fixed retention rate) be more likely to cause fallback. It may be more productive to use two independent measures of resource consumption: total words allocated (which measures the passage of time), and total live data (which prevents excessive retention).

5.9.5 Promotion policy

In a generational garbage collector, promotion policy—which objects in the nursery are moved to the shared heap—is a strong determinant of performance. Objects which remain in the nursery for too long are repeatedly copied, increasing collection overhead. Objects which are promoted and die soon after will continue occupying memory until the next global garbage collection, increasing the memory footprint of the program and reducing TLB and virtual memory performance.

Every object which is reachable from tenured space is unconditionally tenured. In a functional language such as Eager Haskell updates to tenured objects are rare, and it is likely reachable objects will eventually be promoted regardless of the tenuring policy we choose. Tenuring them sooner eliminates copying overhead.

Of those objects which remain, only objects which have survived at least one garbage collection (and those objects reachable from them) are considered as candidates for promotion. This is easy to track; we simply keep track of the portion of the nursery which is occupied after each nursery collection. This is the same information we use to compute $words_{\text{retain}}$. However, rather than promoting all surviving data, we further constrain the collector: we divide the GC stack into two segments, the old segment and the new segment. The old segment contains a mixture of references to tenured data, references to static data, and references to data which has survived at least one nursery collection. Only the new segment may contain references to data allocated since the last nursery collection. Only objects reachable from the old segment are considered as candidates for

promotion. We assume that data in the new segment is actively being used in computation, and is more likely to die than data in the old segment.

Within the old segment we apply a similar heuristic to empty objects which are referenced directly by the stack. We assume these objects will be filled as soon as control returns to the stack frame which contains them. We keep a separate watermark for empty objects, only promoting those which are in the oldest part of the old segment of the stack.

5.9.6 Tenured space management

The tenured object space in the Eager Haskell implementation is designed to be as simple as possible. In its simplest incarnation, objects are allocated using the C `malloc` function and freed using `free`. Two bitmaps are used to track tenured data. One bitmap is used for marking during tenured collection. The second tracks the locations of objects so that dead objects can be identified and freed during the sweep phase.

There are two cases where objects are allocated directly in tenured space, rather than being promoted there from the nursery. Some arrays are too large to reasonably be copied. Such objects are expensive enough to create and initialize that we assume they will be long-lived in any case. Large arrays are therefore allocated directly in tenured space. In addition, the run-time system relies on external C libraries to support certain operations (most notably multiple-precision arithmetic). These external libraries do not obey the Eager Haskell calling conventions, and therefore cannot make use of the nursery heap pointer. The tenured allocator is used in these functions as a fail-safe allocation mechanism.

We saw in Section 5.9.2 that a write barrier must be used to track references from tenured space into the nursery. In addition, references from top-level data structures into the heap must also be tracked. These references are created when top-level computations (Constant applicative forms; see Section 6.6) are evaluated. This can be seen as a special case of the write barrier. The write barrier code detects when the object written does not reside in the heap; these objects are placed on a special *root list* which is traced when the mark phase is in progress.

Tenured garbage collection happens entirely asynchronously. When tenured space becomes sufficiently full, a flag is set indicating that a mark phase is under way. A mark bitmap is allocated and cleared. Marking is then interleaved with nursery collection: rather than ignoring references to tenured space, the collector marks them.

The mark phase continues until every processor has performed a full nursery collection. While

the mark phase is in progress, all newly-promoted data is assumed to be live (objects are allocated black [60, 143]). As a result, all live data will be marked. The sweep phase is (for the moment) very naive: the entire heap is swept in a single pass. Incremental sweeping is preferred in order to reduce caching costs and allow newly-deallocated objects to be reused quickly; we anticipate that such a change would be comparatively simple.

5.9.7 Problems with the tenured collector

The tenured collector pays for simplicity by being suboptimal in a number of respects. The amount of memory used by the tenured allocator is not bounded; we can still promote objects from the nursery after a mark phase is initiated. When nursery retention is high tenured space can often grow dramatically due to promotion during marking. In practice, however, nursery retention is only a problem when tenured space and nursery space are similar sizes—meaning the nursery is very large (due to high retention) but tenured space is very small (and usually set to grow substantially). We therefore accept soft bounds on memory consumption as a minor artifact of our collection strategy.

A mark phase is typically initiated in the middle of a nursery collection. However, marking is not considered complete until a *full* nursery collection is complete. Thus, on a uniprocessor the mark phase typically spans *two* nursery collection phases. We delay the start of marking until the start of the next nursery collection. On a multiprocessor (where other processors will be initiating nursery collections in parallel) this is likely to mean that the mark phase begins while the initiating processor is completing its nursery collection.

The use of monolithic mark bitmaps can lead to excessive memory consumption. The amount of memory required is equal to $1/32$ of the total address range of allocated objects. If tenured objects are allocated in a contiguous region, this amounts to a small (3.1%) storage overhead. However, there are often large holes where no tenured objects can be allocated—for example, the allocation of chunks for the nursery invariably creates holes in tenured space. Consequently, the actual space overhead of monolithic bitmaps is much larger. This space overhead is also reflected in sweeping time, as the sweep phase traverses the entire mark and allocation bitmaps.

Finally, the tenure allocator may not be easy to parallelize. Most `malloc` implementations rely on a single global lock, serializing object allocation even in multi-threaded programs. No attention is paid to issues such as false sharing of cache lines; such matters are only problematic on multiprocessor machines, and solving them usually requires slowing down uniprocessor allocation. Fortunately, special multiprocessor `malloc` implementations such as Hoard [25] address these problems well,

and can be used as plug-in replacements for the system `malloc` implementation.

However, there are other concerns in using `malloc` for tenured data. Much of the meta-data kept in the mark bitmaps and in object tags is redundant with information kept by `malloc` in its own data structures. The tenure allocator effectively pays twice to track this data. In addition, tenure allocation is very bursty—objects are allocated during nursery collection and freed *en masse* by the sweep phase. An ordinary `malloc` implementation assumes a much steadier pattern of allocation and deallocation [144]. Finally, the tenure bitmaps must be kept up to date using expensive atomic memory operations. By integrating bitmap maintenance with the synchronization already performed by the allocator, we can hope to reduce overall synchronization.

5.9.8 Towards better storage management

Fortunately, it is possible to make incremental modifications to the tenured allocator to address many of these shortcomings. Some tweaks are simple: for example, the sweep phase has been modified to ignore nursery chunks, mitigating one source of holes in the allocation bitmap.

A custom-written BiBoP allocator is now used for commonly-allocated small objects. Each chunk contains same-sized objects (though not necessarily the same tag). Objects are allocated by looking for clear bits in the allocation bitmap. Allocation can occur in parallel, requiring only a single atomic operation (test and set) to manipulate the bitmap. During the sweep phase the mark bitmap is copied to the allocation bitmap, implicitly freeing unmarked objects. This is substantially faster and more efficient than calling `malloc` and `free` for each object, though the underlying allocator is still used for large or uncommon object sizes. An additional refinement which has been implemented is to use separate chunks for pointer-free objects. Such chunks need not be scanned during marking, reducing the load on the cache and TLB.

Using a BiBoP allocator will permit further streamlining of the tenure allocator in the future. It is possible to maintain mark bitmaps on a per-chunk basis, perhaps by organizing the bitmaps into multi-level structures akin to processor page tables. This technique is used with great success in the Boehm-Demers-Weiser collector [28], where the conservative collection scheme can create numerous holes in the heap. Objects outside BiBoP chunks can be tracked using a naive technique such as storing them in a doubly-linked list. For large objects (more than 128 words) this technique has a lower space overhead than optimal use of a mark bitmap. This must be weighed against the complexity of distinguishing BiBoP objects from large objects during the mark phase; at the moment all tenured objects are treated identically.

Chapter 6

Lowering Transformations

The Eager Haskell compiler essentially uses a single intermediate representation—λ_C—for the entire compilation process. Optimization passes rewrite λ_C into λ_C; static analyses are defined on λ_C. This allows compilation phases to be re-ordered with comparative freedom. Nonetheless, the λ_C generated by desugaring is not suitable for code generation. We must eventually constrain the form of this code in order to make sure it is suitable for code generation.

Thus, the transformations on λ_C really divide into two classes. *Optimizations* transform λ_C into equivalent λ_C; these passes are outlined in Section 6.1 and are (with the exception of Bottom Lifting, described in Chapter 8) well-understood. *Lowering* phases transform λ_C into a lower-level form suitable for code generation. The first step in lowering is to convert the program to argument-named form, a process which was described in detail in Section 3.5.5. This is followed by the following transformations:

- **Expand string constants** into the corresponding Haskell lists, or alternatively into applications of a string expansion function to some more compact internal representation.
- **Hoist constants** (except for small constants) to top level (Section 6.2).
- **Lambda lift** (Section 6.3)
- **Split huge expressions** (Section 6.4).
- **Top-level common subexpression elimination** (CSE) (Section 6.5)
- **Convert constant applicative forms** (CAFs) to thunks (Section 6.6)
- **Insert pseudo-constructors** (Section 6.7)

- **Insert back edges** (Section 6.8)

- **Make synchronization explicit** (Section 6.9)

In this chapter we examine the lowering phases in detail. Many of them are quite simple; we focus the majority of our attention on the final phase, which is the insertion of explicit synchronization into a λ_C program. First, however, we sketch the optimization phases performed on Eager Haskell programs once they have been type checked.

6.1 Optimizations

The Eager Haskell compiler does all of its optimizations on λ_C before any of the lowering transformations. With the exception of bottom lifting, which we cover in Chapter 8, these transformations are well-understood; briefly:

- **Simplification** consists mainly of ordinary reductions according to the rules of the λ_C calculus given in Figure 3-2: β, δ, χ, etc. Most other optimization phases depend heavily upon the program transformations performed by the simplifier.

- **Inlining** is really an application of the ι rules from λ_C; however, this can result in uncontrolled code growth. Therefore, unlike most other simplification rules inlining cannot be done unconditionally, but must be controlled by static heuristics. The Eager Haskell compiler, in common with other optimizing Haskell compilers [100], is very aggressive about inlining. Inlining enables many subsequent program transformations, including deforestation and strictness-based optimizations.

- **Local CSE**. In Section 6.5 we give a general correctness argument for CSE.

- **Strictness-based case reorganization**. Strictness analysis provides information on which arguments are unconditionally required by a particular function [92, 101]. This information allows us to perform code motion on **case** expressions when it would otherwise be unsafe according to the rules of λ_C. This is because strictness analysis gives information about infinite unfoldings of computations.

- **Class specialization** [57] specializes functions which make use of Haskell's type classes [45, 19, 90] to particular instances of those classes. This turns most instances of parametric poly-

morphism into monomorphism, and transforms calls of unknown functions (class methods) into calls of known functions (particular instances of those methods).

- **Deforestation** of list computations removes intermediate data structures, usually enabling additional optimizations. The deforestation performed by the Eager Haskell compiler is described in the author's Master's thesis [70] and is related to the shortcut to deforestation [38]. Unlike its precursors, the deforestation pass in Eager Haskell re-structures code to use iterative evaluation as much as possible, and to maximize the parallelism of the generated code. A more general approach to deforestation of *pH* and Eager Haskell programs has been explored by Jacob Schwartz [117]; it allows arbitrary code to be deforested, but does not allow the compiler to choose traversal direction or use associativity to eliminate inter-iteration dependencies and thus increase parallelism.

- **Full laziness** generalizes the hoisting of loop invariants. It must be done carefully in order to prevent large increases in space usage [110]. Section 6.2 gives a general correctness argument for full laziness. In an eager language, there is a risk that full laziness will create unnecessary computation, by hoisting an expression from a context where it would never have been run.

- **Arity raising**. We can η-abstract a function if all its call sites are known; such a transformation can be derived directly from the rules of λ_C. The arity analysis required is described by Jacob Schwartz in his thesis [117].

- **Bottom lifting** hoists error handling code out of function bodies, using type and strictness information to identify expressions which are guaranteed to diverge. We devote Chapter 8 to this topic, as it is necessary in Eager Haskell to prevent the eager evaluation of divergent functions such as *error*.

Many of the concerns addressed in this chapter in the context of lowering transformations apply equally to program optimization. For example, any **case** hoisting transformation risks losing eagerness as described in Section 6.9.1.

6.2 Constant Hoisting

In Section 5.3, we noted that *all* values in Eager Haskell will be boxed—that is, they must be represented by a tagged data structure in memory. In order to mitigate the expense of boxing, argument-named form permits static constants to be included in primitive expressions *P* (see Figure 3-10).

Constants which are immediately used in a primitive computation need not be boxed. Thus, our code generator can handle an expression such as $n + 1$ or $x \geqslant 5.0$ without boxing.

However, many constants must still occur in boxed form—they may be stored into data structures or passed as arguments to functions:

```
let x = 35
in fib x
```

It is very easy to construct an *aggregate constant* by applying a constructor to arguments which are themselves constant:

```
let nil = []
    h = 'h'
    i = 'i'
    iStr = i : nil
    hi = h : iStr
in hi
```

We would like to avoid generating code to construct constants on the heap. Instead, the code generator should emit a correctly-formatted heap object at compile time. We therefore need some way to distinguish static constants (especially aggregate constants) from dynamically-generated data.

We would also like static constants to be *shared*. If a constant occurs within a function, the statically-compiled heap object must be used by every call to that function. If a constant occurs in multiple functions, we would like to combine all instances of that constant.

We can accomplish both these objectives—distinguishing static and dynamic constants and sharing constants—by hoisting constants to top level. Any constructor at top level is a constant, and can be compiled as such; constructors which occur elsewhere are dynamic. If the same constant occurs in multiple places, all occurrences will be hoisted to the top level; it is then a simple matter to combine them using top-level CSE (Section 6.5). Constant hoisting is actually a specific case of full laziness—a static constant is a group of bindings which taken together have no free variables. Full laziness in general, and constant hoisting in particular, can be justified very easily using the rules of λ_C; see Figure 6-1.

Constant hoisting is logically separated from full laziness for several reasons. First, static constants play an important role in many simplifications, such as **case** discharge and constant folding (static δ reduction). It is therefore useful to leave constants where they will be used until late in compilation. Second, allowing constants in primitive expressions means that we do not wish to

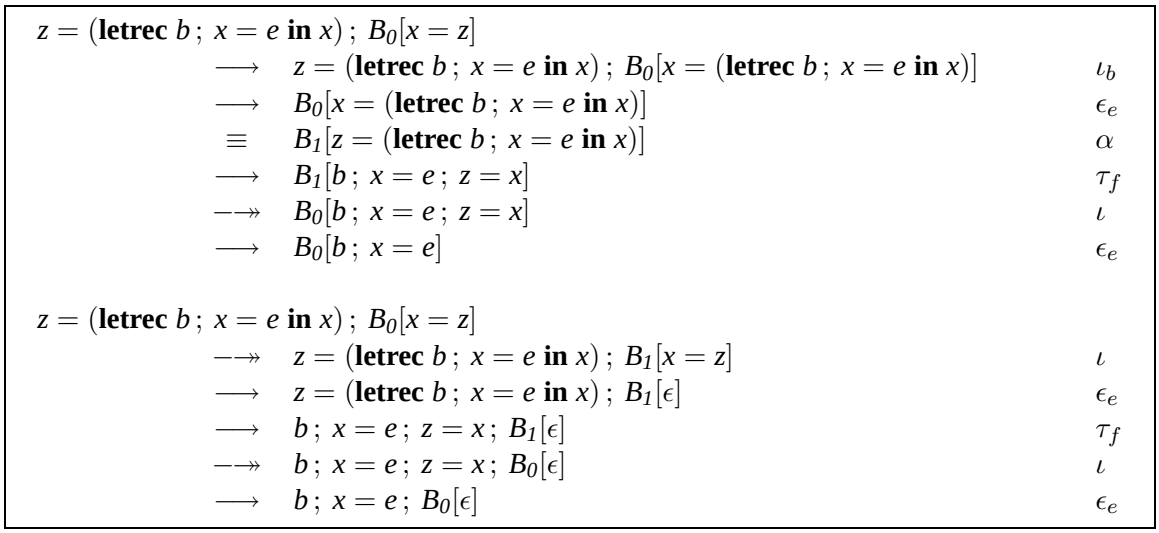


Figure 6-1: Correctness of full laziness in λ_C. Here $B_0[\,] = B_1[\,][x/z]$

unconditionally hoist constants; again, we should leave them in place until it is clear which primitive expressions will exist in the final program. Finally, it is frequently undesirable to perform full laziness on expressions with aggregate type [110]. However, constant expressions should be hoisted regardless of their type; we therefore must distinguish constant expressions when hoisting.

There is one final trick involved in the compilation of static constants. Small integer constants and nullary constructors are very common in Haskell programs. Moreover, such constants are frequently created dynamically as the result of primitive operations and function return. As noted in Section 5.3, the run-time system includes static tables of small constants. The code generator generates a reference to the appropriate static table when it encounters a boxed small constant. This avoids having the garbage collector do the redirection later on.

6.3 Lambda lifting

Any language which permits first-class, nested, lexically scoped functions must choose a representation for closures. A closure packages up the free variables and entry point of a nested function so that it can later be invoked. Numerous possible closure representations exist; sophisticated schemes for choosing among them have been explored in the LISP, ML, and Scheme communities [124, 62, 119, 7]. In principle, any of the closure representation techniques used in strict languages can be applied to Haskell programs. However, in practice extant Haskell implementations use one or two closure representations uniformly. There are a number of factors to account for this:

- Non-strictness itself is the major source of overhead in non-strict languages; implementation effort is better spent eliminating this overhead than optimizing closure representations.
- Uniform closure representations are simple to implement.
- Currying is popular in Haskell, and curried functions must be represented in an efficient manner (see Section 5.5).
- Closure conversion algorithms in strict languages focus on identifying short-lived closures; these can use a stack-based closure representation. The non-strict semantics of Haskell mean that function closures are likely to escape the context in which they were created, so very few opportunities for stack-based representation exist.

Eager Haskell represents closures by lambda lifting [53, 92, 99]. Lambda lifting a function replaces its free variables by function arguments. A closure is created simply by partially applying the lambda-lifted function to the actual free variables. However, the lambda lifting algorithm avoids creating closures by passing the free variables at function call sites wherever possible. Thus, no closure is created for a full-arity application of a known function. However, lambda lifted functions may in general require a large number of additional arguments.

Note also that Eager Haskell uses a *flat* representation for partial applications. It is possible to partially apply a function repeatedly. If that function has been lambda-lifted, then we must repeatedly fill in all the free variables in each closure. More sophisticated closure conversion algorithms share a single vector of free variables in this case. Indeed, it is possible to share free variable vectors when closing over multiple functions. However, care must be taken to ensure that this does not capture unnecessary free variables and cause space leaks [119].

Closure conversion need not be expressed as a program transformation. In the Glasgow Haskell Compiler, closures are explicitly distinguished, but no representation is chosen for them. Instead their free variables and arguments are tracked. A thunk is simply a closure which does not have any arguments; the two share exactly the same representation. The closure representation, fixed by the code generator, segregates boxed and unboxed data. Partial applications are represented in a different fashion from other closures; they contain a pointer to the function closure which was partially applied, along with an ordered vector of function arguments. A partial application cannot segregate boxed and unboxed data; they are freely intermixed and additional data is required for the garbage collector to distinguish which objects are pointers.

Performing lambda lifting as a separate compiler pass as in Eager Haskell has several advantages. First, it simplifies the code generator, which can assume that all free variables refer to top-level constructs. Second, it allows us to clearly separate the issue of closure representation from that of code generation. Ignoring typing, most closure representations could be expressed in terms of existing language constructs such as tuples. However, it is challenging to perform closure conversion in a type-preserving manner, particularly in the presence of unboxed types [129, 82].

6.4 Splitting huge expressions

The Eager Haskell compiler aggressively inlines functions which only have a single call site. This frequently enables further optimization based on the calling context—constant folding of arguments, fetch elimination (ι_d), case discharge (χ), common subexpression elimination, and so forth. However, it can also lead to extremely large functions in the compiler output (in some small benchmarks the entire program becomes a single top-level *main* function with a few internal loops). We are generating C code which is then compiled with an optimizing C compiler; the compilation time of gcc (when optimizing) is quadratic in the size of the largest function. As a result, it was sometimes taking hours to compile the C code produced by the compiler.

Conceptually, the solution to this problem is quite simple: split any large function into smaller functions. Functions cannot be split arbitrarily, as doing so might disrupt the flow of control in the compiled program. We only split entire definitions or case disjuncts, where changes in control flow already occur. Code which is split must be a certain minimum size: it is not beneficial to create a function if the resulting function call will be more expensive than the code which was split.

The actual splitting process relies on an unusual feature of our intermediate representation: it is possible to represent a λ-expression with no arguments at all. The lambda lifter treats such an expression as it would any other function, adding arguments to represent free variables and lifting the resulting function to top level.

Thus, we perform splitting by introducing zero-argument functions immediately before lambda lifting, replacing an expensive but splittable expression e with

$$\textbf{let } x = \lambda \to e \textbf{ in } x$$

A binding such as $x = \lambda \to e$ causes the lambda lifter to replace *every* occurrence of x with a call $x\ \vec{x}$. If there were more than one occurrence of x this would result in duplication of work.

$x = e\,;\ I_B[e]$	$=$	$x = e\,;\ I_B[x]$	ι_b
letrec $x = e\,;\ b$ **in** $I_E[e]$	$=$	**letrec** $x = e\,;\ b$ **in** $I_E[x]$	ι_e
let $x = e$ **in** $I_E[e]$	$=$	**let** $x = e$ **in** $I_E[x]$	ι_e
case $x = e$ **in** $I_D[e]$	$=$	**case** $x = e$ **in** $I_E[x]$	ι_c

Figure 6-2: Reverse instantiation is common subexpression elimination in λ_C.

Introducing an extra binding as shown ensures that the lambda-lifter introduces exactly one call to the split function.

6.5 Top-level common subexpression elimination

In procedural languages common subexpression elimination is usually restricted to primitive operations; in Haskell any expression may be subject to CSE. This is justified by reversing the rules for instantiation in λ_C—these reversed rules can be seen in Figure 6-2. Top-level CSE is concerned primarily with the first of these rules, which allows us to eliminate identical bindings:

$$\begin{array}{ll} x & = C_2\ 5\ 7 \\ y & = C_2\ 5\ 7 \end{array} \quad = \quad \begin{array}{ll} x & = C_2\ 5\ 7 \\ y & = x \end{array}$$

Note that CSE alone is not enough; ideally CSE ought to eliminate the identity binding $y = x$ as well. At the top level of a module, this task is complicated by the Haskell module system; both x and y might be exported to the outside world, in which case it may not be possible to eliminate all occurrences of either one. This is one of the reasons top-level CSE is separated from ordinary CSE; the other reason is that aggressive inlining can sometimes result in many copies of a single function. Top-level CSE after lambda lifting eliminates all but one of the extra functions. Finally, top-level CSE ensures that there is at most one copy of each static constant in a module.

One drawback of top-level CSE is that it is *per-module*. No attempt is made to combine identical top-level objects in separate modules. Thus, a function which is inlined in many different modules might leave a copy of its code in each of those modules. Mechanisms such as type class specialization [57] and rewrite rules [94] are a better solution than inlining if a function can be simplified on its own, without integrating it into its calling context.

When two bindings x and y can be combined, top-level CSE attempts to preserve whichever one is exported. For example, if x is exported, we simply replace all occurrences of y with x. If two exported constants are combined, replace one of them with an identity binding $y = x$; this represents a static indirection in the generated code, which is usually more space-efficient than including two

static copies of the same data.

Functions complicate matters somewhat; the compiler attempts to preserve the arity of all exported functions. For example, imagine x and y are defined as follows:

$$\begin{aligned} x &= \lambda a\ b \rightarrow (a * b) + a \\ y &= \lambda c\ d \rightarrow (c * d) + c \end{aligned}$$

In this case x and y can be combined. However, if we simply replace y by $y = x$, y will have arity 0 and all calls to y will pass through *GeneralApply* (see Section 5.5). For efficient compilation y must have the form $\lambda c\ d \rightarrow \ldots$, preserving its arity. Thus, we replace y as follows:

$$\begin{aligned} x &= \lambda a\ b \rightarrow (a * b) + a \\ y &= \lambda c\ d \rightarrow x\ c\ d \end{aligned}$$

It is fairly unusual to explicitly export the same function under two different names; we would therefore expect this corner case fairly infrequently. In practice, a definition can be exported to other modules when one of its uses is a candidate for inlining. This causes wrappers and static indirections to be generated more often than might be expected.

6.6 Constant applicative forms

We have detailed two compilation phases—constant hoisting and lambda lifting—which lift constant expressions to the topmost level of the program. The eventual goal is to have the compiler treat all top-level bindings as static constants for which no code needs to be generated. Bindings elsewhere in the program will be evaluated dynamically and will require code.

However, Haskell allows the programmer to write top-level bindings which are *not* constant. Haskell implementors refer to these expressions as *constant applicative forms*, or CAFs [92]. A CAF may represent an enormous amount of computation, so it is not acceptable to run it more than once. We must therefore ensure that the value of a CAF is preserved when it is run. This presents problems at a number of levels. First, we need to distinguish CAFs so that the code generator treats them as dynamic computations rather than static top-level objects. Second, we need to include some mechanism in the run-time system to record the value of a CAF when it has been run. Finally, we must complicate the run-time system with a mechanism—the root list, described in Section 5.9.6—for tracking CAFs which point into the heap.

In Eager Haskell we evaluate CAFs lazily. In the *CAF conversion* phase, we transform each CAF into a top-level thunk as follows:

$$t = e \quad \longrightarrow \quad \begin{array}{l} t = \mathit{thunk}\ t\_CAF \\ t\_CAF = \lambda \rightarrow e \end{array}$$

The code for the CAF *t* is encapsulated in a function *t_CAF* with no arguments. This function can only have top-level free variables, and thus need not be lambda lifted. A zero-argument function cannot be directly referenced in user code; there is no indication that it needs to be applied, nor any direct way to store its result. However, we can create a thunk for it; this thunk contains the function and no arguments. The code generator generates code for *t_CAF* just as it would for any other function. The thunk forcing mechanism can handle zero-argument functions without difficulty. When *t* is forced, it will be overwritten with the final value of *t_CAF*, and if *t* contains heap pointers it will be placed on a root list to be traced by the garbage collector.

6.7 Pseudo-constructors

In Section 6.5 we noted that top-level bindings of the form $x = y$ should be compiled as static indirections; in Section 6.6 we posited the existence of a *thunk* primitive to mark top-level thunks. However, at run time these constructs are simply represented by a special tag and some fields containing pointer data—that is, we construct them in exactly the same way we would construct an ordinary data object, the only difference being their special tag value. We exploit this fact to further simplify the code generator. We introduce *pseudo-constructors INDIR* and *THUNK* to represent the special tags, and transform the code into constructor expressions:

$$\begin{array}{lcl} x = y & \longrightarrow & x = \mathit{INDIR}\ y \\ y = \mathit{thunk}\ f\ a\ b & \longrightarrow & y = \mathit{THUNK}\ f\ a\ b \end{array}$$

Additional pseudo-constructors are introduced for language constructs with similar behavior. For example, bottom thunks (Chapter 8) use a special pseudo-constructor *BOTTOM*, and the *thunk* primitive is used internally to implement arrays (Chapter 9). Indeed, static partial applications can be represented using pseudo-constructors (though at the moment, for historical reasons, the Eager Haskell compiler handles them explicitly).

6.8 Back edge insertion

Once an execution mechanism has been chosen, the trickiest aspect of compiling a non-strict language is generating code for mutually dependent bindings:

```
letrec a = zipWith (+) p p
       p = 1 : a
in p
```

Under the lazy strategy (Figure 4-3), we create a thunk for every binding in a **letrec** block, then rely on the mutual dependencies between these thunks to fix an evaluation order for our code. To create the mutually dependent thunks, the compiler generates code which operates in two phases [92]. First, storage is allocated for all the thunks in the block—in this case, *p* must be large enough to hold a cons cell and *a* must be large enough to hold the application *zipWith* $(+)$ *p p*. Having allocated the storage, the thunks are then filled in. Here the final two arguments of *a* will point to the storage allocated for *p*. It is unsafe for other computations to use *p* or *a* until both have been completely filled in. Once this has happened, *p* can be returned and execution continues as normal.

In an eager language it is less clear how to proceed. Eagerly evaluating *a* requires a valid heap object for *p*. Simply transposing the bindings cannot help—the cons cell for *p* can then be built, but will require a valid heap object for *a*. We solve the problem by allowing *empty* heap objects to be created. Empty objects are full-fledged, valid heap data and can safely be used in computation. When an empty object is encountered, computation must suspend until it is filled in. The existence of these empty objects was noted in Section 5.3; they represent points in our semantics where the corresponding binding resides on the heap (Section 4.5). In the case of **letrec**-bound variables, this is because the relevant bindings occur later in program order and have not yet been reached.

In principle, the Eager Haskell code again proceeds in two steps: first, empty objects are allocated for each binding in the **letrec**. Then the bindings are executed (eagerly) in order. Each binding stores its result into the corresponding empty object, and changes its tag to indicate that it has been filled.

In practice, we need not allocate an empty object for *every* **letrec** binding. Instead, we allocate them only for bindings which are actually referenced before they are defined. Thus, in the above example an empty object would be allocated for *p* but not for *a*.

```
letrec  {- Back edge [p] -}
        a = zipWith (+) p p
        p = 1 : a
in p
```

If we construct a graph of dependencies between bindings, these will be the back edges of that graph. The back edge insertion phase adds annotations to the program which explicitly mark which variables lie along back edges and where empty objects must be allocated for them.

Note that the bindings in a **letrec** block need not occur in any particular order. We would consider different bindings to be back edges depending on the order chosen. The back edge insertion phase also chooses an order for bindings. A constructor or partial application can be allocated and filled in without suspending; by contrast, a more complex expression may require the value of a still-empty binding. For this reason, the Eager Haskell compiler orders constructors before bindings. After ordering p (a value binding) before a (a function call) and inserting back edges our example will look like this:

```
letrec  {- Back edge [a] -}
        p = 1 : a
        a = zipWith (+) p p
in p
```

6.9 Making synchronization explicit

The strategies given in Chapter 4 restrict instantiation to strict contexts $S[\,]$. Any synchronization which might be required to obtain the value of a variable is also restricted to a strict context (witness the force and suspend rules in Figures 4-3 and 4-6); a binding need not be examined (and synchronized upon) unless its value is needed for further computation.

We imagine that instantiation corresponds simply to loading a value from memory. Synchronization, however, is a good deal more expensive; in Eager Haskell it involves loading and checking the tag on a data structure. Tying together instantiation and synchronization leads to excessive synchronization. Consider this variation upon the humble fibonacci function, as compiled by the Eager Haskell compiler:

```
myfib x                 =
  case x < 2 of
    False               →
      let y             =  x − 1 in
      let z             =  y − 1 in
      let a             =  myfib y in
      let b             =  myfib z in
      a + b + z
    True                →  x
```

Here, x occurs in two different strict contexts—in the test $x < 2$ and in the binding for y. However, the binding for y cannot be computed until **case** $x < 2$ has discharged—thus, we *know* that x will be computed in both cases, and can avoid synchronization.

6.9.1 Introducing synchronization

In the Eager Haskell compiler we make synchronization explicit. This decouples instantiation and synchronization: instantiation must occur in any strict context; synchronization occurs only where it is explicitly indicated by a **case** expression. Introducing synchronization is similar in many respects to converting to fully named form (Sections 3.5.4–3.5.3). The η_l (lift) rule explicitly synchronizes an arbitrary expression. We can apply the rule to every variable which occurs as part of a primitive expression. This explicitly separates synchronization and instantiation. For our example, this yields the following code:

```
myfib x                          =
  case (case x of _ → x < 2) of
    False                        →
      let y                      = (case x of _ → x − 1) in
      let z                      = (case y of _ → y − 1) in
      let a                      = myfib y in
      let b                      = myfib z in
      case z of _                →
      case a of _                →
      case b of _                →
      a + b + z
    True                         → x
```

Once synchronization has been inserted, a hoisting step is required to return the program to argument-named form, using the strict hoisting rule σ_s. Compare this to naming (Section 3.5.4), where after naming using ν new bindings are lifted using σ_m. In our example, only the outermost **case** expression *requires* hoisting:

```
myfib x                          =
  case x of _                    →
  case x < 2 of
    ...
```

6.9.2 Eliminating excess synchronization

Making synchronization operations explicit does not actually reduce the amount of synchronization in the program. This task falls to the ι_l (unlift) rule, which eliminates nested synchronization. This allows us (for example) to eliminate the synchronization for *x* in the binding of *y* in *fib*. Similarly, the χ_d rule can be used (in conjunction with instantiation and uninstantiation) to eliminate synchronization on value bindings. When **case** *x* **of** $\_ \rightarrow e$ is eliminated the identity binding **let** $\_ = x$ **in** *e*

often results. This is easily avoided by tracking which variables will be subject to **case** elimination; we avoid inserting synchronization on these variables in the first place. Hoisting **case** expressions as they are introduced further reduces the need for excess synchronization.

However, synchronization elimination purely according to the rules of λ_C has two major problems. Consider the first part of *myfib* after synchronization elimination:

```
myfib x                  =
  case x of _            →
  case x < 2 of
    False                →
      let y              =  x − 1 in
      let z              =  (case y of _ → y − 1) in
      let a              =  myfib y in
      ...
```

Note that the binding for *y* will always yield a value when it is executed. We call bindings such as *y* and *a* which contain no explicit synchronization operations *non-suspensive*.

Note that while $a = myfib\ y$ is a non-suspensive binding, suspension may occur non-locally inside the recursive call $myfib\ y$. Thus, a non-suspensive binding does not necessarily yield a value. However, any non-suspensive binding which does not involve function call (constructors, non-synchronizing **case** expressions, and primitive functions) *will* yield a value. We call such a binding *unlifted*. No synchronization is required for occurrences of unlifted variables such as *y* once they have been bound.

We therefore *ought* to be able to avoid synchronization on *y* when computing *z*. We work around this by noting that an unlifted binding can be replaced by **case**:

```
myfib x                  =
  case x of _            →
  case x < 2 of
    False                →
      case y             =  x − 1 of _ →
      case z             =  y − 1 of _ →
      let a              =  myfib y in
      ...
```

This is justified by hoisting existing **case** expressions, instantiating, and erasing. Indeed, the compiler could simply represent unlifted bindings in this way rather than using **let**. We choose to represent the bindings using **let** as a notational convenience for the code generator. Therefore we obtain the following code for *myfib* after synchronization elimination:

```
myfib x                  =
  case x of _            →
  case x < 2 of
    False                →
      let y              = x − 1 in
      let z              = y − 1 in
      let a              = myfib y in
      let b              = myfib z in
      case a of _        →
      case b of _        →
      a + b + z
    True                 → x
```

6.10 Eliminating additional synchronization

There are a number of ways to improve upon the simple synchronization introduction described in the previous section. Because each synchronization operation carries a run-time cost, it is worth eliminating as many synchronization operations as possible. In this section we discuss several techniques for removing unnecessary synchronization.

6.10.1 Hoisting to eliminate redundant synchronization

We might consider hoisting synchronization which is *not* required to restore the program to argument-named form. For example, the following contrived example synchronizes twice on *y*:

```
let z                = (case y of _ → y − 1) in
case y of _          →
case z of _          →
y + z
```

If we hoist **case** *y* past the binding for *z* we obtain:

```
case y of _          →
let z                = (case y of _ → y − 1) in
case z of _          →
y + z
```

which simplifies to:

```
case z of _          →
let z                = y − 1 in
y + z
```

The binding for *z* is now unlifted, eliminating synchronization on *z* and simplifying the compilation of the binding for *z* itself.

Hoisting **case** expressions in this fashion is not always beneficial. Consider hoisting **case** *a* in *myfib*:

```
myfib x                 =
    ...
    let a               = myfib y in
    case a of _         →
    let b               = myfib z in
    case b of _         →
    a + b + z
    True                → x
```

We have made the binding $b = myfib\ z$ strict in *a*; this eliminates most of the eagerness in *myfib*. If we are attempting to run *myfib x* in parallel, that attempt has just failed. If *b* happened to consume a large data structure, then we have increased the space requirements of the program. The Eager Haskell compiler is conservative about hoisting synchronization past a **let** expression: hoisting is only performed if the expression bound is unlifted, or the definition involved is strict in the hoisted variable. Thus, the compiler hoists the contrived example, but does not hoist **case** *a* in *myfib*.

6.10.2 Using Transitivity

There are often transitive relationships between program bindings. Consider the code on the left-hand side of Figure 6-3. It should be sufficient to synchronize on *y*. Knowing that *y* is computed should *imply* that *x* is computed. This is shown by the unfold/fold argument given in the figure. We account for transitive dependencies when introducing synchronization into λ_C.

Unfortunately, a fully computed value may turn out to be an indirection. This complicates the use of transitive dependencies to eliminate synchronization; generated code would need to check for indirections even if synchronization is eliminated. In the common case (full location, no indirection) this is just as expensive as full synchronization. The Eager Haskell compiler and run time system take several steps to ensure that indirection checks need only occur at synchronization points.

First, the code generator ensures that the "most computed" value of a variable is kept. Once again consider the code in Figure 6-3. If *x* is an indirection pointing to the value 5, **case** *x* in the binding for *y* will call the run-time system, eliminate the indirection, and re-bind *x* to point directly to 5. Finally *y* is bound to 10. When we synchronize on *y*, *x* is bound to a value as well.

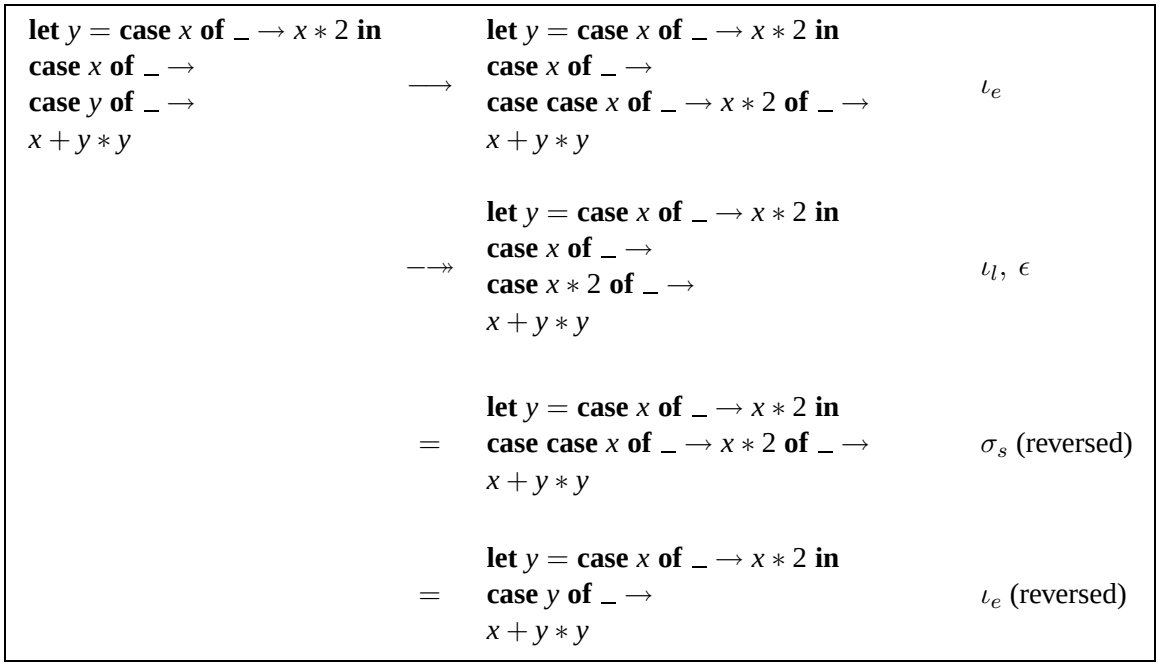


Figure 6-3: Synchronization elimination for transitive dependencies

Second, the run-time system eliminates *all* indirections in a suspended frame before resuming it. Consider what happens if *x* is initially empty. Then **case** *x* in the binding for *y* will suspend, and **case** *y* will suspend in turn. Now there are two different suspensions with a separate copy of *x*. Imagine *x* later evaluates to an indirection pointing to the value 5. When the result is demanded, the binding for *y* will be demanded *separately*. It does not matter that *y* will synchronize on *x* and eliminate the indirection; this will not change the second copy of the frame used to compute **case** *y*, where *x* will still point to an indirection.

6.10.3 Partitioning versus explicit synchronization

Past work on synchronization elimination in eager languages has focused on *partitioning*. Code is divided into *partitions*, each of which synchronizes on a series of inputs, then runs to completion, possibly spawning new partitions as it does so. This approach was implemented in the Id compiler by Ken Traub, who describes the technique in his dissertation [133]. Later work introduced interprocedural analyses to improve partitioning [112, 114, 33, 34, 132]. Similar work was done in the context of the strict language SISAL [111, 128].

Compilation based on partitioning has one important problem: there is no clear and natural connection between the unpartitioned and partitioned program. On a modern architecture, very dy-

namic control flow is expensive, and usually partitions end up being run in a particular order in practice. Virtual machines exist which simplify the task of creating and ordering partitions, including TAM [115, 42, 41] and PRISC [84]. However, functions and case disjuncts still incorporate a notion of "inlets", one per input (or group of inputs), which perform necessary synchronization when a value arrives from another thread. The work on the SMT [16] abstract machine and the code generator for *pH* [31] were the first attempt to consider thread ordering primarily as a compilation problem, rather than a mechanism to be provided by the abstract machine. Partitions are grouped into *non-suspensive threads*. This provides a natural flow of control; it also allows control dependencies between partitions in a thread to be used to eliminate synchronization. Arguments are passed by reference rather than by sending them individually to entry points.

Taking its cue from SMT, Eager Haskell focuses primarily on making sequential, non-suspensive control flow efficient. Synchronization elimination is an important part of doing so; however, we must not go to great lengths to eliminate synchronization if doing so will complicate control flow. Thus, while synchronization elimination in Eager Haskell is similar to demand set partitioning [133], we do not attempt to impose a new intermediate representation with special provisions for threaded control flow.

6.10.4 Interprocedural synchronization elimination

At the moment, the synchronization introduction phase of the Eager Haskell compiler only accounts for transitive dependencies within one procedure. However, functional programs are frequently composed of many very small functions. We might hope to remove a good deal more synchronization simply by propagating interprocedural dependency information.

There are several techniques that can be applied to compute such dependency information. Interprocedural partitioning analyses [112, 33, 34, 132] attempt to compute a partial order among inputs and outputs of a function in order to impose ordering constraints among partitions. However, those algorithms which were successfully implemented had very high complexity in practice [113]. In addition, one of the chief benefits of interprocedural partitioning analysis was the ability to group multiple function inlets into a single entry point. Because *pH* and Eager Haskell use a single entry point for functions, the benefit to using such an analysis is likely to be much smaller.

let $y = f\ x$ **in** $g\ y$	$\longrightarrow$	**case** $y = f\ x$ **of** $\_ \rightarrow$ $g\ y$	(in GHC)
	$\longrightarrow$	**let** $y = f\ x$ **in** **case** y **of** $\_ \rightarrow$ $g\ y$	(in Eager Haskell)

Figure 6-4: Worker/wrapper requires additional synchronization under Eager Haskell. Here g is strict in y.

Using Strictness Information

The interprocedural analysis problem becomes much simpler if, instead of partially ordering inputs and outputs, we restrict our attention to a much simpler property: whether a particular input is *always* required during computation. This is strictness analysis [92]. The literature on strictness analysis is vast; however comparatively few algorithms are in daily use, and these are generally the simplest ones. Simple analysis produces useful improvements in practice; more complex analyses simply do not yield enough additional precision to justify their cost.

For example, the Glasgow Haskell Compiler computes strictness directly as a boolean property of function arguments [101]. Strictness information is used to divide functions into a *worker* and a *wrapper*. The wrapper ensures all strict arguments have been computed, then invokes the worker. The worker is compiled assuming the strict arguments are available. The hope is that many strict arguments can safely be computed before a function call; by inlining the wrapper the compiler attempts to eliminate much of the synchronization at each call site based purely on the calling context. In particular we can eliminate synchronization on the strict variables of loops and recursive calls entirely.

The Eager Haskell compiler performs strictness analysis (repeatedly), representing strictness using boolean expressions. This is used for strictness-based code motion and for a weak worker-wrapper transformation directed primarily at unboxing tuples (not at reducing synchronization). The present synchronization introduction algorithm uses strictness information to determine whether hoisting synchronization past a function call will result in a loss of eagerness. However, strictness is not used directly to eliminate synchronization. The problem (once again) is indirections. If $y = f\ x$ and f is strict in its single argument, then synchronizing on y will ensure that x has also been computed. However, x may still refer to an indirection which was eliminated within the call to $f\ x$. An indirection check is just as expensive as synchronization; little has been saved.

$$
\begin{array}{lcll}
L & ::= & E & \mid\ U \\
 & \mid & \textbf{let } x = E \textbf{ in } L & \mid\ \textbf{letrec } B \textbf{ in } L \\
 & \mid & \textbf{let } \underline{x} = U \textbf{ in } L & \\
 & & & \\
E & ::= & \textbf{case } P \textbf{ of } D & \mid\ x\ \vec{x}_k \\
 & \mid & \textbf{case } \underline{x} \textbf{ of } D & \mid\ \underline{x}_k\ \vec{x}_k \\
 & \mid & \textbf{case } \underline{x} = x \textbf{ of } D & \\
 & & & \\
U & ::= & V & \mid\ P \\
 & & & \\
P & ::= & \underline{x} & \mid\ C_0 \\
 & \mid & p_k\ \vec{P}_k & \\
 & & & \\
B & ::= & x = E & \mid\ B\,;\ B \\
 & \mid & \underline{x} = U & \mid\ \textit{Back edge}\{x\} \\
 & & & \\
D & ::= & C_k\ \vec{x}_k \rightarrow L & \mid\ D\,;\ D \\
 & \mid & \_ \rightarrow L &
\end{array}
$$

Figure 6-5: Fully synchronized, lowered λ_C. Compare with Figure 3-10.

If instead we synchronize on x in the calling context, we transform every binding of the form $y = f\ x$ to a binding of the form $y =$ **case** x **of** $\_ \rightarrow f\ x$. There is no longer any concern about indirections, as the synchronization is handled within the local frame. This is the effect of the worker/wrapper transformation in GHC. However, this can actually *increase* the amount of synchronization in an Eager Haskell program. The problem occurs when a strict argument is itself the result of a function call. Consider the example in Figure 6-4. In GHC, the binding $y = f\ x$ will be unwrapped, yielding a **case** expression as shown. However, this **case** expression carries no runtime cost—the call $f\ x$ is *guaranteed* to return a value. Indeed, the transformation has eliminated the need to construct and force a thunk for y. By contrast, Eager Haskell *cannot* guarantee that $f\ x$ returns a value; if resources are exhausted during the call, it will suspend. Thus, a separate synchronization step is required. Performing the worker/wrapper transformation on g therefore replaces a *single* synchronization in g with a synchronization operation at every call site of g. This will in general increase the size and complexity of generated code.

Computedness

In Eager Haskell, *computedness* can be used to eliminate synchronization. Computedness indicates which arguments of a function are *certain* to be values when the function is called. In principle,

this is easy to compute: consider every call site of a function. If an argument has been computed at every call site, that argument is certain to be computed, and it need not be synchronized. At this level the property can be computed using bit vectors to represent sets of strict arguments.

The real challenge, of course, is in identifying call sites. In a higher-order language, they can be approximated using control-flow analysis [122, 83, 55]. Because Eager Haskell compiles files a module at a time, full control flow analysis is impossible. However, the vast majority of functions are only called at full arity. We can do a good job of computedness analyses on these functions. Furthermore, we can use a technique similar to the worker/wrapper transformation or to loop preheader insertion [7] by splitting recursive functions into a *header* which performs necessary synchronization and a *body* which is always invoked locally at full arity. In this way we hope to capture many of the benefits of strictness analysis without increasing the amount of code for synchronization, while at the same time maximizing the effectiveness of a very simple computedness analysis.

6.11 Canonical lowered λ_C

Figure 6-5 shows the final syntax of *lowered* λ_C after performing the lowering transformations described in this chapter. The most obvious difference is that lowered λ_C explicitly distinguished unlifted constructs from lifted constructs. We use underlines to distinguish unlifted variables $\underline{x}$ from their lifted counterparts x. Unlifted expressions U are distinguished syntactically from lifted expressions E, and unlifted bindings $\underline{x} = U$ from lifted bindings $x = E$. When an unlifted variable is a back edge in a **letrec**, it is referred to in lifted form as x before the binding and in unlifted form as $\underline{x}$ afterwards. We distinguish three forms of **case** expression: **case** P **of** D, where P is a simple primitive expression which need never be boxed; **case** $\underline{x}$ **of** D, which dispatches on an unlifted variable and requires no synchronization; and **case** $\underline{x} = x$ **of** D, which synchronizes on x and suspends if necessary.

Beyond the distinction between lifted and unlifted constructs, there are a few other distinctions between lowered λ_C and argument-named λ_C. Known full-arity applications $\underline{x}_k\ \vec{x}_k$ are distinguished from unknown applications $x\ \vec{x}_k$; recall from Section 5.5 that known applications can call the target function directly, whereas unknown applications must invoke the *GeneralApply* function in the runtime system. Pseudo-constructors mark thunks, static partial applications, and variables which occur in an expression context. The bindings in **letrec** expressions are considered to occur in program order; back edges are marked explicitly.

Chapter 7

Eager Code Generation

In Chapter 6 we examined lowering transformations intended to simplify the task of code generation. In this chapter we detail the actual code generation process. The code generator begins with the low-level λ_C shown in Figure 6-5. The final output of the code generator is a C program; there is no special low-level intermediate representation.

The mechanics of suspension and garbage collection are the main source of complexity in the code generator. Minimizing the number of save points (Section 7.1) reduces the need to keep the frame up to date, and decreases the complexity and size of the entrypoint mechanism sketched in Section 5.4. The structure of the frame (Section 7.2) determines how variables are saved and restored and how the garbage collector identifies live variables. Minimizing the number of points at which allocation can occur reduces the amount of code required to keep the frame up to date, but can complicate garbage collection (Section 7.3). Once frame structure and save point placement have been determined, the remainder of code generation is reasonably straightforward; the remainder of the chapter details the mapping from λ_C code to C.

7.1 Save points

In Eager Haskell there are four situations in which local data must be saved and restored from the shadow stack: When computation suspends (Section 5.6), when allocation fails and the garbage collector is invoked (Section 5.2.3), when a suspensive binding is started, and when one function calls another one. This last case exists because any function call may invoke the garbage collector (every function may allocate: if nothing else, the exception mechanism may need to allocate a thunk for the call).

We refer to places where the compiler saves and restores live data as *save points*. The Eager Haskell compiler attempts to combine save points as much as possible. Consider the λ_C code for the *zip* function from the Haskell prelude:

```
zip xs ys                =
  case xs of
  []                     → []
  (x : xt)               →
    case ys of
    []                   → []
    (y : yt)             →
      let tl             =  zip xt yt in
      let hd             =  (x, y) in
      hd : tl
```

Here both **case** *xs* and **case** *ys* may suspend. Note, however, that if **case** *ys* suspends it is perfectly safe to repeat the entire computation of *zip* upon resumption. Doing so simply results in a redundant test of *xs*. The compiler therefore re-uses the same save point for both **case** *xs* and **case** *ys*. Similarly, the recursive call *zip xt yt* always requires the live variables to be saved and restored. If the allocation of *hd* invokes the garbage collector, there is no need to save any state at all. After garbage collection, execution can be resumed immediately after the recursive call, *before* the live variables are restored. The attempt to allocate *hd* will be repeated.

The rules for determining save points are very simple. At compile time, we keep track of the preceding save point, if any. A **case** expression will share this save point (if none exists, a new one is introduced). A call to the allocator can share a preceding save point, but subsequent code may not and the save point must be killed. A function call always requires its own save point, as the live variables are unconditionally saved across the call. A suspensive binding kills the save point; however, the live variables of subsequent computations are saved and restored by the thread spawn, so it acts as a new save point.

By assigning the same save point to nested **case** expressions, the number of function entry points is reduced, in turn reducing the amount of testing and branching required for the entry point mechanism. In the case of *zip* there is a single entry point—the beginning of the function—and as a result there is no need for any kind of entry point check at all. This is typical of Haskell code: functions written using pattern matching result in a series of nested **case** expressions, and there is never any need to check the entry point.

There is one restriction on sharing of save points by **case** expressions: A **case** cannot discrimi-

nate upon a field fetched in a **case** expression sharing the same save point. This typically occurs as a result of nested pattern matching, yielding code like this from Queens:

```
h m v1                          =
    case v1 of
      (n, i2)                   →
        case i2 of
          []                    → True
          v2 : v3               →
            case v2 of _        →
            ...
```

Here, a new save point is required for **case** *i2* because *i2* is fetched from *v1*, and another new save point is required for **case** *v2* because *v2* is fetched in turn from *i2*. This restriction is another effect of the use of indirections. Imagine that *v1* is a tuple whose second field *i2* is an indirection, and that the two outer **case** expressions share a single save point. When **case** *i2* encounters the indirection, the run-time system will be invoked and control will resume with **case** *v1*. However, *v1* will still contain an indirection, and **case** *i2* will once again fail.

An alternative to splitting **case** expressions in this way would be to have the run-time system remove indirections reachable transitively from the frame. However, barrier indirections cannot be eliminated until the referenced data has been promoted by the collector. Eager tenuring of nursery data tends to move empty objects out of the nursery, increasing write barriers and dramatically reducing performance. We therefore accept additional save points as a necessary cost of an efficient garbage collection strategy.

The only cost of grouping save points is the expense of a small amount of extra computation in the rare event that a **case** expression suspends or that the garbage collector is invoked. In addition to reducing the number of entry points, grouping save points has a number of other beneficial effects. The C code produced by the compiler is shorter, resulting in smaller binaries and faster compile times. Combining entry points also appears to improve the *quality* of the machine code generated by the C compiler. Most combined save points occur at natural control flow boundaries, such as function call sites and the beginnings of threads; fewer basic blocks means greater scope for code transformation.

7.2 Frame structure

It is tempting to view the frame simply as an activation frame, where local variables are kept during computation. However, in Eager Haskell the shadow stack is used primarily to communicate *live pointer data* between compiled code and the run-time system. The compiler must therefore ensure that the frame is *precise*—that is, that the run-time system can identify which frame data is still in use. If frame information is imprecise, there will be space leaks: the garbage collector will retain large data structures which are no longer in use (as they will be referred to either from the stack or from a suspension). It is important for the compiler to guarantee precision: in a functional language there is no way for the programmer to eliminate space leaks which are artifacts of compiler imprecision [119, 107, 106, 140, 123]. By contrast, procedural languages generally accept imprecision; the programmer can fix space leaks by "nulling out" variables that cause excess retention.

The most flexible technique is to emit liveness maps, which indicate which frame entries actually contain live pointer data. This allows the frame to be structured in the most efficient possible manner—indeed, pointer and non-pointer data can be commingled on the shadow stack, simplifying unboxing (Section 13.3.5). However, the run-time system must find the appropriate liveness map in some fashion. Efficient techniques associate liveness information with return addresses [60, 143]. This is difficult when compiling via C: There is no simple way to create static tables relating return addresses to liveness information.[1] As a result, a compiler producing C must push additional liveness information onto the stack at every function call.

Instead, in Eager Haskell we ensure that the frame contains only live data. This means that the frame will always be as small as possible. Saving and restoring the frame when suspension occurs is a simple matter of copying a segment of the shadow stack; however, the compiler must copy data from one part of the frame to another when the frame shrinks. Thus, the stack will be smaller than if an explicit descriptor were passed, but more instructions are required to save and restore stack state. The frame need only be kept up to date at save points; minimizing the number of save points mitigates the cost of stack shuffling. However, frame updates are unavoidable at call sites, so it is unclear whether the Eager Haskell implementation benefits from keeping the frame small. We expect that liveness maps will be required as the compiler evolves.

[1] Indeed, creating a single static table from information contained in multiple object files produced by multiple compiler runs requires non-portable linker support [67].

7.3 Allocation

As noted in Section 7.1, heap allocation can share its save point with prior code, but subsequent code will require a new save point. This can be a particular problem when a series of allocations occur in quick succession. Worse still, each such allocation will perform a separate check of the heap pointer and will require separate code to invoke the garbage collector. Fortunately, the Eager Haskell allocator makes it simple to *batch* allocations: a single allocation action yields a large chunk of memory which can be initialized to contain multiple heap objects.

The existence of back edges and recursive bindings complicates allocation. When a variable is recursively bound, an empty object must first be allocated to hold the variable's value (Section 6.8). Later, the empty object must be *updated* to reflect the actual value of the binding.

The present Eager Haskell compiler simplifies the problem by assuming that every binding updates a previously-allocated empty location. For constructors and primitives it is easy to arrange for the empty location to be large enough to accommodate the resulting value. Even when a binding contains a conditional, space can be allocated for the largest of the possible results—unused space will be reclaimed by the garbage collector.

It is not possible in general to statically determine how much space the result of a polymorphic function call will require. The Eager Haskell compiler allocates a fixed-size *continuation object* (or simply continuation) for every function call. The continuation is passed to the function; if the result of the function is small enough, the continuation is updated directly, otherwise the continuation is updated with an indirection pointing to newly-allocated storage which holds the result.

Experiences with the code generated by the Eager Haskell compiler have pointed out a number of poor features in this technique for dealing with updates. The need to pass a continuation object to every function call naturally increases function call overhead. Also, the Eager Haskell compiler assumes every single binding is an updating store, which means every binding requires a write barrier for generational garbage collection.

In addition, the compiler presently batches allocations at control flow boundaries (multi-branch or suspensive **case** expressions). This means that the space for long chains of bindings—even if they are non-recursive—is allocated in one go and then gradually filled in. This in turn makes it more likely some objects will be promoted between allocation and update, requiring a write barrier. It also increases the time spent in garbage collection by adding live (but empty) objects which must be moved. The compiler has traded the cost of checking the heap pointer repeatedly (due to less

aggressive batching) for the cost of performing additional write barriers (by keeping empty objects alive across garbage collection). The overhead of a write barrier is approximately the same as the overhead of allocation in the common case; nothing has been gained.

An alternate implementation of updates would compile code such that only bindings on back edges were candidates for update. A back edge annotation causes an empty object to be allocated which is large enough to hold an indirection. This is used as a proxy for the final value of the binding. When the binding has been computed, the proxy can be updated with an indirection pointing to the new binding. This approach has a number of benefits. Continuation objects can be eliminated, leading to a more efficient (and more natural) function call mechanism. Allocation can be batched so that groups of unlifted bindings are allocated at once; all bindings will perform initializing stores. Finally, only empty proxies are subject to update, and they are updated only with indirections, so the write barrier can use a simple mechanism for recording outstanding writes.

The problem of updates exists in Haskell as well as in Eager Haskell, and the mechanisms used in lazy compilers closely parallel the ones described in this section. Most early Haskell implementations took some pains to ensure that objects were updated in place whenever possible [92, 99, 93]. However, implementations must support indirections irrespective of the existence of updates. Because there are efficient mechanisms for eliminating indirections, providing a special mechanism for update-in-place may well be a false economy. Recent versions of GHC have eliminated update-in-place entirely in favor of indirection [75].

7.4 Functions

Generating code for a lambda-lifted function is a straightforward process. Figure 7-1 shows the code for the outermost portion of the *fib* function. The first line defines the static closure for the function, `Main$fib`. The static closure is used when the *fib* function is passed as an argument to a higher-order function or stored into a data structure. The tag of the closure indicates that *fib* takes a single argument, and that the closure does not contain any partially applied arguments. The body of the function itself is named `Main$fib$P`. The entry point, `EntryPt`, is passed using the C calling conventions. Every Eager Haskell function which uses a non-zero entry point starts by checking `EntryPt`. Note the `untaken` annotation, indicating that `EntryPt` is expected to be zero. The *fib* function uses two entry points, zero (combined with the wrapper code for the function itself, and thus shown in the figure) and three. If additional entry points are used, the body of the `if` contains

```
static const PHOb Main$fib[2] = { MkTag(1, 1), &Main$fib$P };

static PHFunc(Main$fib$P) {
    if (untaken(EntryPt != 0)) {
        goto Restore3;
    }
    {
        PHPtr $3, x$1 ;     /* C locals for continuation and argument */
        /* The beginning of the function is a valid suspension point. */
        static const PHOb SuspDesc0[4] = { MkTag(UTagTag, 3),
                                           MkTag(1, 5),
                                           &Main$fib$P,
                                           0x2};  /* $3 is computed. */
      Restore0:
        x$1 = SP[0];        /* Restore argument from stack */
        $3 = SP[1];         /* Restore continuation from stack */
        if (0) {
          TailEntry:
            EntryPt = 0;
        }
        CheckStack(Main$fib);
        /* Lambda body */
        ...
    }
}
```

Figure 7-1: Code for the outermost part of the one-argument function *fib* in module *Main*. This is a slightly cleaned-up and commented version of actual compiler output.

a `switch` statement dispatching on `EntryPt`.

The Eager Haskell compiler keeps data in C variables, saving and restoring it from the shadow stack only when necessary. On entry to *fib*, the argument and the continuation must be restored from the stack. This happens immediately after the `Restore0` label. The beginning of the function is also the save point for the conditional in the *fib* function. A descriptor `SuspDesc` must be generated for every save point; it is this descriptor which is passed to the run-time system when suspension occurs.

Functions which call themselves tail recursively are particularly common in Eager Haskell. Self tail calls use a different calling convention: instead of pushing the arguments onto the stack and calling, the C locals and the stack are both updated directly. This allows the code for restoring arguments from the stack to be bypassed, and the tail call branches directly to `TailEntry`. `EntryPt` controls the behavior of code in unlifted bindings, and must therefore be reset to 0 to reflect the fact

```
static const PHOb Main$con_1529[3] = { (PHOb)MkTag(2,2),
                                       (PHOb)fieldTagCAF(MkTag(1,0)),
                                       (PHOb)fieldTagCAF(MkTag(1,0))};
...
    if (Reserve(4)) {
        Suspend();
        gc(4, SuspDesc0);
        goto Restore0;
    }
    con_1522$8 = HPat(4);
    InitTag(con_1522$8, MkTag(Empty, 2));
    /* , :: , */
    con_1522$8[1] = (PHOb)IntBox(1);
    con_1522$8[2] = (PHOb)e_1296$6;
    StoreCon(con_1522$8, MkTag(1,2));
```

Figure 7-2: Code for the static constant $[[]]$ and the dynamically-constructed tuple $(1,$ *e_1296*$)$, taken from the *Queens* benchmark.

that execution is continuing from the beginning of the function.

Note that the Eager Haskell compiler generates all the code associated with a particular construct in one go, even if some of that code is commonly executed from elsewhere. Thus `TailEntry`, variable restore code, and the like are all generated in place and wrapped in `if (0) {...}`. If (as in *fib*) the function does not call itself tail recursively, then `TailEntry` will be dead code and can be eliminated by the C compiler.

The final piece of code before the body of the *fib* function itself is the stack check. This implements the exception mechanism used to throttle execution of Eager Haskell programs. The stack pointer is compared to a threshold stored in memory. If the stack is too large, the run-time system creates a thunk for the call to *fib*; the static closure `Main$fib` includes all the necessary information for thunk creation. When an exception occurs the stack size is set to zero, causing all stack checks to fail as described in Section 4.7.

7.5 Constructors

Figure 7-2 shows examples of the code generated by the compiler for static and dynamic constructor expressions. A function closure is essentially a static constant, and unsurprisingly the code for static list constant $[[]]$ in Figure 7-2 looks very similar to the static closure for *fib* in Figure 7-1.

Dynamic constructor expressions produce two code fragments. Code for allocation is batched as described in Section 7.3 (in the example shown it happens that only a single object is allocated); later code actually fills in the object fields. Allocation occurs in three steps. A call to `Reserve` sets aside the appropriate storage. If garbage collection is required, execution suspends and the garbage collector is invoked; after garbage collection execution is re-started from the preceding save point. Otherwise, the allocated objects are emptied. `InitTag` indicates that this is an initializing store; because the tag is `Empty`, there is no need to initialize the remaining fields in the newly-allocated object. Later, the fields of the object are filled in and the tag is immediately updated with `StoreCon`. This indicates that the object contains live data, and performs the necessary write barrier. As a result, the fields of the object must be completely initialized before `StoreCon` occurs. On a multiprocessor, the `StoreCon` operation must also enforce memory coherence guarantees as described in Chapter 11.

Recall from Section 6.2 that Eager Haskell stores small *Int* constants and nullary constructors in two static tables. Figure 7-2 includes references to both of those tables: `fieldTagCAF` generates a reference to the nullary constructor table, and `IntBox` generates a reference to the table of *Int* constants.

Constructor expressions in tail position must update the continuation rather than updating a newly-allocated value. If the constructor is small enough to fit in a continuation, the generated code is identical; the only difference is that the continuation does not need to be allocated. If the constructor is large, a new object is allocated and emptied. The continuation is updated with an indirection pointing to the still-empty object. The new object is later filled in as usual.

7.6 Function application

Recall from Section 5.5 that the Eager Haskell implementation shifts the complexity of curried function application into the run-time system. Compiling a function application is therefore reasonably straightforward. We make a distinction between known, full-arity applications and other applications, and between ordinary calls and tail calls. Because self tail calls are handled specially, this leads to five cases, three of which are shown in Figure 7-3.

An ordinary full-arity function application must save live variables to the stack, push the function arguments and continuation, and then call the function. On return the live variables are restored from the stack once again. The call to *fib* shown in Figure 7-3 happens to have four live variables;

```
PushVar(5);                    /* Make space for argument, live vars */
SP[2] = app_1235$6;            /* Store live variables */
SP[3] = prim_1240$7;
SP[4] = app_1236$8;
SP[5] = x$1;
SP[0] = prim_1239$5;           /* Store argument */
SP[1] = app_1235$6;            /* Store continuation */
/* Non-tail call */
app_1235$6 = CallOb(Main$fib$P(0));
prim_1240$7 = SP[1];           /* Restore live variables */
app_1236$8 = SP[2];
x$1 = SP[3];
$3 = SP[4];

PopVar(2);                     /* Pop extra frame entries */
SP[0] = c2_992$2;              /* Store closure           */
SP[1] = TagBox(MkTag(1,0));    /* Store arguments         */
SP[2] = c3_994$1;
return GeneralApply(2);        /* Apply closure to two arguments */

SP[0] = prim_1540$6;           /* Update both arguments on stack */
SP[1] = es_1253$4;
l_1250$2 = es_1253$4;          /* Update local variables for arguments */
tl_1252$1 = prim_1540$6;
goto TailEntry;                /* Loop */
```

Figure 7-3: Three cases of function application. First, an ordinary call to the *fib* function seen in Figure 7-1. Second, a tail call to an unknown function *c2* with two arguments *c3* [ ]. Finally, a self-tail call from the internals of the *length* function.

a single argument is passed to the *fib* function. Functions are always called with the argument `0`, indicating that execution starts at the beginning of the function (recall that all other entrypoints exist for suspension; these are handled exclusively by the run-time system). Function results are returned using the C calling conventions; the `CallOb` wrapper macro exists to support experimentation with those conventions.

An unknown function application works nearly identically to an ordinary full-arity application. The only difference is that the closure is pushed on the top of the stack, and `GeneralApply` is called with the number of arguments being applied.

Tail calls work very similarly to ordinary calls. However, there is no need to save and restore live variables across the call; instead, the existing frame is popped and the tail-recursive arguments replace it. The result of the tail call is `return`ed directly to the caller. This makes it easy for the C

```
PHPtr prim_1540$6;
if (Reserve(2)) {
    Suspend();
    gc(2, SuspDesc0);
    goto Restore0;
}
ReservePrefetch(2);
prim_1540$6 = HPat(2);
InitTag(prim_1540$6, MkTag(Empty, 1));
/* (Untouched binding) */
StoreInt(prim_1540$6, (LoadInt(tl_1252$1)+ 1));

if (((LoadInt(n_1319$5)+ LoadInt(v_1321$9))==
     (LoadInt(m_1556$3)+ LoadInt(v_1555$2))))
{
    ...
} else {
    ...
}
```

Figure 7-4: Code for primitive expressions. The first fragment is the code for the binding *prim_1540* $=$ *tl_1252* $+ 1$ from the *length* function (the code also allocates space for the box). The second is the code for **case** $(n\_1319 + v\_1321) \equiv (m\_1556 + v\_1555)$ **of** ... from Queens.

compiler to identify the tail recursive call and transform it into a jump. We show a tail recursive call to `GeneralApply` in Figure 7-3.

As noted in Section 7.4, we handle self tail recursion specially. Figure 7-3 shows a tail-recursive call from inside the *length* function. The tail-recursive arguments are stored both into the stack and into the appropriate local variables. The code then branches to the `TailEntry`; there is no need to restore the arguments from the stack. The hope is that arguments which are already loaded into registers can remain there across the tail call, rather than being written to memory and immediately read back in.

7.7 Primitive expressions

The nested primitive expressions of λ_C translate to simple C expressions. Explicit synchronization (Section 6.9) insures that variable references within primitive expressions refer to fully computed values. Thus the variable references in Figure 7-4 simply fetch *Int* values from their boxes using the `LoadInt` operation. When a primitive expression is bound, we must box the result. The first code

fragment in Figure 7-4 allocates a box `prim_1540` to hold an *Int*. The `StoreInt` operation then writes an *Int* value into the now-allocated box. Similar boxing and unboxing operations exist for all primitive types.

When a primitive expression is the discriminant of a boolean **case**, the Eager Haskell compiler generates a direct conditional rather than boxing and unboxing the boolean value. This is shown in the second code fragment in Figure 7-4. This example also shows nested primitive expressions in action: two additions and an equality test are performed with no intervening boxing.

The Eager Haskell compiler does provide a second mechanism for performing primitive applications which is more similar to the function application mechanism. A primitive of n arguments is mapped to a C function taking a continuation argument and n boxed arguments and returning a boxed result. This provides an interface for various runtime routines which are written in C and manipulate full-fledged Eager Haskell objects rather than primitive values. For example, the vector routines used in implementing arrays use this convention; see Chapter 9. The naming phase of the compiler (Section 3.5.5) treats these primitives like ordinary function applications and names their arguments and results. They are otherwise treated like ordinary primitives—arguments are explicitly synchronized, and free variables need not be saved and restored across calls.

7.8 Case expressions

The simplest possible translation of **case** expressions in Eager Haskell transforms them into C `switch` statements, with one branch per **case** disjunct and a separate disjunct to handle synchronization. This is exactly how multi-disjunct synchronizing **case** expressions are compiled, as seen in the first code snippet in Figure 7-5. When a **case** expression mentions fields of the matched object, they are fetched on entry. The *length* function only uses the tail (and not the head) of its list argument; the code shown corresponds to the case disjunct $(\_ : \mathit{es\_1253}) \rightarrow \ldots$. The `HandleSuspend` macro handles suspension. The first two arguments are the entry point and descriptor of the appropriate save point. The third argument is the variable, *l_1250*, which is being synchronized upon. The final block of code is executed after the run-time system has suspended computation. It retrieves the continuation (saved with the rest of the live variables before the run-time system is called), pops the live variables, and returns.

The compiler produces specialized code for particular sorts of **case** expression. The special cases handle instances where a **case** expression has one disjunct, or has two disjuncts and is non-

```
TagSwitch(l_1250$2) {
  /* [] :: [] */
  TagCase(1,0):
    ...
    break;
  /* : :: [] */
  TagCase(2,2):
    es_1253$4 = asPtr(l_1250$2[2]);
    ...
    break;
  default:
    HandleSuspend( 0,
                   SuspDesc0,
                   l_1250$2,
                   {
                       $3 = SP[2];
                       PopVar(3);
                       PHSuspRet($3);
                       ReturnTag($3);
                   });
}


if (TagIs( fap_1127$2, 1, 2 )) {
    PHPtr cf_800$10;
    PHPtr r_801$9;
    cf_800$10 = asPtr(fap_1127$2[1]);
    r_801$9 = asPtr(fap_1127$2[2]);
    ...
} else {
    ...
}


HandleTouch( 0,
             SuspDesc0,
             tl_1252$1,
             {
                 $3 = SP[2];
                 PopVar(3);
                 PHSuspRet($3);
                 ReturnTag($3);
             });
```

Figure 7-5: Code for three **case** expressions. First, **case** *l_1250* from the *length* function. Second, a tag check for the already-computed *fap_1127* from the Counter program. Finally, the synchronization on the *Int* value *tl_1252* from the *length* function.

synchronizing. We have already seen in Figure 7-4 that primitive boolean **case** expressions are compiled directly into `if` statements. Two-disjunct, non-synchronizing **case** expressions need only check a single tag. They are therefore compiled into `if` statements, as seen in the second example in Figure 7-5. Single-disjunct non-synchronizing **case** expressions are trivial: No tag checking is required, and fields are simply fetched from the referenced object.

Single-disjunct synchronizing **case** expressions result from the insertion of explicit synchronization (Section 6.9); they are handled by the `HandleTouch` macro as shown in the final fragment in Figure 7-5. `HandleTouch` checks whether the provided object has been computed and suspends if it has not. The check is annotated to indicate that no suspension is usually required.

7.9 Suspensive bindings

Suspensive bindings introduce thread spawns into Eager Haskell programs. If computation of a binding suspends, subsequent code (the post-region) must nonetheless be executed. When the binding is resumed, the post-region must *not* be re-executed. This works by assigning an entry point number to the post-region which is strictly larger than any possible entry point in the spawned code itself, but strictly smaller than any prior entry point. For example, in Figure 7-6 the spawn is assigned entry point 1 and the spawned code includes entry point 2. When the spawned code suspends, it jumps to a special *spawn entry*, here labeled `Spawn1:`. When control leaves the spawned code—either normally or because of suspension—the `EntryPt` is compared to the entry point of the spawn. If it is larger (`EntryPt > 1`), execution began within the spawned thread; the post-region was run when the thread originally suspended. The entire frame is popped and control returns to the run-time system. If it is smaller, then execution began prior to the spawn, and control continues into the post-region.

The Eager Haskell compiler assumes that spawned code can suspend at any moment. Local variables must in a consistent state before the post-region is executed. As a result, the live variables of the post-region are saved to the stack before the spawn. After the spawned code has run, the saved variables are restored and the post-region is executed. Variables which are modified within the spawned code (usually the variable being bound such as *v_898* in the figure, and any live variables which are synchronized upon) are *not* ordinarily restored after the spawned code is run; their newly-computed values are used instead. However, if the spawned code suspends these variables may be in an inconsistent state; as a result the spawn entry includes code to restore them.

```
/* Thread spawn */
PushVar(2);                 /* Allocate space for live variables */
SP[0] = v_898$5;            /* Save variables for post-region    */
SP[1] = app_1537$6;
SP[2] = l_1052$7;
{
  ...
    default:
    HandleSuspend( 2,
                   SuspDesc2,
                   actualProgramArgs_1557$4,
                   {
                     PopVar(1);
                     goto Spawn1;  /* Continue with post-region */
                   });
  ...
}
if (0) {
  Spawn1:
    v_898$5 = SP[0];        /* Reload variable computed in thread   */
}
if (untaken((EntryPt > 1))) {
    /* Unspawn; we resumed within the spawned thread. */
    PopVar(6);              /* Pop live variables, return to run-time */
    ReturnCon((PHPtr)NULL);
}
app_1537$6 = SP[1];         /* Reload variables live in post-region */
l_1052$7 = SP[2];
e2_846$2 = SP[3];
e1_844$3 = SP[4];
$9 = SP[5];
/* Post-region */
...
```

Figure 7-6: Code for a thread spawn resulting from the binding *v_898* = **case** *actualProgamArgs* **of** . . . in Queens.

Chapter 8

Bottom Lifting: Handling Exceptional Behavior Eagerly

The eager strategies we presented in Chapter 4 have one important shortcoming: They do not yield the expected semantics for the Haskell *error* function. This is because *error*, when called, prints an error message on the screen and immediately terminates the program. For example, consider the following term:

let *unused* = *error* `"You␣used␣the␣unused␣variable"`
in *const* 7 *unused*

The lazy strategy will evaluate *const* 7 *unused* to obtain 7. An eager strategy might choose to evaluate *unused* eagerly, causing the program to terminate prematurely. Even worse, the hybrid strategy could result in either behavior, depending on whether fallback occurs. If we eagerly evaluate a call to *error*, we may terminate the program when the lazy strategy would have produced a result. In order to compile Haskell eagerly, we must identify error handlers and treat them specially.

In a safe language such as Haskell error checks are common; aggressive program transformation is necessary to produce efficient code in the face of such checking. At the same time, error handlers have stronger semantics than ordinary expressions. By identifying error checks at compile time, we can transform error checking code in ways that would not be permitted for other program constructs.

We discuss error handlers in the broader context of *divergence*. Divergent expressions will never yield a value, either because they signal an error or because they fail to terminate. We present a semantics for divergence using the same general techniques as the non-deterministic exceptions found in Glasgow Haskell [102, 81, 76]. These past efforts presented a large-step semantics for

divergence. Unfortunately, a large-step semantics makes no distinction between reduction rules and strategy. Because the details of reduction strategy are of paramount importance in designing the Eager Haskell implementation, we present a small-step semantics for divergence in Section 8.1. In Section 8.2 we describe how the various strategies can be extended to incorporate divergence. In the hybrid strategy used in Eager Haskell all reductions involving divergence can be performed inside the run-time system.

In order to compile divergent code, the Eager Haskell compiler uses a technique we call *bottom lifting*. Bottom lifting identifies error handlers and hoists them out of program code. Hoisting error checking in this way is beneficial even if error handling code is otherwise treated just like other program code. This is because many functions in Haskell consist of just a few lines of code. For example, the prelude provides simple accessor functions such as *head* on lists:

$$\begin{array}{ll} \mathit{head}\ (x : \mathit{xs}) & = \ x \\ \mathit{head}\ [\,] & = \ \mathit{error}\ \texttt{"head:␣null␣list"} \end{array}$$

In practice, it is more expensive to call *head* than it is to perform the computation directly. As a result, Haskell compilers perform extensive function inlining. What happens when we inline, say, (*head args*) in *print* (*fib* (*read* (*head args*)))?

$$\begin{array}{lll} \mathit{main} & = & \\ \quad \textbf{do}\ \mathit{args} & \leftarrow & \mathit{getArgs} \\ \qquad \mathit{print}\ (\mathit{fib}\ (\mathit{read}\ (\textbf{case}\ \mathit{args}\ \textbf{of} & & \\ \qquad\qquad & & (x : \mathit{xs}) \rightarrow x \\ \qquad\qquad & & [\,] \rightarrow \mathit{error}\ \texttt{"head:␣null␣list"}))) \end{array}$$

We have duplicated the error message. This turns out to be larger than the rest of the code we inlined!

There is a simple solution to this problem: don't inline the error handling code. We split the definition of *head* as follows:

$$\begin{array}{ll} \mathit{head}\ (x : \mathit{xs}) & = \ x \\ \mathit{head}\ [\,] & = \ \mathit{headError} \\ \mathit{headError} & = \ \mathit{error}\ \texttt{"head:␣null␣list"} \end{array}$$

We can inline *head* while leaving *headError* alone. We call this splitting bottom lifting because it extracts code which diverges—which is semantically equivalent to $\perp$—and lifts it into a separate procedure or thunk. A single error handler is shared by every occurrence of *head* in our programs.

$$
\begin{array}{rcll}
E & ::= & \ldots & \mid \uparrow(E) \\
N & ::= & \ldots & \mid \uparrow(E) \\
V & ::= & \ldots & \mid \uparrow(V) \\
\\
I_E[\,] & ::= & \ldots & \mid \uparrow(I_E[\,]) \\
S[\,] & ::= & \ldots & \mid \uparrow(S[\,])
\end{array}
$$

$$
\begin{array}{lcll}
S[\uparrow(e)] & \longrightarrow & \uparrow(e) & \bot_s \\
\textbf{case } x = e_1 \textbf{ of } \_ \rightarrow \uparrow(e) & \longrightarrow & \uparrow(e) & \bot_c \\
\uparrow(\uparrow(e)) & = & \uparrow(e) & \bot_e \\
\uparrow(\textbf{ letrec } bs \textbf{ in } e) & = & \textbf{letrec } bs \textbf{ in } \uparrow(e) & \bot_l \\
\uparrow(e) & \longrightarrow & e & \bot_d
\end{array}
$$

Figure 8-1: Syntax, contexts, conversions, and reductions for divergence.

We can spot divergent functions using a combination of strictness analysis and type information; the techniques used are described in Section 8.3. However, bottom lifting comes into its own when error handling code is part of a larger function. Consider this definition of *tail*:

```
tail (x : xs)          = xs
tail [ ]               =
  let file             = "PreludeList.hs"
  in let line          = 37
     in let column     = 0
        in matchError file line column
```

Here the entire nested **let** expression is devoted to error handling. We can use our semantics to enlarge the bottom region so that it encompasses the bindings for *file*, *line*, and *column*. The transformations enabled by distinguishing divergent expressions are discussed in Section 8.4.

8.1 Semantics of divergence

We distinguish a divergent expression e by marking it with an uparrow, $\uparrow(e)$. In Figure 8-1 we present the semantics of divergent terms. These semantics are actually fairly straightforward. When a term $\uparrow(e)$ occurs in a strict context $S[e]$, no progress can be made until e converges. Thus, when e diverges we can replace the whole context $S[\uparrow(e)]$ by $\uparrow(e)$ ($\bot_s$). We can push divergence markings through **let** and **letrec** as in $\bot_l$. Finally, $\bot_e$ is equivalent to $\bot_s$ when read as a left-to-right reduction; no semantic harm can result from *adding* redundant exception markings, however, and we therefore state it as a separate equivalence.

The definitions of strict contexts and instantiation contexts must be extended in order for instantiation and **case** hoisting to work as expected in the presence of divergence. We simply need to extend $I_E[\,]$ and $S[\,]$ to permit instantiation and hoisting past divergence. Note in particular that **case** $x = e_0$ **of** _ $\rightarrow \uparrow(e) = \uparrow($ **case** $x = e_0$ **of** _ $\rightarrow e)$ [η_l,σ_s,η_l reversed]; this rule is used in Section 8.4 to enlarge a region of divergent code.

Our semantics for divergence is non-deterministic. The rule $\perp_s$ can choose any one of several divergent primitive arguments. More subtly, rules such as σ_t reorder expressions and also permit more than one divergent outcome. Performing reductions according to the rules in Figure 8-1 *refines* the program [81]—possibly decreasing the number of exceptional outcomes which may result. If we allowed $\perp_s$ to be interpreted as an equivalence, we would discover that all divergent expressions are equivalent:

$$\begin{array}{rcll} \uparrow(e_0) & = & \uparrow(e_0) + \uparrow(e_1) & \perp_s \text{ (reversed)} \\ & = & \uparrow(e_1) & \perp_s \end{array}$$

This is a perfectly reasonable semantic view, but then *e* in $\uparrow(e)$ serves no useful purpose; $\uparrow(e) \equiv \perp$. In practice, we expect that *tail* $[\,]$ will not print "Division by zero", and we therefore expect some distinction to exist between different divergent terms. This does mean that refining reductions are non-confluent: $\uparrow(e_0) + \uparrow(e_1)$ can reduce to the incommensurable values $\uparrow(e_0)$ and $\uparrow(e_1)$.

8.1.1 The meaning of a divergent expression

We have not addressed one crucial question: What does $\uparrow(e)$ actually mean, and how does that meaning compare to the meaning of *e* itself? We can take two positions on this issue. The simplest view is that $\uparrow(e)$ is simply an *annotation*, which indicates that *e* will diverge when evaluated. Under this view, it is perfectly legal to erase the annotation ($\perp_d$); its only purpose is to guide our reduction strategy. This is the view currently taken by the Eager Haskell implementation.

Alternately, we can view $\uparrow(e)$ as propagating a special *exception value*. Such a value can be caught by the top-level *IO* computation. This approach is described in [102], where the exception-handling primitive is named *getException*. From a semantic standpoint, this is quite simple; we simply need to replace $\perp_d$ with the equivalences *getException* $\uparrow(e) =$ *return* \$! *Bad* \$! *e* and *getException v* = *return* (*OK v*). At the language level *getException* :: $a \rightarrow$ *IO ExVal* and *e* :: *ExVal*. The *IO* type of *getException* reflects its non-determinism; indeed, the semantics of [102] collect all exceptional outcomes as a set, and *getException* selects non-deterministically from this set. This avoids the need for refining reductions in the purely-functional part of the language. In order to

$h \bullet \langle x = \uparrow(e)\,;\, k\rangle\, r$	$\equiv$	$x = \uparrow(e),\ h \bullet \langle k\rangle\, r$	(suspend on bottom)
$x = e,\ h \bullet \langle y = S[x]\,;\, k\rangle\, r$	$\equiv$	$h \bullet \langle x = e\rangle\, \langle y = S[x]\,;\, k\rangle\, r$ $e \notin var \wedge e \notin V \wedge e \notin \uparrow(E)$	(force)
$x = \uparrow(e),\ h \bullet \langle y = S[x]\,;\, k\rangle\, r$	$\longrightarrow$	$x = \uparrow(e),\ h \bullet \langle y = x\,;\, k\rangle\, r$	(propagate bottom)
$x = \uparrow(e),\ h \bullet \langle y = x\rangle$	$\longrightarrow$	$x = e,\ h \bullet \langle y = x\rangle$	(diverge)

Figure 8-2: The hybrid reduction strategy in the presence of distinguished divergent terms.

obtain an acceptable equational semantics in the face of divergence, we must either equate divergent terms, or work with a collecting semantics of this sort.

Note that there is a disconnect here: For everyday equational reasoning, it is simplest if all divergent expressions are considered identical. The programmer making this assumption will, however, assume that any divergent behavior which is exhibited is related to the original program code. This is the reason we state in Section 8.1 that the *implementation* (and thus the underlying semantics) must distinguish different divergent terms.

Note that regardless of whether $\uparrow(e)$ represents divergence or an exception value, it is always safe to mark a non-terminating expression as divergent. In either case, we collapse away portions of the term which depend strictly upon the divergent expression. In the former case, we eventually erase the annotation; if we evaluate *e*, we get stuck. In the latter case, the divergence can be caught using *getException*, in which case *e* will be evaluated strictly and non-termination will once again result. The semantics given in Figure 8-1 are a direct reflection of the equivalences which hold in the presence of non-termination. These equivalences are outlined in various sources, including two works by Barendregt [22, 23].

8.2 Evaluation strategies for divergence

Adding divergent terms to λ_C only has a small effect on many of our reduction strategies. The strict and lazy strategies only require the erasure rule $\perp_d$; neither strategy requires special mechanisms for dealing with divergent terms. (Note, however, that if $\uparrow(e)$ is viewed as an exception mechanism as described in Section 8.1.1 then there is no $\perp_d$ rule, and $\perp_s$ reductions must be used to unwind execution to the nearest *getException*.) The hybrid strategy must be modified as shown in Figure 8-2. Divergent bindings are never evaluated directly; they always suspend immediately (suspend on

bottom). Ordinarily, a divergent binding cannot be forced (force). Instead, a computation which depends upon a divergent binding will diverge and this divergence must be propagated. We can justify the reduction rule (propagate bottom) given in the strategy using a combination of instantiation, $\perp_e$, and $\perp_s$. A divergent annotation is erased using $\perp_d$ only if we must do so to make progress. In this case, there will be only a single binding remaining on the stack (diverge).

8.3 Identifying bottom expressions

The most basic step in bottom lifting is identifying subexpressions of the program which diverge. Fortunately, there are several techniques which assist us in doing so.

8.3.1 Strictness information

Strictness analysis is the fundamental means of identifying error handling regions. A function *f* is strict in its argument if $f \ \perp = \perp$. Note, however, that functions which return $\perp$ have the special property that $f \ x = \perp$ for *any x*. Thus, divergence is the degenerate case of strictness analysis, where information about function arguments and free variables turns out to be irrelevant! Numerous techniques have been proposed for performing strictness analysis [92, Ch. 22], with widely varying tradeoffs between analysis speed, precision, and detail. For the purposes of bottom extraction, however, any reasonable strictness analysis is sufficient. In the Eager Haskell compiler, where boolean functions represent strictness, a divergent expression will have the constant function *False* as its strictness information. We need not evaluate the abstract function—we just simplify it (a necessary step in computing its fixed point).

Note that once we have performed bottom lifting, we need not perform strictness analysis on the bottom expressions again. Instead, we can use the results of bottom extraction to refine strictness analysis. In practice, this reduces analysis time slightly, but does not improve precision (since bottom extraction uses the same information base as strictness analysis does).

8.3.2 Type information

Consider for a moment the type of Haskell's *error* function:

$$error :: \forall\, a : String \rightarrow a$$

According to the type, the result of a call to *error* must be of *any and every* Haskell type. There is only one value that belongs to every type, and that value is $\perp$. This is the free theorem for *error*'s

type. There is a general class of free theorems [136] with exactly the same property:

$$x :: \forall\, \alpha : \tau_1 \to \tau_2 \to \ldots \tau_n \to \alpha,\ \alpha \notin \tau_i \quad \Longrightarrow \quad x\ e_1\ e_2 \ldots e_n \equiv \bot$$

In essence, if a function's result can have any type at all, the function will diverge when called.

Because Haskell already performs type inference, it is quite simple to check this condition. This test is effective: error handling code entails a call to the Haskell *error* function or its kin, so we will always encounter a function with an appropriate type.

In practice, we use type information to refine strictness analysis. The two techniques are complementary: types tells us certain expressions are bottom in the absence of strictness analysis, and strictness analysis propagates that information through the rest of the program. Thus, our strictness analysis phase performs two checks before it actually analyzes an expression: First, it skips analysis if it encounters an already-lifted bottom. Second, it skips analysis if type information indicates the expression is bottom.

8.3.3 Compiler assistance

The following definition of *tail* works identically to the one given earlier:

$$tail\ (x : xs) \qquad = \ xs$$

The compiler must insert an explicit error check when it is compiling the pattern match. The compiler can automatically tag the error-handling code as a bottom expression, or the error handler can be emitted in lifted form from the start. In either case, bottom lifting costs next to nothing.

8.4 Enlarging the lifted region

We would like to make the region we extract as large as possible. The Eager Haskell compiler does not explicitly remember a type or a strictness for every single subexpression in a program; propagating this much information and keeping it up to date is onerous. The analyses described so far are only useful for variable bindings, function applications, and variable references. In the code for *tail* given in the beginning of the chapter, only the call to *matchError* is identified as a bottom expression; this expression must be enlarged to include the **let** bindings which enclose it.

The semantics in Figure 8-1 provide the tools we need to enlarge regions of divergent code. The rule $\bot_l$ (applied in reverse) allows us to hoist divergence past **let** and **letrec**. For example, the

$$
\begin{array}{lll}
\uparrow(\ \textbf{case}\ x = e\ \textbf{of}\ [\,] \to e_0\,;\ z : zs \to e_1) & & \\
\quad = & \uparrow(\ \textbf{case}\ x = (\textbf{case}\ y = e\ \textbf{of}\ \_ \to y)\ \textbf{of}\ [\,] \to e_0\,;\ z : zs \to e_1) & \eta_l \\
\quad = & \textbf{case}\ y = e\ \textbf{of}\ \_ \to \uparrow(\ \textbf{case}\ x = y\ \textbf{of}\ [\,] \to e_0\,;\ z : zs \to e_1) & \sigma_s \\
\quad = & \begin{array}{l}\textbf{case}\ y = e\ \textbf{of} \\ [\,] \to \uparrow(\ \textbf{case}\ x = y\ \textbf{of}\ [\,] \to e_0\,;\ z : zs \to e_1) \\ v : vs \to \uparrow(\ \textbf{case}\ x = y\ \textbf{of}\ [\,] \to e_0\,;\ z : zs \to e_1)\end{array} & \eta_a,\ \chi_p \\
\quad = & \begin{array}{l}\textbf{case}\ y = e\ \textbf{of} \\ [\,] \to \uparrow(\ \textbf{case}\ x = y\ \textbf{of}\ [\,] \to e_0\ [y\ /\ x]\,;\ z : zs \to e_1\ [y\ /\ x]) \\ v : vs \to \uparrow(\ \textbf{case}\ x = y\ \textbf{of}\ [\,] \to e_0\ [y\ /\ x]\,;\ z : zs \to e_1\ [y\ /\ x])\end{array} & \iota_c \\
\quad = & \begin{array}{l}\textbf{case}\ y = e\ \textbf{of} \\ [\,] \to \uparrow(e_0\ [y\ /\ x]) \\ v : vs \to \uparrow(e_1\ [y\ /\ x]\ [v\ /\ z]\ [vs\ /\ zs])\end{array} & \iota_d,\ \chi_p,\ \chi_d \\
\quad \equiv & \begin{array}{l}\textbf{case}\ x = e\ \textbf{of} \\ [\,] \to \uparrow(e_0) \\ z : zs \to \uparrow(e_1)\end{array} & \alpha
\end{array}
$$

Figure 8-3: Hoisting divergence from a multi-disjunct case.

expressions in *tail* have the form **let** $i = e_0$ **in** $\uparrow(e)$ and are thus rewritten as $\uparrow(\ \textbf{let}\ i = e_0\ \textbf{in}\ e)$. The rule σ_s (again applied in reverse) allows us to hoist divergence out of a simple **case** expression; this can be extended to more complex **case** expressions as shown in Figure 8-3. The enlargement step takes 25 lines of code in the Eager Haskell compiler.

During enlargement the compiler also marks functions which return bottom expressions, making note of their arity. Such functions are error handlers which have already been factored out of the code by the programmer, or by previous compiler passes; we do not attempt to lift them again.

8.5 Lifting divergent terms

Lifting a divergent term is simply another form of lambda lifting (see Section 6.3). When the bottom lifting phase encounters an expression of the form $\uparrow(e)$, e is abstracted with respect to its free variables and hoisted to top level. The only special feature of bottom lifting is that it is performed *before* most optimizations, and can be repeated as often as necessary during compilation. In particular, bottom lifting occurs before inlining decisions are made; by lifting out divergent expressions, functions are made smaller and more easily inlineable. Early lifting does mean that divergent expressions will not be optimized based upon the context in which they occur. However, since divergent expressions

represent exceptional control flow, we actually disable most optimization in such code anyhow. Only *contracting* optimizations are performed in divergent regions.

8.6 Divergent computation at run time

After bottom lifting, all divergent expressions in the body of the program will have the form $\uparrow(x)$ or $\uparrow(f_k\ \vec{x}_k)$. This makes it particularly easy to represent such a computation. The marking on an expression of the form $\uparrow(x)$ can simply be dropped; the annotation is useful only to indicate to the optimizer that x diverges. The transformations described in Section 8.4 can be crafted to ensure that the binding for x itself has the form $\uparrow(f_k\ \vec{x}_k)$.

Expressions of the form $\uparrow(f_k\ \vec{x}_k)$ are simply a special form of thunk—the *bottom thunk*. They are represented and run in precisely the same way as ordinary thunks (Section 5.7). Like ordinary thunks, bottom thunks can be transformed into pseudo-constructors before code generation (Section 6.7).

The run-time system must perform the reductions in Figure 8-2. Forcing a divergent binding $x = \uparrow(f\ \vec{x}_k)$ always fails, resulting in suspension (suspend on bottom). If an attempt is made to force a suspension $y = S[x]$ which is dependent (directly or transitively) upon a divergent variable $x = \uparrow(f\ \vec{x}_k)$, the suspension is overwritten with an indirection to the divergent expression $y = x$ (propagate bottom). Similarly, when a thunk $y = x\ \vec{y}_k$ applies a divergent function $x = \uparrow(f\ \vec{x}_k)$, the thunk is also overwritten with an indirection $y = x$. Finally, if the result of forcing the outermost expression is divergent, we rewrite the tag of the bottom thunk so that it becomes an ordinary thunk (diverge). This thunk can then be forced as usual, causing the program to actually diverge.

8.7 Related work

The hybrid execution strategy in Eager Haskell requires the compiler to distinguish divergent expressions in some fashion. At its heart, however, bottom lifting embodies a basic principle of compiler construction: focus effort on the code which will be most frequently executed. The entire field of profile-based optimization is predicated on the notion that aggressive optimization should be restricted to procedures that are heavily executed. The technique we describe here uses obvious static information to weed out code which is unlikely to run.

We know of no work either on selective inlining of just the hot portions of a procedure body

or hoisting infrequently-executed code fragments. There are, however, a number of closely related branches of research that deserve attention. Program slicing has been used to improve the readability or debuggability of programs, and to identify related computations in order to parallelize them [141, 131]. A program slice is an executable fragment which contains only the computations necessary for producing a particular value. We imagine our bottom-lifted program as an executable fragment which produces meaningful results in the absence of errors in the input.

Procedure extraction hoists related computations into a separate procedure [44]. As with program slicing, extraction efforts are directed at improving the readability or debuggability of programs, and there is substantial crossover between the fields.

The transformations we perform on divergent expressions generalize and simplify optimizations which already existed in our compiler. Shivers describes a similar set of transformations to hoist dynamic type checks in scheme programs [122]. Similar concerns motivate special treatment of exception handling code in procedural languages. For example, the Jalapeño compiler for Java uses an intermediate representation where basic blocks continue past exceptional control flow [32].

Chapter 9

Implementing Lazy Arrays

In Haskell, the *Array* data type must be used in fairly restricted ways. An *Array* is constructed using the function *array* (or one of its variants), which is an ordinary Haskell function whose arguments are the array bounds and an *initializer*—a list of tuples $(index, value)$ specifying the contents of the array. These tuples are usually constructed using a list comprehension. Arrays are accessed using the indexing operator "!". Examples of both construction and indexing can be found in the code for the `wavefront` benchmark in Figure 9-1.

There are a few crucial constraints on array comprehensions such as the one shown in the figure. First, arrays are created *in bulk*—the array is allocated and its contents are filled in using a single operation. Second, arrays are bounds-checked. This is not in itself surprising; bounds-checked arrays are an integral part of a modern strongly-typed programming language.

However, bounds checking interacts in a very unusual way with bulk construction. What if one of the initializer elements has an out-of-bounds index? Haskell requires that the result of *array* be undefined in this case. In this way, if an array has an out-of-bounds initializer, the error can be caught and signaled when the array is used. However, this means that an array is undefined until

```
wave                          :: Int → Array (Int, Int) Float
wave n                        =  a
  where a                     =
        array ((1, 1), (n, n))
          ([((1, j), 1.0)     |  j ← [1 .. n]]++
           [((i, 1), 2.0)     |  i ← [2 .. n]]++
           [((i, j), (a!(i − 1, j) + a!(i, j − 1) + a!(i − 1, j − 1)) / 3.0)
                              |  i ← [2 .. n], j ← [2 .. n]])
```

Figure 9-1: The `wavefront` benchmark

every single index has been computed and bounds-checked.

This behavior is pronounced if we consider cases where the array is constructed non-strictly. In `wavefront` array elements depend on the values above them and to their left. This recursive dependence is valid in Haskell because only values, and not indices, depend on other array elements. Johnsson [56] gives examples of graph operations which are more efficient if arrays are constructed *lazily*. This allows both values *and* indices to depend on the values of other array elements. Accessing an as-yet-undefined array element causes more of the initializer elements to be evaluated; an out-of-bounds initializer causes accesses to as-yet-undefined elements to fail, rather than forcing the entire array to be undefined.

Ironically, "lazy" arrays in the style of Johnsson are actually *better* suited to eager evaluation. We can evaluate both indices and values eagerly, filling in array elements as we go. If the initializer is written in an order which respects dependencies between array elements, there is no need for array construction to ever suspend. In the `wavefront` example, all elements depend on previously-defined elements (if we read the initializer from left to right), and thus the result can be built eagerly without suspending. With Haskell-style arrays computation of element values must suspend until the array has been fully initialized.

Under Johnsson's semantics, array indices are still computed one at a time in the order in which they are specified—though that computation is now interleaved with other execution. In Eager Haskell we take Johnsson's lazy array semantics one step further, by providing lazy, order-independent arrays. An array element obeys I-structure semantics: it may be either empty or full, and if it is empty any accessing computation will suspend until it becomes full. Indices for initializers may be computed in any order; array elements are filled in as their indices are computed. If at any time an index overlap occurs the current array operation fails.

In practice, initializers are evaluated eagerly; if a particular index computation suspends, it is deferred and re-tried later. Ordinarily, we expect the programmer will write initializers which are properly ordered, and suspension should not be necessary. However, consider what happens if fallback occurs while an array is being constructed, causing array elements to remain undefined. When we access such an element, it will not contain a suspension pointing to its contents. The suspended initialization cannot be associated with a particular element of the array, and risks being thrown away. There needs to be a mechanism for associating initialization computations with the array as a whole. As we shall see in the remainder of this chapter, given the need for such a mechanism it is not difficult to provide the very flexible ordering of initialization we desire.

```
data (Ix a) ⇒ Array a b        =  MkArray a a (Vector b) SignalPool

array (l, u) ivs               =  MkArray l u (iVectorToVector v) pool
    where size                 =  rangeSize (l, u)
          v                    =  makeIVector size
          pool                 =  signalPool$
                                  map (λ(i, x) → iVStore v (index (l, u) i) x) ivs

MkArray l u v pool!i           =  vFetch v pool (index (l, u) i)

vFetch v pool i
    | slotFull v i             =  getIVector v i
    | poolEmpty pool           =  error "array element undefined."
    | otherwise                =  thunk vFetch v (forcePool pool) i
```

Figure 9-2: Implementing arrays using *SignalPool*s

The standard Haskell *Array* constructors can be defined using a monadic mechanism such as state transformers [63]. The State Transformer approach constructs a mutable array and fills its elements in using update operations. When all elements of the array have been initialized, the mutable array is "frozen", converting it into the immutable array which is visible to the rest of the program. We cannot easily use such mechanisms for lazy array construction; the final array is not available until every update is complete. We therefore need an entirely new mechanimsm—signal pools.

9.1 Signal Pools

In order to track suspended initializer computations, we define an abstraction, the *signal pool*, which can efficiently represent suspended initializers. A simplified implementation of arrays using this abstraction can be found in Figure 9-2. We make use of I-structure vectors. The *iVStore* primitive stores a value into the array, returning the void value $()$ once the array slot has been imperatively modified. The signal pool is constructed from the results of all the *iVStore* operations using the *signalPool* function. Every array includes a signal pool.

Elements are fetched by the *vFetch* function. We check if an element is properly defined using the *slotFull* function. If it is, we need not bother with the pool at all. If both the pool and the slot are empty, there are no outstanding writes and the array element is simply undefined. An error is signaled. Finally, if there are entries in the pool then the pool must be forced in the hopes that the

slot will be filled. We then attempt to fetch the array element once again.

Note that the list passed to *signalPool* is *not* an efficient representation. It contains *both* evaluated and unevaluated computations. Imagine all but two elements of an n-element array have been computed. If we access one of the undefined elements, we would need to traverse the initializer list, skipping over as many as $n - 2$ elements, before finally discovering a candidate to force. Meanwhile, the initializer list must be constructed and stored, more than doubling the space required by an array! Clearly, most of these entries are already $()$ and can be ignored.

We therefore require that signal pools take up minimal space. We add an entry to the pool only when an initializer computation suspends. Thus, we construct the pool by running the initializer computations. We check whether a given computation has completed; if it has not, we create a data structure which can be used to later force the computation. We collect these data structures together into the pool.

9.2 Using *seq*

Haskell already includes a function that does exactly what we want! The *seq* combinator evaluates its first argument, then returns the value of its second argument. The Eager Haskell compiler turns calls to *seq* into touch operations. If the touch operation succeeds, execution continues normally; if the touch operation fails, then a suspension is created. Thus, it seems as if we can create a signal pool simply by joining together computations with *seq*:

$$\begin{array}{ll} \textbf{type } \mathit{SignalPool} & = () \\ \mathit{signalPool} & = \mathit{foldr}\ \mathit{seq}\ () \end{array}$$

This yields a very efficient implementation of signal pools. The *signalPool* function can be inlined, and the *map* and *foldr* functions deforested [38, 70], so that the signal pool is computed directly rather than generating an intermediate list. Unfortunately, inlining the *seq* combinator makes computation overly strict. The compiler will happily transform the program so that subsequent calls to *iVStore* will not happen if a store operation suspends. This will cause deadlock if array accesses occur out of order:[1]

$$a = \mathit{array}\ (0,\ 2)\ [(a!0,\ 2),\ (a!2,\ 1),\ (2,\ 0)]$$

[1]Note that Johnsson's original conception of lazy arrays would prohibit this example as well—stores occur in initializer list order. The looser semantics we describe here are based on a desire to parallelize array generation, and on the fact that map/reduce deforestation can be allowed to change the order of list elements to transform recursion into iteration.

```
type SignalPool  =  ()
signalPool       =  foldr lastExp ()
poolEmpty        =  isWHNF
forcePool        =  try
```

Figure 9-3: Implementing *SignalPool*s

becomes:

```
...
  let x = vFetch a 0 in
  case iVStore a x 2 of _    →
  let y = vFetch a 2 in
  case iVStore a y 1 of _    →
  case iVStore a 2 0 of _    → ()
```

The final store will never occur—the computation of *x* will suspend, causing the entire signal tree to deadlock.

The cause of this problem is simple: the Eager Haskell compiler has aggressively optimized our program assuming it is side-effect-free. The internals of *array* have side effects. The solution to the problem is surprisingly simple—prevent the compiler from inlining *seq* by putting it inside a wrapper function *poolSeq*. Then we obtain:

```
...
  let x = vFetch a 0 in
  let u = iVStore a x 2 in
  let y = vFetch a 2 in
  let v = iVStore a y 1 in
  let w = iVStore a 2 0 in
  u `poolSeq` v `poolSeq` w `poolSeq` ()
```

Here every computation will initially suspend except the computation for *w*. The computation *w* `poolSeq` () will run to completion, yielding () and thus a minimal signal pool.

9.3 Fairness using *lastExp*

Unfortunately, this is still not quite enough. The problem is that the remaining pool elements must be forced in order. If we attempt to fetch either undefined element, *u* will be forced, forcing *x* in turn. Because *v* is still suspended, *x* will again attempt to force the pool, resulting in deadlock. We would prefer that the pool attempt to make progress on *all* of its elements. In order to do so, we

must control suspension much more carefully. We use the function *lastExp*. This function can be defined as follows:

$$
\begin{array}{lll}
\mathit{lastExp} & :: & a \to a \to a \\
\mathit{lastExp}\ a\ b \mid \mathit{isWHNF}\ a & = & b \\
\quad \mid \mathit{isWHNF}\ b & = & a \\
\quad \mid \mathit{otherwise} & = & \mathit{thunk}\ \mathit{lastExp}\ (\mathit{try}\ a)\ (\mathit{try}\ b)
\end{array}
$$

If either argument is defined, then we discard it and return the other one. This ensures that *lastExp* still yields a minimal signal tree. If neither argument is defined, we attempt to compute both arguments and suspend. The final implementation is shown in Figure 9-3.

Chapter 10

Results

In this chapter, we present measurements of various aspects of the Eager Haskell compiler. We use a selection of benchmarks culled from the the `nofib` Haskell program suite, plus a series of test programs used during the development of the *pH* and Eager Haskell compilers. We begin by detailing the benchmarks and their coding style (Section 10.1). We then present high-level run times for Eager Haskell and for Glasgow Haskell (Section 10.2). Most programs run more slowly under Eager Haskell than when compiled with the more mature GHC; however, the slowdown is usually not more than a factor of two. The exceptions either trigger compiler bugs or violate one or more of the assumptions made in the Eager Haskell implementation.

Succeeding sections look at particular aspects of compiled code. We begin with garbage collection (Section 10.3). We then look at function application (Section 10.4) and fallback behavior (Section 10.5). Finally, we look at the cost of suspension (Section 10.6) and variable forcing (Section 10.7).

The real promise of eager execution is that it eliminates the need to annotate programs (especially tail recursion) to control strictness. The multiplier benchmark is one example of a program which must be annotated in this way. We conclude the chapter with a case study of the multiplier benchmark, examining its behavior with and without annotation and optimization (Section 10.8).

Three kinds of numbers are presented in this chapter. Times are always in seconds. Ratios are generally presented as percentages, and marked as such. Event counts are listed in tables as the raw numbers collected from program runs. In graphs times and ratios are presented in the obvious manner. Most event counts are normalized with respect to mutator time; this allows us to perform side-by-side comparisons of programs with radically different run times and vastly different

Name	Abbrev	Source	Description	Lines	Code
Fib	fib	*pH*	recursive nfib, all ints between 1 and 37	35	10
clausify	claus	spectral	convert to clausal form, 7 times	188	87
fibheaps	fheap	spectral*	array fibheap versus sort, size 10000	286	89
Queens	queen	*pH*	n-queens, all ints between 1 and 12	55	26
queens	qu-no	imaginary	n-queens, problem size 10	14	9
Paraffins	para	*pH*	enumerate paraffins up to size 23	210	102
paraffins	p-no	imaginary	various paraffins stats, up to size 17	89	65
Primes	prime	new	sieve: every 100 primes through 50000	49	36
multiplier	mult	spectral	pipelined multiplier, 2000 cycles	503	289
Wave	wave	*pH*	float wavefront array relaxation, 200×200	46	20
MatrixMult	MM	*pH*	int matrix multiply, 100×100	52	22
Gamteb-acaro	gam	*pH*	Monte Carlo; uses trees, size 4000	702	518
gamteb	g-no	real	uses arrays, multiple source files, size 2048	702	526
symalg	sym	real	compute sqrt(3) to 15000 places.	1146	885
anna	anna	real	strictness analyzer, using “big.cor”.	9521	6383

Table 10.1: Benchmarks presented in this chapter. For comparison purposes, the Eager Haskell compiler itself has 32328 lines and 21522 lines of code.

proportions of time spent in garbage collection.

10.1 The benchmarks

The benchmarks used in studying the Eager Haskell compiler are summarized in Table 10.1. The *pH* benchmarks were used (sometimes in a slightly older form) to study the performance of the *pH* compiler; results for *pH* on a Sun multiprocessor can be found in Alejandro Caro’s Ph.D. thesis [31]. All these benchmarks use a single source file.

The remaining benchmarks are part of the nofib suite of Haskell programs [89]. These are roughly divided into three classes. The *imaginary* benchmarks are small, contrived programs written specifically as benchmarks. The *spectral* benchmarks are slightly larger (but still small) programs written for purposes other than benchmarking. The *real* benchmarks are large multi-file applications.

Several programs exist in both benchmark suites. The *pH* versions have generally been tweaked to deforest gracefully, or to eliminate trivial uses of infinite lists. The *pH* benchmarks also use larger problem sizes.

The benchmarks were chosen to provide a broad range of problems and coding style. Most notably, the *pH* benchmarks do not contain any infinite lazy computations and tend to be array-heavy. By contrast, many of the nofib-only benchmarks create large or infinite tree-like data structures.

10.1.1 Fib

The fib benchmark runs n-fibonacci (fibonacci with the recursive call $fib\ (x-1)+1+fib\ (x-2)$) for all *Ints* between 1 and 37. The program is little more than arithmetic and function calls, and uses no non-strictness whatsoever. We include fib as a measure of two aspects of the compiler: call overhead and slowdown due to the lack of unboxing.

10.1.2 Clausify

The clausify benchmark parses the logical expression $(a = a = a) = (a = a = a) = (a = a = a)$ and converts it to clausal form a seven times. This is an exercise in symbolic manipulation of tree-like data structures.

10.1.3 fibheaps

The fibheaps benchmark generates a list of 10000 random *Ints* and sorts them two ways: using the *sort* function from the prelude, and by constructing a fibonacci heap and then enumerating it. The *deleteMin* routine uses a small-scale array accumulation; the efficiency of this operation and of the built-in *sort* determine the overall speed of the benchmark.

10.1.4 Queens

The queens benchmark enumerates the solutions to the n-queens problem using a naive generate-and-test algorithm. The computation mainly takes the form of list comprehensions. Queens is used as a test of deforestation [38, 70]; most of the intermediate lists can be eliminated. This also permits shortcutting of control flow.

We evaluate two nearly-identical versions of queens. The *pH* queens is listed first in all figures. It has been tuned to use a deforestable intermediate list in the innermost loop. It enumerates all solutions with board sizes between 1 and 12. The nofib queens, listed second, enumerates only the 10x10 solutions. The innermost loop is written using recursion.

10.1.5 Paraffins

The paraffins benchmark [85] enumerates all the paraffins (molecules with the chemical formula C_nH_{2n+2}). This is a highly recursive problem: very large lists of partial solutions are constructed,

and once constructed each partial solution remains live for the remainder of the run. As a result, paraffins is extremely GC-intensive, even when a generational collector is used.[1]

Again, two nearly-identical versions of paraffins were evaluated. The *pH* paraffins, listed first, simply lists the number of paraffins of size n for each $n \leqslant 23$. The nofib paraffins, listed second, prints a number of statistics relating to counts of partial solutions in addition to solution counts. It only enumerates paraffins for which $n \leqslant 17$. Because of the combinatorial explosion in the number of solutions (and the amount of live data in the program), the *pH* version takes quite a bit longer to run. Larger problem sizes ($n > 23$) do not fit comfortably into physical memory.

10.1.6 Primes

A number of versions of the sieve of Eratosthenes have been written in Haskell, ranging in complexity from two lines to a couple of pages of code. Non-strict sieves produce their output list incrementally, and frequently the result is an infinite list of primes which is cut to size as needed. Because the sieve makes use of a long, persistent list, it has proven troublesome in the past for generational garbage collection [109].

The sieve used here prints the first of every 100 primes less than 50000. It has a couple of unique features. The list of candidate primes is bounded above by 50000. The actual sieving operation itself subtracts the infinite ordered list of multiples of the current prime from the list of candidate primes. Subtraction of ordered lists requires 7 lines of code; with this primitive, the sieve itself can be written in four lines. Because this version of the primes benchmark generates an infinite list for every prime number, it places a good deal of stress on the thunking and cutoff mechanisms in the Eager Haskell run-time system.

10.1.7 Multiplier

The multiplier benchmark performs a gate-level logic simulation of a clocked multiplier. This benchmark is a classic example of the lazy style, using infinite streams to represent wire values as they change over time. Only the outermost portion of the program limits list lengths in any way. We therefore expect it to pose a major challenge to the Eager Haskell execution mechanism. There are two dangers: on the one hand, if the compiler is aggressive in detecting infinite list creation, much of the run time will be spent forcing thunks. A lazy Haskell implementation is focused on

[1] A Java version of paraffins is used by Sun to benchmark GC performance.

making this efficient, whereas this has been a secondary concern in Eager Haskell. On the flip side, if such lists are not detected then they must be caught by the cutoff mechanism. This usually means that large structures are generated; if several are alive at once it is likely they will stress the garbage collector as well as the runtime.

The multiplier benchmark includes annotations intended to control function strictness. These annotations ensure that wire values are computed in time order. We have argued in Section 1.3 that eager execution should not require annotation for efficient execution. In Section 10.8 we examine this claim in more detail by comparing annotated and unannotated versions of the benchmark on larger problems.

10.1.8 Wavefront

The core of the wavefront benchmark is pictured in Figure 9-1. Wavefront represents the inner loops of an array relaxation algorithm. A 200×200 array of *Float*s is initialized non-strictly starting from the edges. This tests our ability to handle array non-strictness gracefully using the techniques from Chapter 9.

10.1.9 Matrix Multiply

The matrix multiply benchmark squares a 100×100 matrix of *Int*s. The naive textbook algorithm is used: each element is computed using a single inner product operation. The inner product is written using a deforestable list comprehension:

$$f\ i\ j = sum\ [a!(i,\ k) * b!(k,\ j)\ \ |\ \ k \leftarrow [1\,..\,n]]$$

Array elements in matrix multiply can all be computed independently. Thus, in contrast to wavefront the order in which the result array is computed (which may change due to fallback) should not matter.

10.1.10 Gamteb

Gamteb is a Monte Carlo photon transport simulation originally from Los Alamos. Heavy use is made of very large tuples and of arrays of *Double*s. Two very different versions of gamteb were evaluated.

The *pH* version of the benchmark (listed first), which uses a problem size of 4000, was obtained by automatically translating the Id benchmark into Haskell. The resulting code was consolidated

into a single module; this has a similar effect to performing whole-program optimization. This version of gamteb uses an array implementation based on complete binary tries. Under Eager Haskell, a version compiled using ordinary Haskell arrays either allocates too much memory at once (if function splitting is insufficiently aggressive) or crashes the C compiler (due to a bug in `gcc`, the C compiler claims to need a terabyte of heap in order to compile).

The nofib version of gamteb (listed second) uses a smaller problem size of 2048. This version of the benchmark appears to have been translated to Haskell by the original authors, and retains a neat division into thirteen source modules. This prevents both Eager Haskell and GHC from performing some of the most aggressive inlining optimizations which are possible with single-file compilation. One happy consequence of this fact, however, is that the Eager Haskell implementation can use arrays without overburdening the C compiler.

10.1.11 Symalg

The symalg benchmark computes the first 15000 digits of the square root of 3. The symalg program itself is designed to perform general arbitrary-precision calculations; it is the second-largest of the benchmarks (11 source files, containing slightly more code than gamteb). In symalg arbitrary-precision numbers are represented lazily using infinite binary trees of arbitrary-precision *Integer*s. This benchmark stresses the thunking and fallback mechanisms in Eager Haskell.

10.1.12 Anna

The final and largest benchmark, anna, reads in program code for a simplified Haskell-like language (using parsing combinators), type checks it, and performs a high-precision strictness analysis. We evaluate the largest of the test files, `big.cor`. Haskell is a nearly ideal language for the sort of heavy symbolic manipulation which occurs in all three stages of anna. Parsing and static analysis both involve a large number of higher-order function calls; this stresses the eval/apply mechanism used in Eager Haskell.

10.2 Eager Haskell versus GHC

Raw timings for benchmark runs are presented in Table 10.2, and graphically in Figure 10-1. All measurements were performed on 2-processor 466MHz Celeron PC with 384MB of RAM. The measurement listed is the median of 10 run times. Neither GHC nor Eager Haskell yet work on

	mutTime	gcTime	time	ghcTime	slowdown%
Fib	26.4555	2.7715	29.2145	15.1400	193.23%
clausify	0.6235	0.1060	0.7295	0.5500	140.00%
fibheaps	0.9585	0.3840	1.3435	0.7150	193.71%
Queens	19.6475	0.9435	20.5875	28.4900	72.41%
queens	0.2970	0.0560	0.3530	0.3200	112.50%
Paraffins	12.6560	28.5800	41.2465	22.6500	182.38%
paraffins	0.9120	1.3295	2.2435	2.0200	114.60%
Primes	35.9675	6.2990	42.2695	20.5850	205.56%
multiplier	4.3935	2.9690	7.3650	1.9250	383.12%
Wave	1.1310	1.4900	2.6190	0.1800	1486.11%
MatrixMult	1.4220	0.0530	1.4750	1.1500	131.30%
Gamteb-acaro	1.6740	0.6420	2.3155	2.2600	104.20%
gamteb	0.7465	0.1860	0.9315	1.1800	79.66%
symalg	0.5410	0.0190	0.5610	1.3200	43.18%
anna	4.0460	0.4630	4.5090	1.4650	308.19%

Table 10.2: Run times (in seconds) of benchmarks under Eager Haskell and GHC. Both user and system times are included. Eager Haskell run time is broken down in to mutator time and garbage collection time.

SMP machines; the main effect of the extra processor should be a decrease in system load from background processes. The machine was running Red Hat Linux 6.1, Linux kernel version 2.2.12-20. Both kernel and libc were recompiled to optimize for execution on i686. A single user was logged in to a text console to perform the timings; the X server and network interfaces were shut down for the duration.

The Eager Haskell compiler was run with optimization switched on (`-O`). The resulting C code was compiled with gcc version 3.0.1. A long list of flags was provided to gcc: `gcc -Os -march=i686 -fstrict-aliasing -fno-keep-static-consts -fschedule-insns2 -fforce-addr -freg-struct-return -fomit-frame-pointer`. Compilation with `-fomit-frame-pointer` allows the frame pointer register to be used as a shadow stack pointer. The `-Os` flag indicates that code should be optimized for space; this proved faster than options which optimize for speed. The `-fno-keep-static-consts` allows the compiler to get rid of unused info tables (such as the ones generated by the Eager Haskell compiler for unused suspension points). The remaining flags tweak minor aspects of program optimization.

The Eager Haskell run-time system was compiled to turn off all instrumentation except for internal timers. These internal timers make direct use of the real-time clock in the Celeron (`rdtsc`); timer calls therefore flush the processor pipeline but do not incur system call overhead. It is these

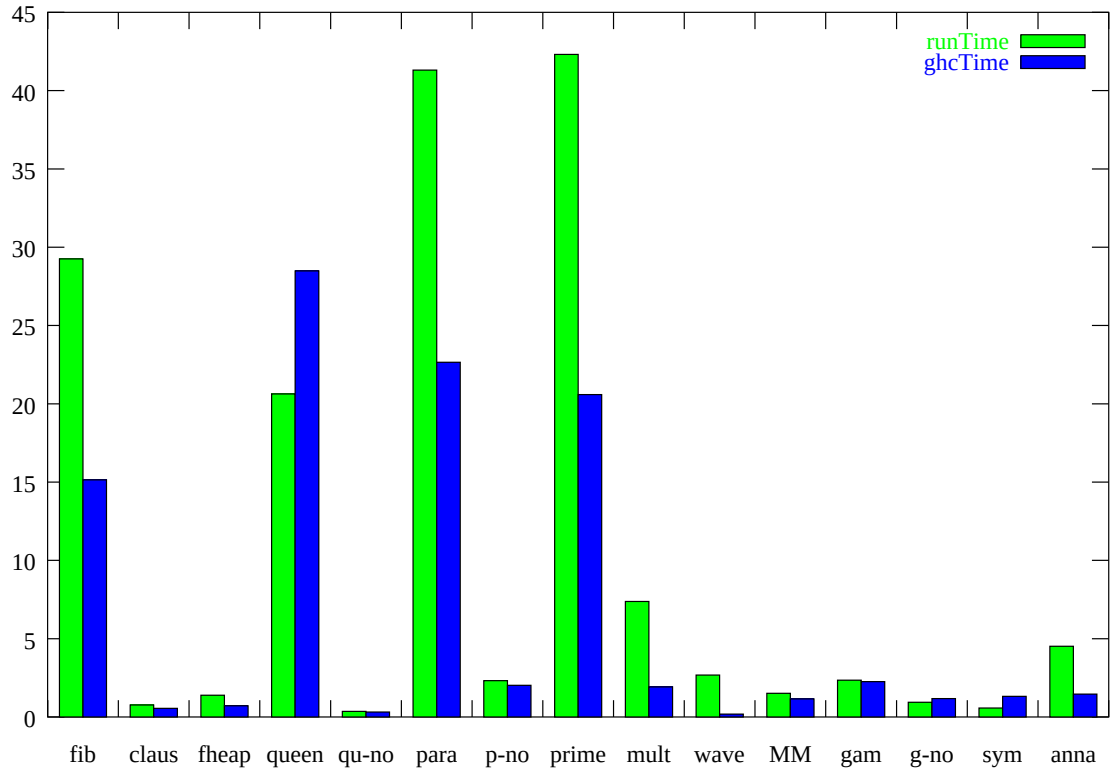


Figure 10-1: Run times of benchmarks under Eager Haskell and GHC.

times which are listed in Table 10.2. The run times reported by the shell are nearly identical (a tiny bit larger); the difference can be seen in Figure 10-2.

GHC was run at its maximum optimization level: `-O2 -fvia-C -optc-Os -optc-march=i686`. The same release of gcc, with the same major compilation flags, was used by both Eager Haskell and GHC. Table 10.2 reports run time as recorded by the shell; this is the sum of both user and system time.

Eager Haskell programs required about 60% more time than their lazy counterparts (this is the geometric mean of the slowdown across all benchmarks). This slowdown is shown graphically in Figure 10-2. These results are encouraging, as the Eager Haskell compiler has a number of notable shortcomings with respect to its more mature peer. For example, GHC compiles fib into straightforward recursion, passing and returning unboxed numbers. In Eager Haskell, by contrast, fib boxes all numbers, and contains a spawn (to preserve eagerness), resulting in multiple entry points. In spite of this, the Eager Haskell version of fib is only twice as slow as GHC.

For several programs, Eager Haskell actually produces faster code than GHC. Consider queens: in the *pH* version, where the innermost intermediate list is deforested, the Eager Haskell compiler runs faster. It is only 10% slower on the nofib version, whose inner loop is written recursively. This

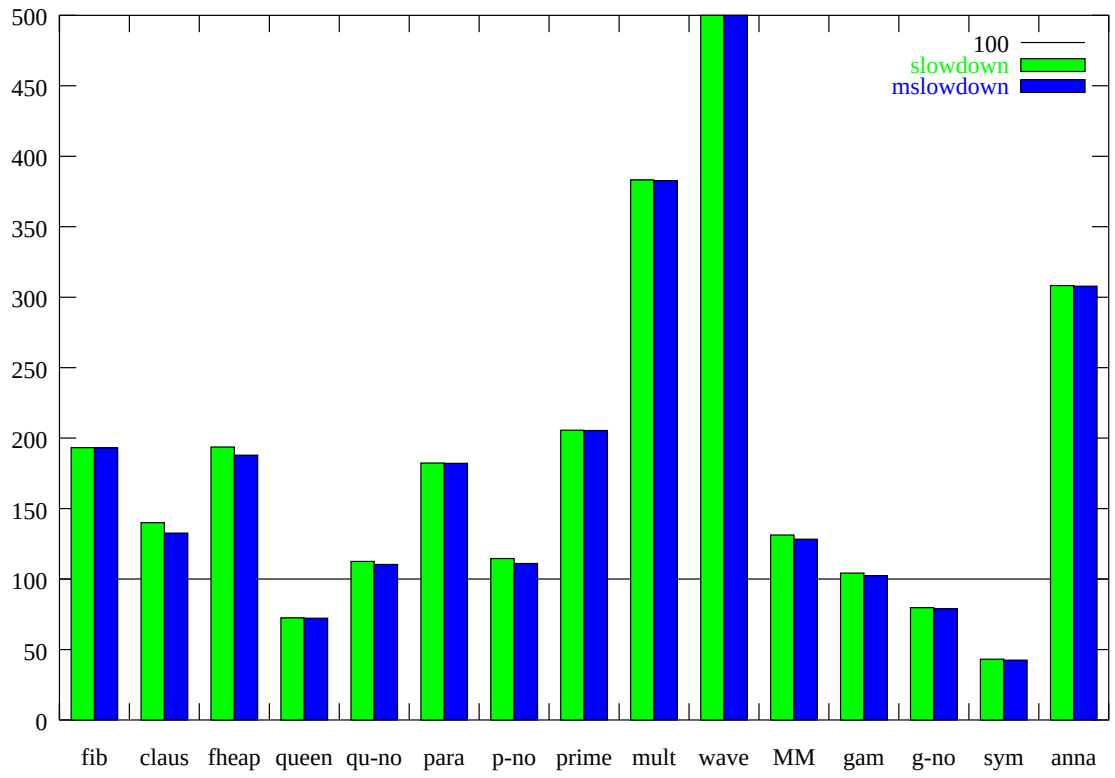


Figure 10-2: Slowdown of Eager Haskell compared to GHC, expressed as a percentage of GHC run time. Lower is better; less than 100% indicates a speedup. The wavefront benchmark was approximately 14.9 times slower than GHC; see Table 10.2. The two bars indicate OS and internal timer measurements for Eager Haskell; they are similar in all cases.

explicit recursion permits GHC to take advantage of unboxing.

A particular surprise is symalg, which constructs an infinite tree. This turns out to be structured ideally for the cutoff mechanism. A large tree is constructed eagerly; after cutoff, the result is plucked from it with little additional evaluation. As a result, nearly all of execution is spent in user code; the run-time system and GC are only rarely invoked. The result is fast execution.

The smaller array-intensive benchmarks generally work fairly well in Eager Haskell: gamteb is approximately the same speed in Eager Haskell and GHC, matrix multiply is only slightly slowed by the absence of unboxing, and the smaller run of paraffins (which uses much less storage) is only about 15% slower.

By contrast, Eager Haskell does very poorly on run-time system and GC-intensive code. Its most notable failure is wavefront, which is nearly 15 times slower! Indeed, if we ignore this benchmark our mean slowdown drops to 37%. The performance of wavefront is due to an unfortunate interaction of poor optimization and bad fallback behavior. In Figure 9-1 we can see that wavefront contains a non-strict dependency between each array element and its neighbors above and to the

left. The compiler breaks this dependency in the wrong place, preventing the array initialization from being deforested. As a result, a list of suspended elements is created, and the suspensions are then copied from the list into the array. Only then can they be forced, yielding the final result.

To make matters worse, array initialization falls back. This points to a major shortcoming of signal pools: they are too fair. Wavefront does best when its loops are executed in program order, from top to bottom and left to right. When an exception occurs in a signal pool, however, computation ordering is effectively inverted. The array is gradually filled with suspensions as signal pool computations attempt to run. After fallback, the last element of the array is demanded. This causes demand to ripple backwards towards the elements which have been computed. We speculate this problem will persist even if wavefront is properly deforested. Better scheduling of the signal pool and faster suspension and resumption are required to mitigate this overhead. The current signal pool implementation is done entirely using Haskell code similar to that shown in Figure 9-3; with GC and run-time assistance performance can be improved dramatically.

The poor performance of the multiplier and anna benchmarks is easier to explain. The multiplier benchmark contains large numbers of infinite lists generated by simple recursive functions. The compiler inserts thunks into such loops to avoid creating large amounts of trivially useless data. However, multiplier does use a good deal of data from every list. Thunks must be created and then forced again. The anna benchmark creates many closures and stresses the apply routine in the run-time system.

10.3 Garbage collection

Garbage collection times for Eager Haskell programs are listed along with the run times in Table 10.2. These times only count time spent copying, marking, and performing write barriers; they do not include the amount of time spent by the mutator allocating storage or checking for write barriers. The graph in Figure 10-3 shows GC time as a percentage of total run time. In general, Eager Haskell programs which outperform their lazy counterparts tend to spend very little time in the garbage collector. However, the reverse is not necessarily the case. We expect high GC overhead in wavefront and multiplier, reflecting additional allocation performed in the process of fallback and thunk forcing. We must look elsewhere for the sources of overhead in the remaining programs.

As expected, Paraffins is extremely GC-intensive; however, it is not dramatically slower than the other programs tested. However, for the larger problem size, GC time alone is larger than run

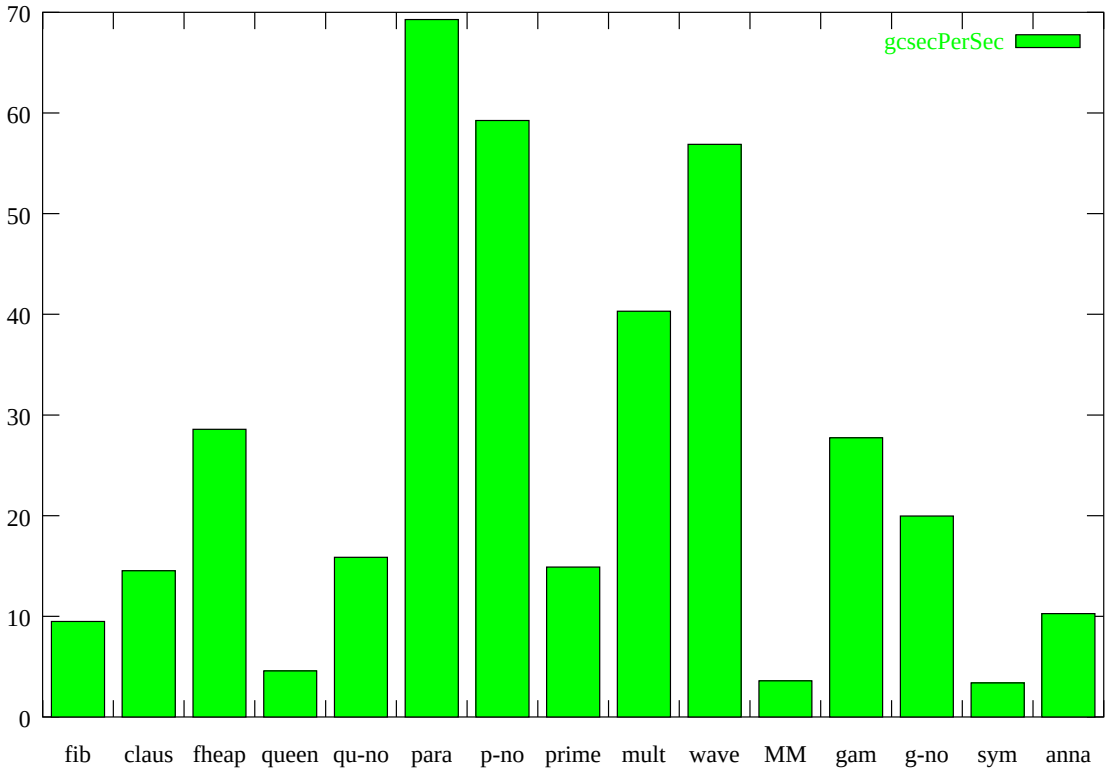


Figure 10-3: Percentage of total run time spent in garbage collector.

time under GHC. If paraffins performance is to match or exceed the same program compiled with GHC, GC performance must improve and mutator code must allocate less aggressively.

As noted in Section 7.3, the Eager Haskell compiler is probably too aggressive in batching allocation points. This results in a large number of write barriers, and may cause empty objects to be copied and promoted. In Table 10.3 and in Figures 10-4 and 10-5 we show the number of locations which are *checked* to see whether a tenured write has occurred, and the number of locations where such a write has actually happened and barrier indirections must be introduced.

No write barriers are necessary when non-pointer objects such as integers and nullary constructors are written. For this reason, the number of allocations and the number of write barriers executed are generally within a factor of two of each other (Table 10.3).

We distinguish between full-fledged write barriers and so-called *indirection* barriers. Indirection barriers occur when the object being promoted is already an indirection; in this case turning it into a barrier indirection is a simple matter of filling in extra fields and changing its tag. Though many indirection barrier checks occur in compiled code, actual indirection barriers are quite rare (Figures 10-4 and 10-5). The fields of non-indirections must be scanned and replaced with barrier indirections where necessary. This is shown in Figure 10-6. Very small numbers indicate that most

	writeBarrier	indirBarrier	barrier	actualWriteBarrier	actualIndirBarrier	actualBarrier	actualPerWrite%	actualPerIndir%	actualPerBar%	rootFou
Fib	169	102334228	102334397	1	0	1	0.59%	0%	0.00%	
clausify	1055107	760005	1815112	145	19	164	0.01%	0.00%	0.01%	
fibheaps	996678	990411	1987089	7541	5	7546	0.76%	0.00%	0.38%	
Queens	9107363	1226403	10333766	1	0	1	0.00%	0%	0.00%	
queens	36926	2035	38961	219	3	222	0.59%	0.15%	0.57%	
Paraffins	9618715	18827	9637542	18920	24	18944	0.20%	0.13%	0.20%	
paraffins	622013	4017	626030	1093	15	1108	0.18%	0.37%	0.18%	
Primes	23943839	158	23943997	98436	39	98475	0.41%	24.68%	0.41%	
multiplier	4473385	1178313	5651698	28724	5566	34290	0.64%	0.47%	0.61%	
Wave	451422	542358	993780	61886	736	62622	13.71%	0.14%	6.30%	
MatrixMult	23571	2051057	2074628	20513	9789	30302	87.03%	0.48%	1.46%	
Gamteb-acaro	517923	1318377	1836300	11929	20194	32123	2.30%	1.53%	1.75%	
gamteb	575479	429062	1004541	4398	8	4406	0.76%	0.00%	0.44%	
symalg	33273	15477	48750	14	1	15	0.04%	0.01%	0.03%	
anna	2543710	929753	3473463	3686	410	4096	0.14%	0.04%	0.12%	

Table 10.3: Write barrier behavior. The first two columns indicate dynamically executed write-barrier checks performed for writes to normal objects and to indirections. These are totaled in the “bar” column. The next three columns indicate how many of these barrier checks actually involved writes to tenured objects. The next three columns expresses the actual barriers as a percentage of barrier checks. The final three columns indicate the number of CAFs executed (these require special write-barrier-like treatment), the number of reserve (batched heap allocation) actions, and the ratio between write barrier checks and reservations. A large number in this column indicates that comparatively few allocated objects required a write barrier. A small number indicates many write barriers for each allocation action.

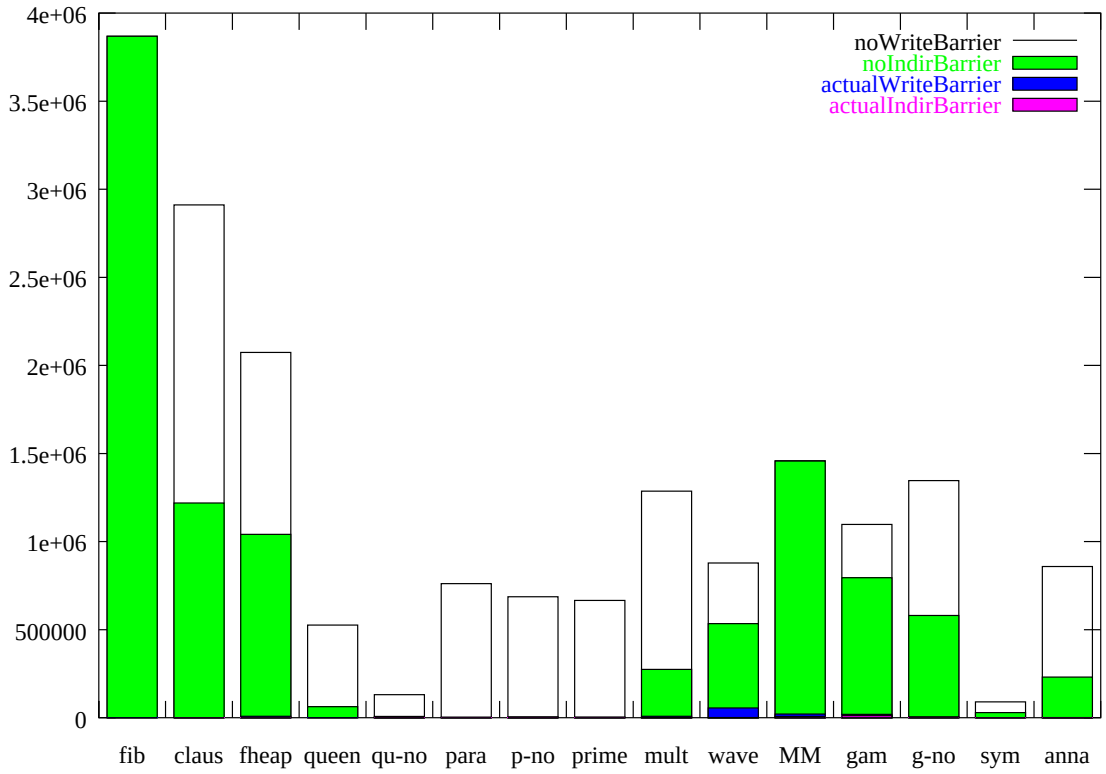


Figure 10-4: Number of write barrier checks, normalized to mutator time.

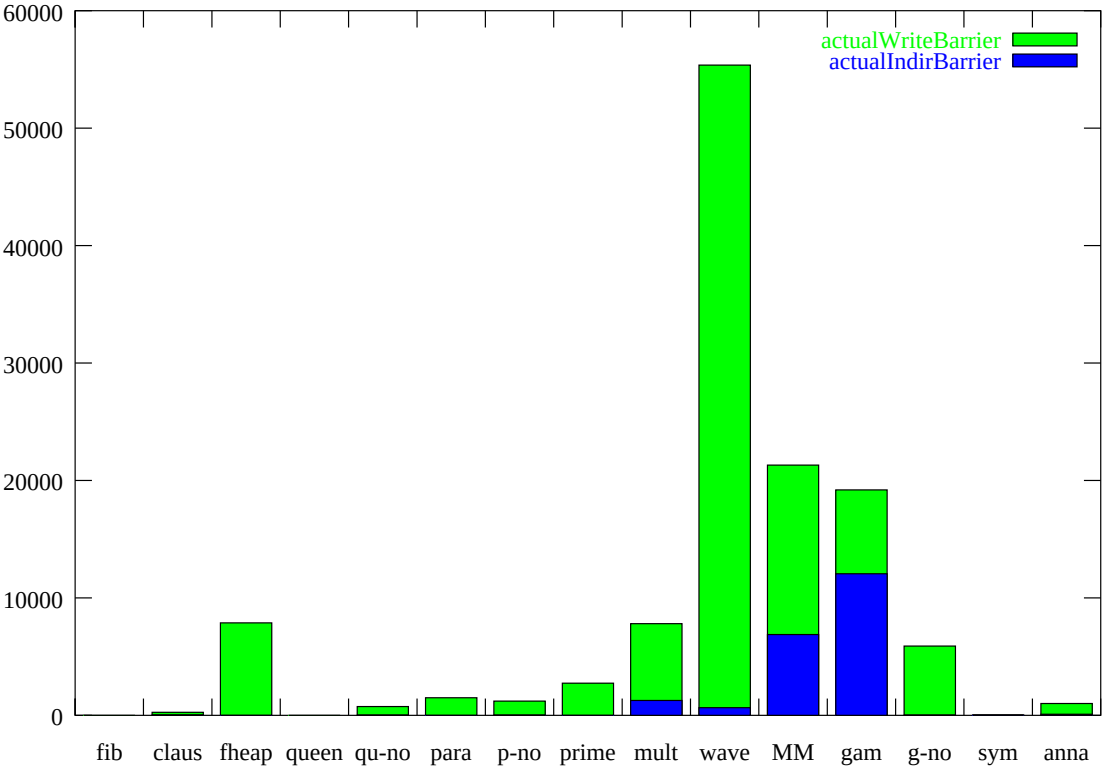


Figure 10-5: Actual number of write barriers triggered, normalized to mutator time.

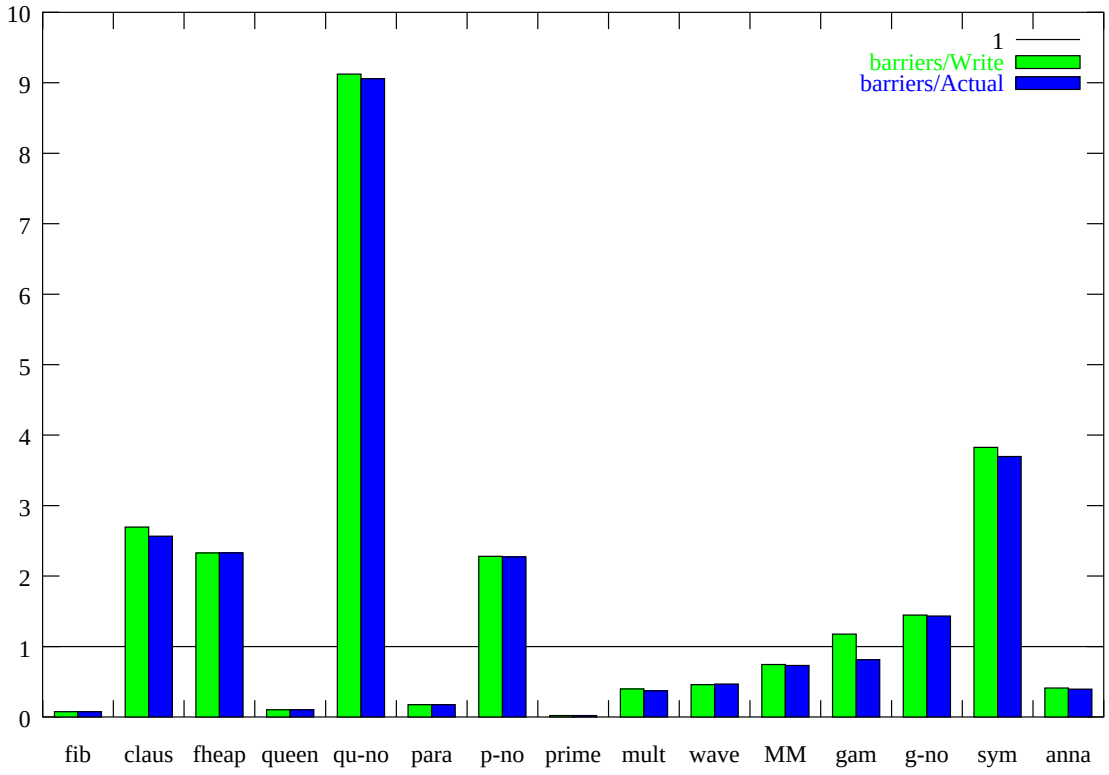


Figure 10-6: Barrier indirections created per write barrier, for non-indirection write barriers and for all write barriers.

write barriers refer to objects that need not be promoted—either because they have already been promoted themselves or as in Fib because they already reside in the static constant table.

Note that the nofib *queens* benchmark appears to promote a great many objects with a small number of write barriers. This is because suspension creation in the run-time system performs an implicit write barrier—the checking required has already occurred for other reasons. This case only occurs when empty data is promoted and then overwritten with a suspension. This is a relatively rare occurrence, and is obvious in this benchmark only because the number of write barriers is itself very small.

Actual write barriers are quite rare even at their most common (about 6% of all checks in wave-front, and less than 2% in all other cases). Note that the array-intensive codes perform many more write barriers. This is to be expected; objects in the large object area are assumed to be tenured, and will drag their contents into tenured space as well. Wavefront is tremendously write-barrier intensive, however. This additional traffic is again a result of the signal pool representation; *lastExp* discards and re-creates a thunk each time it is forced unsuccessfully, and each of these new thunks in turn must be promoted.

	regularCall	otherTail	selfTail	selfOver	runFrame
Fib	204668345	76	371	0	0
clausify	2540317	37760	367990	0	3483
fibheaps	1152932	544852	509845	16	24148
Queens	22381141	3173569	54430754	0	61316
queens	383703	69643	1092586	0	606
Paraffins	9598419	82742	9534415	4947	4556
paraffins	622159	12707	612795	424	266
Primes	13282703	108	6981517	4950	2359878
multiplier	4701923	2506001	723436	0	616953
Wave	373304	619	40108	0	39831
MatrixMult	10828	3	1030321	0	2237
Gamteb-acaro	1309731	208834	1597562	3	53726
gamteb	650120	228239	464566	2	6747
symalg	37918	15466	48511	28	64
anna	8242900	7051493	1793626	3	28922

Table 10.4: Function entry behavior. Raw numbers are given for the four types of function call that may occur in user code. The final column is function calls performed by the run-time system when resuming suspended computations.

10.4 Function Application

There are two basic ways an Eager Haskell function is entered: either it is called, or a suspension is resumed by the run-time system. We break down ordinary function calls into three categories (Section 7.6): self tail calls, other tail calls, and regular (non-tail) calls. Similarly, we break down resumptions into three categories: resumptions that re-start function execution from entry point 0, resumptions that re-start execution elsewhere in the function, but only have a single associated location, and resumptions that have multiple associated locations (these must be associated with a nonzero entry point).

Table 10.4 and the corresponding graph in Figure 10-7 show the breakdown of function entries. The *selfOver* statistic represents self-tail calls for long-lived loops. The *runFrame* statistic indicates function entries due to resumption of suspensions. Unsurprisingly, fib is the most call-intensive of the benchmarks. Both the queens benchmarks and the well-behaved array benchmarks (gamteb, matrixmult, even paraffins) make heavy use of self tail recursion.

The *GeneralApply* function handles unknown function application as outlined in Table 5.2. This function is called both by user code and by the thunk-forcing mechanism in the run-time system. Figure 10-8 distinguishes these two uses of application. The graph is normalized to the to-

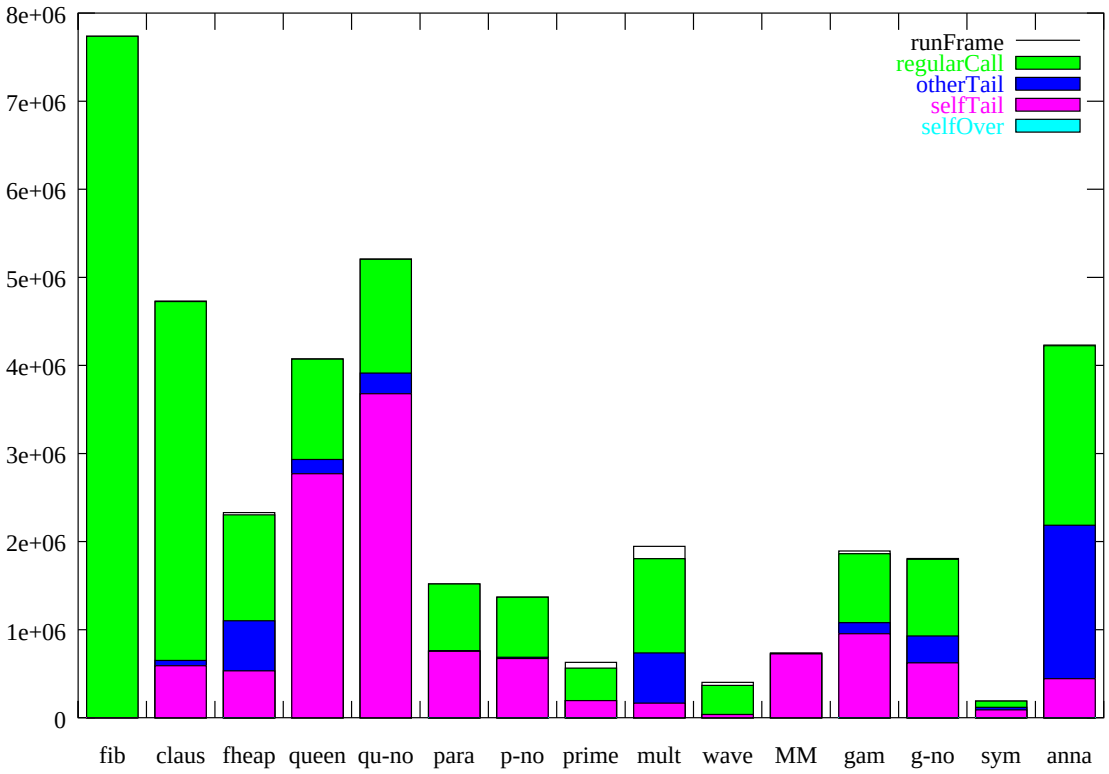


Figure 10-7: Function entries, normalized to mutator time.

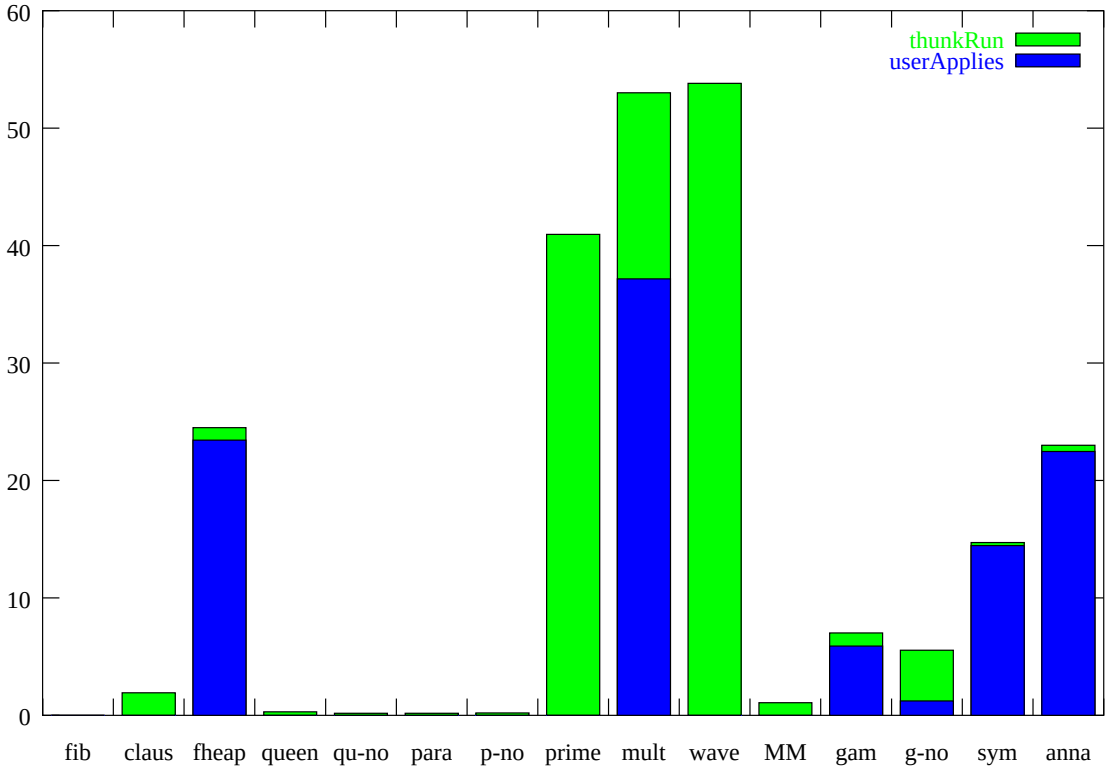


Figure 10-8: Entries to *GeneralApply*, as a percentage of all applications

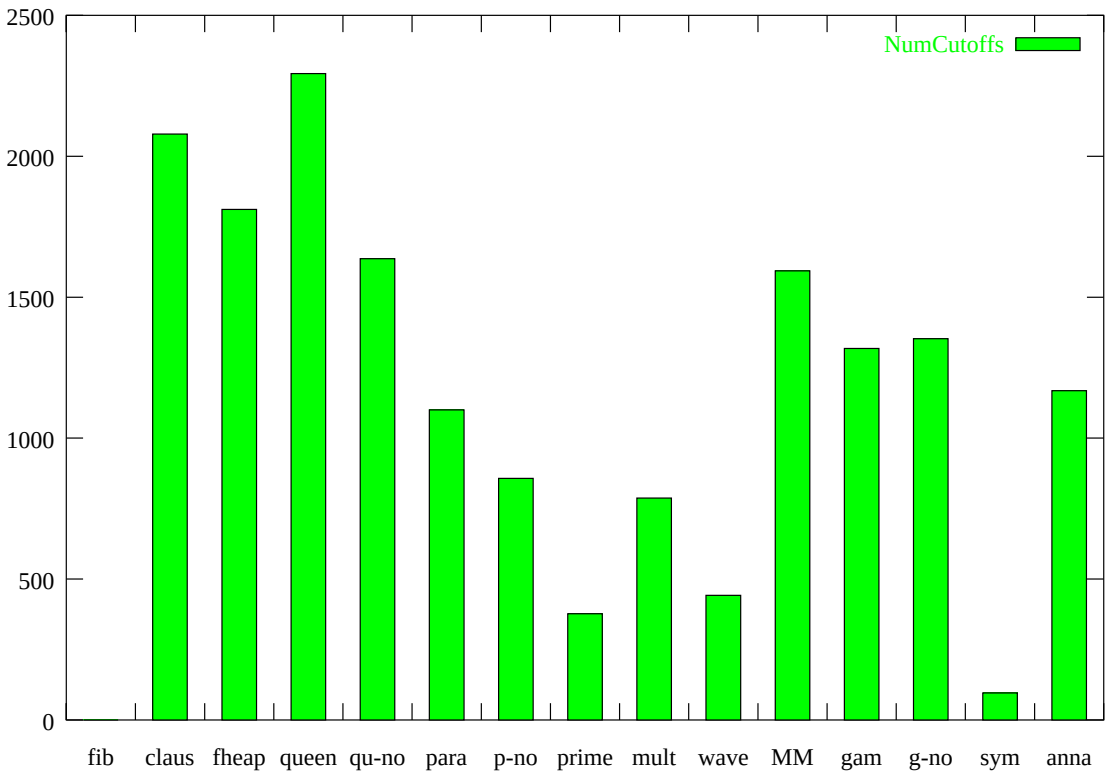


Figure 10-9: Fallbacks per second

tal number of function calls; thus the scale represents the proportion of all calls which pass through *GeneralApply*. Note the prevalence of thunks introduced by the compiler in primes and multiplier, and thunks introduced by fallback in signal pools in wavefront. Multiplier, anna, and fibheaps all include a large number of calls to unknown functions.

Nearly all general applications call a fully-computed closure at exactly full arity. The remaining cases account for only hundredths of a percent of calls to GeneralApply. Two benchmarks, anna and the nofib version of queens, occasionally pass an unevaluated closure to *GeneralApply*. These closures are forced successfully in every case. Over-saturated applications occur only in the anna benchmark. Interestingly, none of the benchmarks partially applies an unknown function. Closures are created only for known functions; this task occurs directly in user code.

Applying a partial application requires stack shuffling. Only one of the benchmarks—anna—makes extensive use of function closures. The cost of stack-shuffling is a likely reason for the poor performance of this benchmark in Eager Haskell.

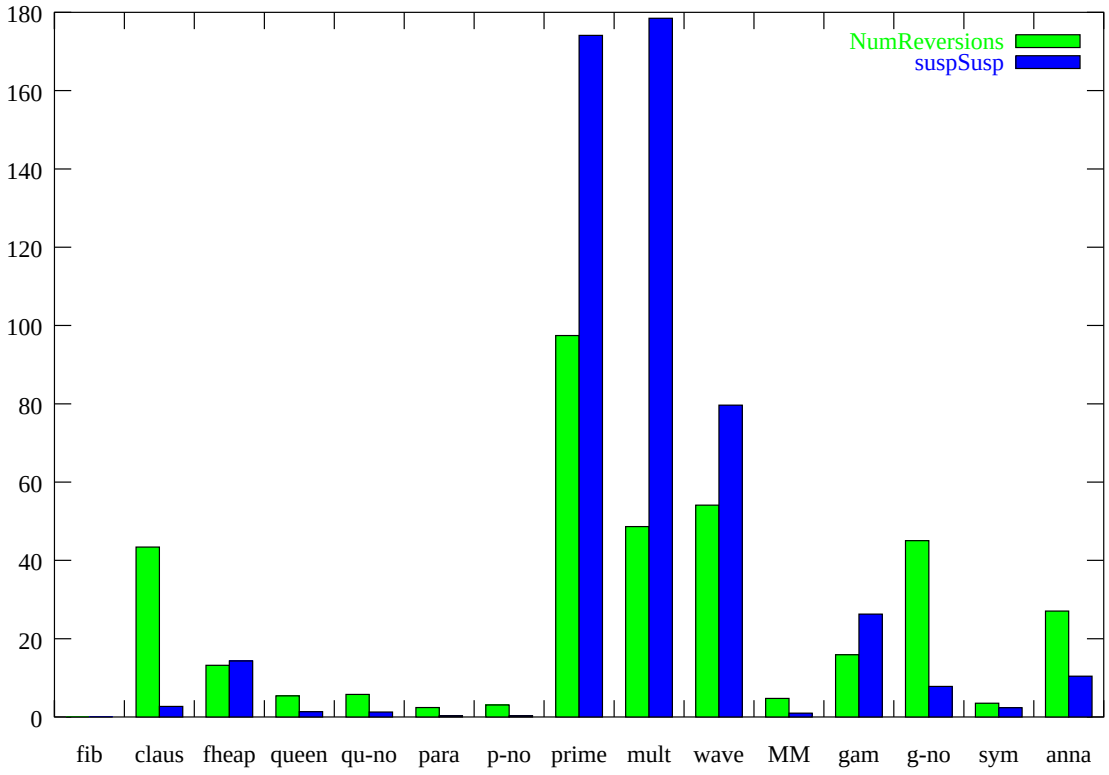


Figure 10-10: The consequences of fallback: thunks reverted and suspensions created, normalized to the number of fallbacks.

10.5 Fallback

When an exception occurs, the execution state is unwound. This has two consequences: function calls are reverted to thunks, and the computations which depend on them are suspended. Figure 10-9 shows the frequency of exceptions in the various benchmarks. In Figure 10-10 we examine the consequences of fallback, by normalizing the number of thunks and suspensions created to the number of fallbacks. This is only accurate for thunk reversion; there are plenty of possible causes for suspension, and they are not distinguished in the graph. Similarly, primes, multiplier and wavefront create large numbers of thunks as a direct result of execution rather than as a result of exceptions. The thunk forcing mechanism cannot distinguish these cases.

It is interesting to note that the thunk-intensive benchmarks also revert a larger number of function calls when exceptions occur. We can think of the height of the bars in Figure 10-10 as a rough measure of the call depth of each program. Benchmarks which create and traverse long lists frequently contain deep recursion, and consequently more thunk reversion occurs. This has another interesting effect: the extra time required to revert and re-start these thunks means that the rate of exceptions in these benchmarks is lower than normal.

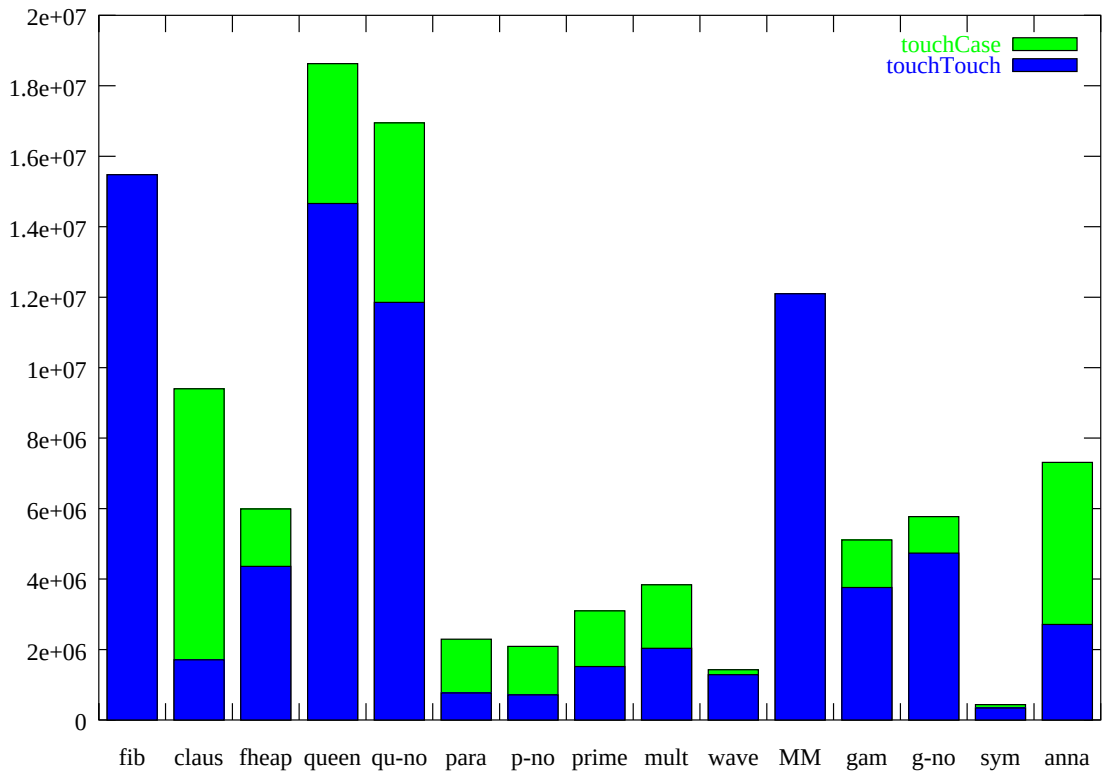


Figure 10-11: Touch operations (checks) normalized to mutator time. Broken down into freestanding touch operations and those combined with a **case** expression.

10.6 Suspension

When user code encounters a location which does not contain a value, that location must be passed to the run-time system. For sum types, this check—the touch—is combined with the **case** expression, but numeric and tuple types require a separate check. Figure 10-11 shows the rate at which checks are performed. Most of these checks succeed; only a small proportion fail and require forcing, as shown in Figure 10-12. Unsurprisingly, the benchmarks with the highest reversion rates also have the highest suspension rates (as high as 11% in multiplier). Most forcing operations yield a value, permitting execution to continue. The remainder (the lower part of the bars in Figure 10-12) require a new suspension to be created.

Once created, a suspension need not be forced. However, most suspensions are eventually forced, a fact that is evident in Figure 10-13. A notable exception (apart from symalg, which does very few touch operations) is anna. This indicates that a large number of unnecessary computations are being run eagerly. It is possible that more aggressive program transformations could eliminate this extra eagerness and thus avoid the unnecessary computation.

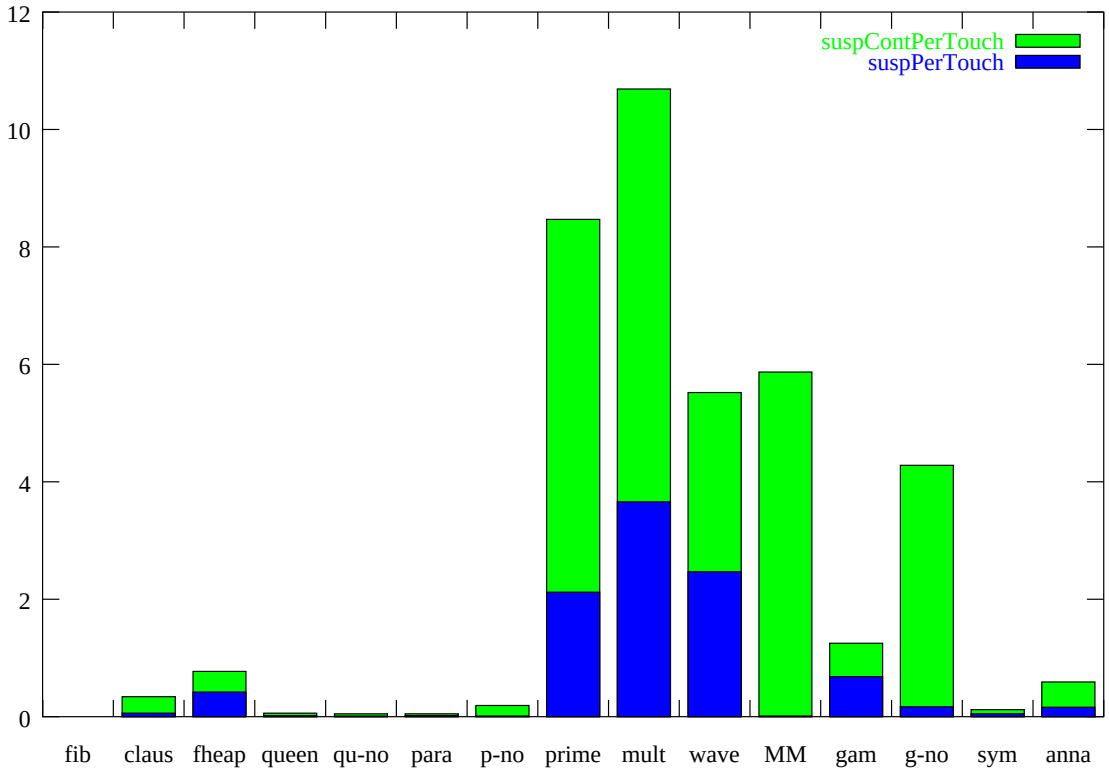


Figure 10-12: Percentage of touch operations which must invoke the run-time system to force a missing value. Lower bar indicates the proportion of such attempts which fail to obtain a value.

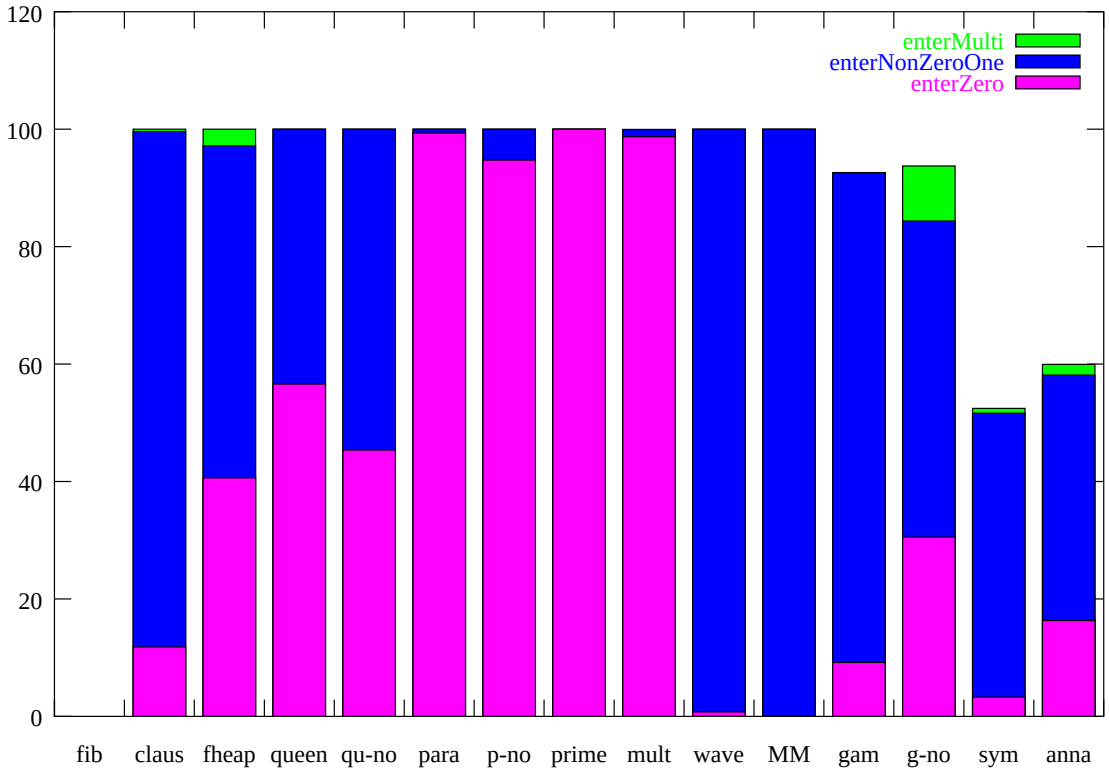


Figure 10-13: Function entries upon resumption, as a percentage of suspensions created.

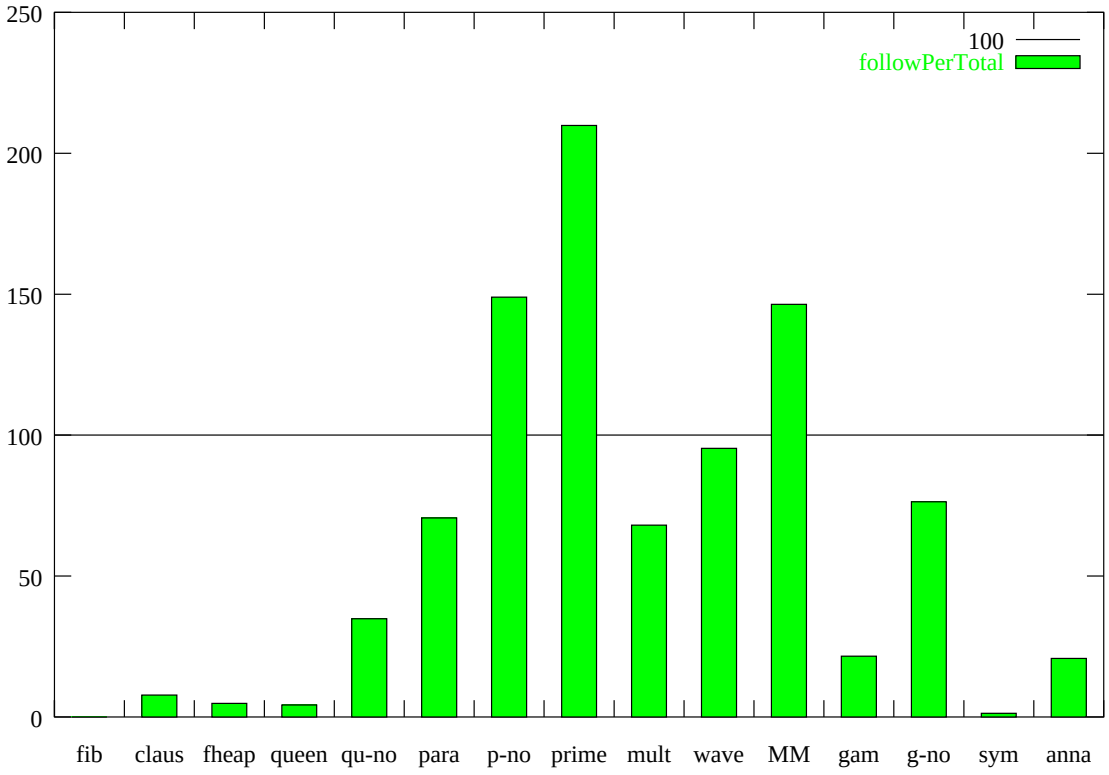


Figure 10-14: Percentage of indirections which are followed. Numbers greater than 100% indicate that indirections are followed more than once on average.

Figure 10-13 also indicates how resumed computations make use of suspension points. Resumption of suspensions is a rare enough occurrence compared to function call that it is lost in the noise in Figure 10-7. Multiple-result suspensions are even rarer and therefore represent only a tiny proportion of all function entries. The very complex machinery required for multiple-result suspensions could undoubtedly be simplified in the common case.

10.7 Forcing variables

What happens during forcing depends on what sort of object is encountered. This is broken down in two ways in Figure 10-15 and Figure 10-16.

Indirections are removed upon entry; in many cases this is sufficient, and the result is returned. Figure 10-14 shows the proportion of indirections which are dereferenced. Three benchmarks follow indirections more often than they create them: Primes, the nofib version of Paraffins, and Matrix Multiply. In all three cases, an enormous number of indirections are created as a result of tenuring. These indirections cannot be removed until the data they point to has been promoted. Already-

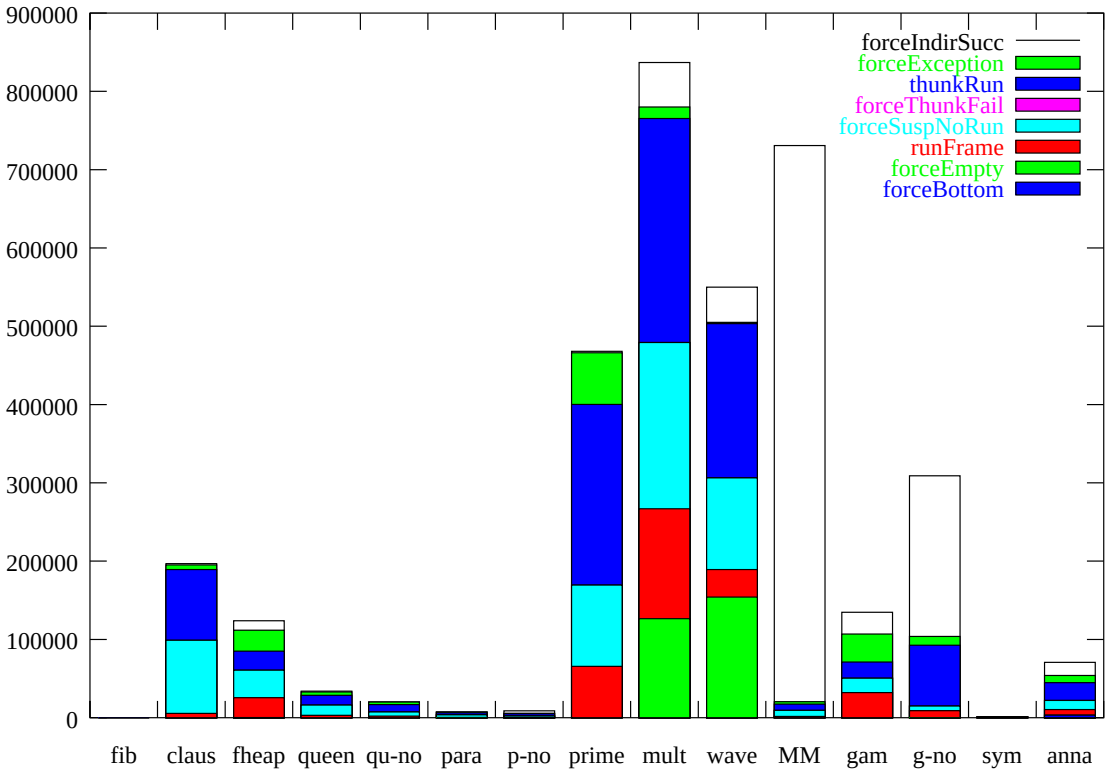


Figure 10-15: Breakdown of variables forced, normalized to mutator time.)

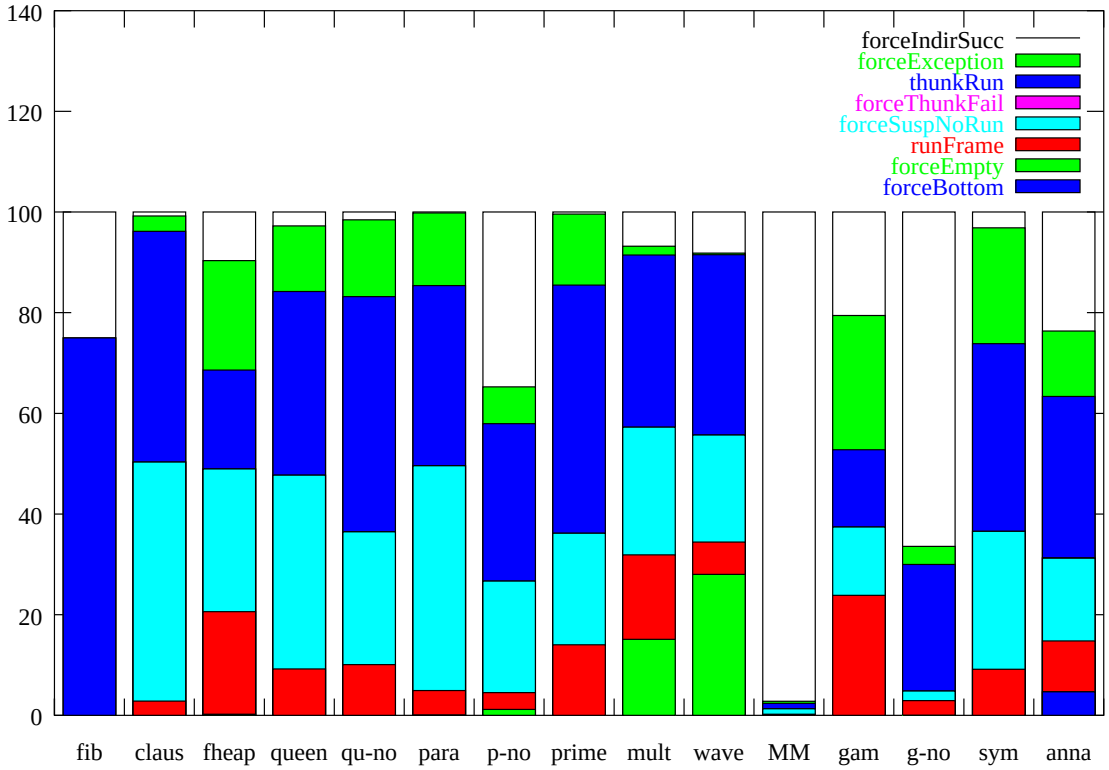


Figure 10-16: Breakdown of variables forced, as percentages of all force operations.

tenured data continues to refer to the indirections until the next full collection. In Matrix Multiply, most array accesses must follow an indirection as a result. In general, it appears that most indirections encountered in user code (and thus by force) reside in tenured space. This is not too surprising; when an indirection is created directly in user code, the pointed-to value is used as much as possible. Only when a write barrier introduces indirections unexpectedly does user code need to follow them. At the same time, the garbage collector eliminates nursery indirections; tenured indirections tend to persist much longer.

When an exception has been signaled, the forcing code removes indirections but does not run suspensions or thunks. As function calls fall back, we would expect their callers to synchronize on the results; such attempts will fail due to the pending exception. However, this seems to be relatively uncommon in most of the benchmarks, as indicated by the "forceException" bar in Figures 10-15 and 10-16. Most benchmarks force a comparatively small number of thunks during fallback; only in fibheaps, gamteb (*pH* version) and symalg are the number of force operations during fallback comparable to the number during regular execution.

Only one of our benchmarks, anna, attempts to force a bottom thunk. Such attempts always fail. Forcing of a bottom thunk is another good sign that the compiled code is excessively eager. Invalid data is used in some computation, and the results are then discarded. It is possible that code motion could eliminate the excess eagerness, improving performance and eliminating computations which yield bottom.

It is similarly uncommon for benchmarks to attempt to force a completely empty object. This occurs primarily as a result of algorithmic non-strictness: a computation refers to its result. This occurs in all three of paraffins, multiplier, and wavefront. Other benchmarks (such as primes) appear at the source level to use non-strictness in this fashion; however, program transformations can eliminate the dependencies.

Most commonly, we are attempting to force either a thunk or a suspension. Only anna ever attempts to force a thunk with an empty closure; these thunks can result only when an empty closure is passed to GeneralApply from user code. Even in anna such thunks are rare. Other thunks are simply forced; the results are then forced in turn if necessary.

The remaining bulk of forced objects are suspensions. Surprisingly, it appears that most suspensions which are forced do not get run. Repeated fallbacks create long chains of suspensions. As the chains are forced, shortcutting removes some elements from the stack; they must be re-examined later. Only the suspensions at the end of the chain are forced. If enough resources are required to

```
inv = lift11 forceBit f
  where f :: Bit → Bit
        f 0 = 1
        f 1 = 0
...
forceBit :: Bit → Bool
forceBit x = (x ≡ 0)
headstrict :: (a → Bool) → [a] → [a]
headstrict force [] = []
headstrict force xs = if force (head xs) then xs else xs
...
lift11 force f [] = []
lift11 force f (x : xs) = headstrict force (f x : lift11 force f xs)
```

Figure 10-17: Example of the original code for the multiplier benchmark. This code implements an inverter.

force them, another exception will occur and the chain will be unwound once more. We conjecture that the length of the suspension chain remains fairly steady for any given benchmark. The relative size of the "forceSuspNoRun" bar versus the "runFrame" bar gives an idea of the usual length of this chain for particular benchmarks. For example, clausify gives rise to long dependency chains. By contrast, the chains of dependent suspensions in gamteb tend to be very short.

10.8 Space-efficient recursion: the multiplier benchmark

We noted in Section 1.3 that the use of eagerness eliminates the need to annotate programs to obtain space-efficient tail recursion. The multiplier benchmark contains such annotations. Infinite streams of bit values represent the inputs and outputs of logic gates in the simulated circuit. Without strictness annotations, wire values can accumulate long chains of thunks over time. The code which implements logic gates is parameterized with respect to a strictness function, as shown in the inverter code in Figure 10-17.

The *headstrict* function used by *lift11* (and its brethren for wider logic gates) is parameterized with respect to an arbitrary strictness function *force*. Examination of the benchmark code reveals that *force* is always *forceBit*. It is straightforward to edit the program to eliminate the excess *force* argument from most function calls. At the same time, *forceBit* was simplified to use the Haskell *seq* function—the use of equality in the original benchmark works around the fact that early versions of

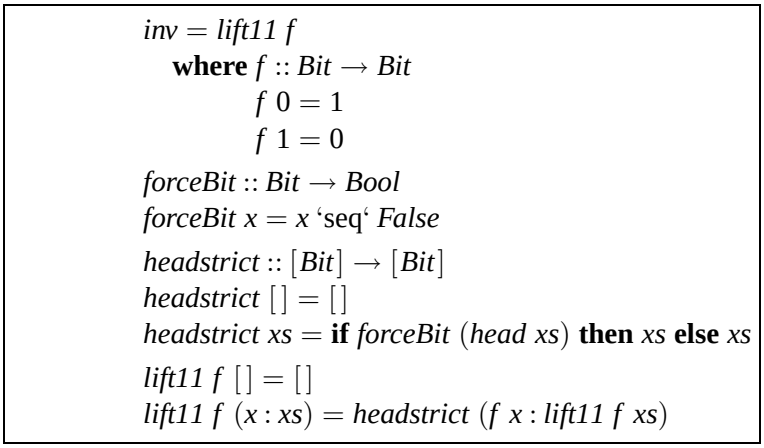


Figure 10-18: The equivalent code after strictness annotations have been propagated. In order to eliminate the annotations entirely, we simply need to change *headStrict* into the identity function.

Haskell did not include *seq*. The result is shown in Figure 10-18.

Using the re-written benchmark, it is very easy to eliminate strictness annotations from program code: simply replace *headStrict* with the identity function. This results in three versions of the multiplier benchmark: the *original* version shown in Figure 10-17, the *inlined* version of Figure 10-18, and the version with no strictness annotations, which we will call the *non-strict* version.

Under GHC we expect the inlined version of the benchmark to run the fastest. The uninlined version will be slowed down by the overhead of passing the *force* parameter from function to function. The non-strict version will suffer from the excessive cost of creating and forcing thunks for each wire value, and from garbage collecting those thunks. We expect the differences to be most pronounced for unoptimized code: since the wire values are simple integers, there is some hopes that strictness analysis and inlining will come to our aid.

Under Eager Haskell we expect the inlined and non-strict versions of the benchmark to run at comparable speeds. The inlined benchmark will perform additional computedness tests on its argument which are avoided by the non-strict benchmark. However, these tests should nearly always succeed, and we therefore expect the additional overhead to be slight. Optimization should work best on the non-strict code, where calls to *headStrict* can be inlined and eliminated. We do not expect to benefit nearly as much with optimization turned off, as *headStrict* will still impose function call overhead.

The actual measured times for the three benchmark versions are shown in Figure 10-19. Note that eliminating strictness annotations is no magic bullet: the Eager Haskell code remains much

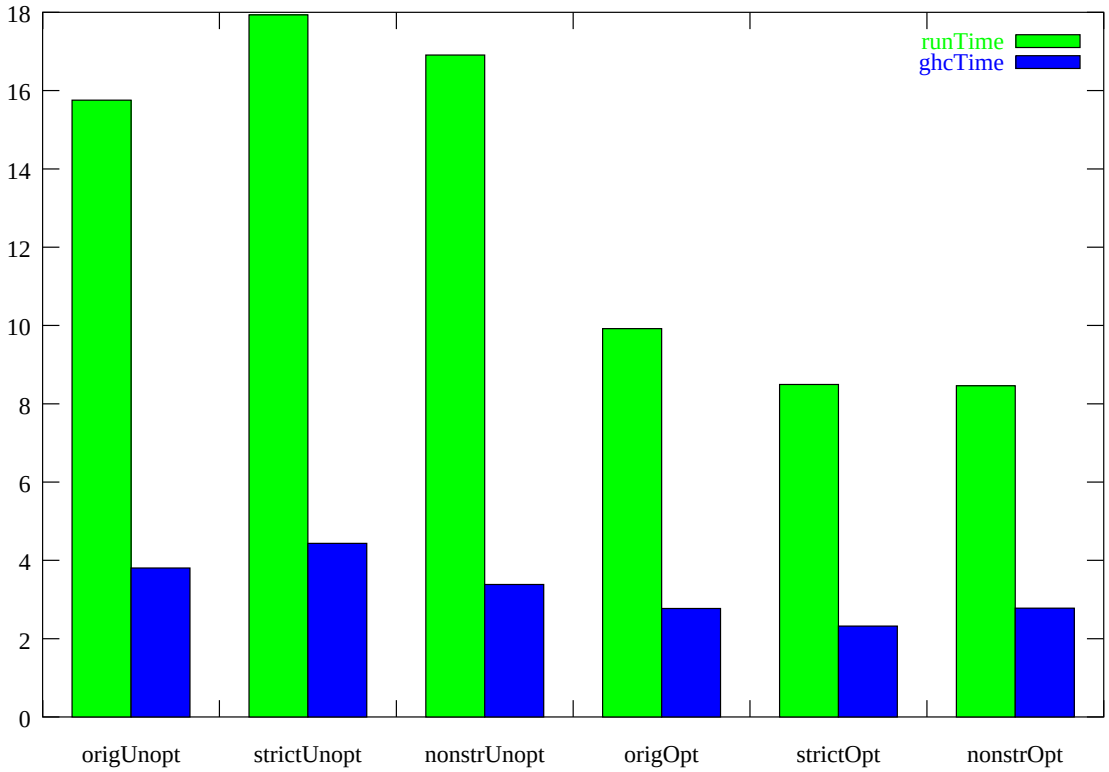


Figure 10-19: Run times of different versions of multiplier benchmark.

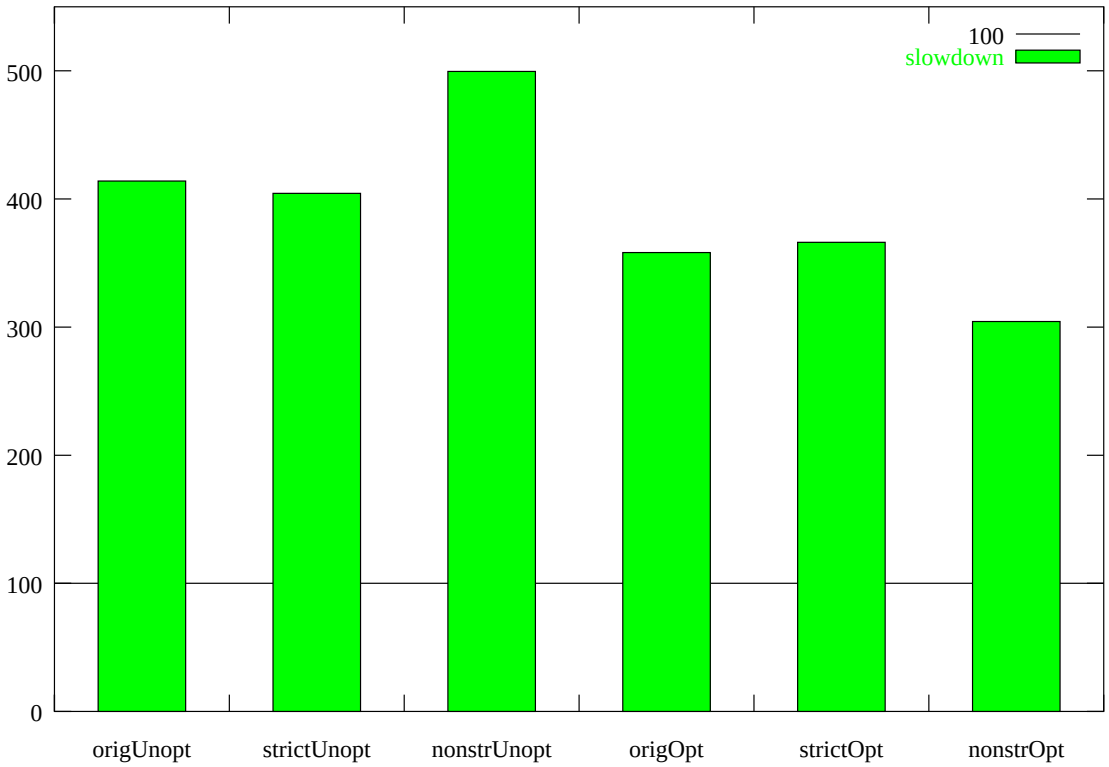


Figure 10-20: Slowdown of Eager Haskell compared to GHC, expressed as a percentage of GHC run time. The two bars indicate OS and internal timer measurements for Eager Haskell; they are similar in all cases.

slower than its lazy counterpart. However, as shown in Figure 10-20 eliminating annotations does improve our relative performance markedly.

In Figure 10-21 we compare the relative speed of the three different benchmarks when compiled with optimization on. Under GHC, eliminating strictness annotations slows the program down by approximately 20%. Interestingly, the inlined version of the benchmark runs at about the same speed as the original benchmark code; apparently inlining the higher-order function *forceBit* has very little effect. It is likely that the compiler is inlining *lift11*, *forceBit*, and *headstrict* and eliminating any trace of the higher-order argument.

Strictness annotations have no measurable effect on the run time of the optimized Eager Haskell code. The third set of bars in Figure 10-21 shows that the two inlined versions run at almost exactly the same speed. Note, however, that in the original benchmark Eager Haskell appears to be much less aggressive in inlining *lift11*. This increases mutator time by about 3%; however, higher-order functions persist in the original code, causing more allocation of continuations and a large increase in garbage collection overhead.

The unoptimized benchmarks tell a murkier story. The run-time ranking of the GHC programs is inverted: the non-strict code now runs the fastest! This is most likely due to the fact that the non-strict benchmark does not call *forceBit*. It is surprising, however, to see that this affects run time by upwards of 14%.

Even more puzzling is the fact that the inlined code is dramatically *slower* than the (equivalent) original benchmark code. This is true for both compilers. This turns out to be due to slight tweaks made to the *lift21* function in the benchmark to adjust the way annotations were used. This increases the call overhead of *lift21* when optimization is switched off.

We noted that eager evaluation does have a liability of its own—it is expensive to create and then force infinite lazy lists. The multiplier benchmark is centered upon such lists. The compiler attempts to identify cyclic computations and introduces thunks along the back edges of uncontrolled loops. This happens in three places in the multiplier benchmark. However, in practice large portions of every list in the program are forced during execution. For all three versions of the benchmark, optimized run times were similar (within a few percent) regardless of whether or not thunk introduction was performed. This suggests that the cost of creating and forcing thunks is roughly comparable to that of computing excess list elements within resource bounds—if most of those elements are later consumed.

With annotations erased, *lift11* is equivalent to *map*, and the other *lift* functions are equivalent to

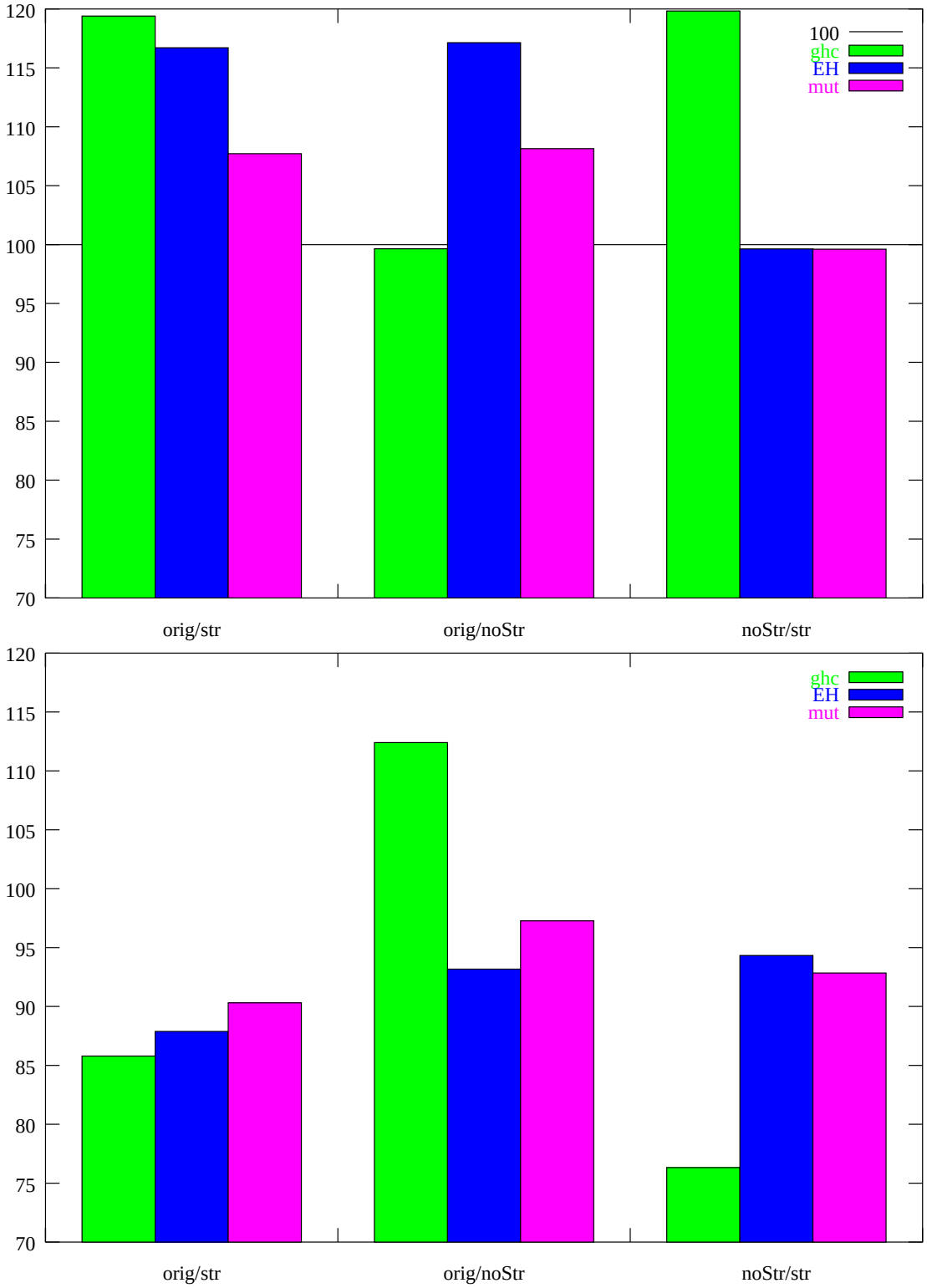


Figure 10-21: Percentage speedup of multiplier when annotations are modified. Numbers less than 100% indicate slowdown. The baseline is 70% so differences are clear. Top shows optimized figures, bottom unoptimized ones. The final bar is mutator time under Eager Haskell.

versions of *zipWith*. These are prelude functions, and all of them can be deforested by the compiler. Indeed, it is possible the original version of multiplier used these functions, and the *lift* functions were introduced only when space leaks made them necessary. We did not investigate the benefits provided by rewriting the program in this way.

Our observations of multiplier are encouraging: the basic eager approach of resource-bounded computation appears to avoid the space leaks which require lazy programmers to explicitly annotate their code. At the same time, the overhead of excess space and time for infinite lazy computations can be controlled. It is clear, however, that the overhead of such programs must be reduced if performance is to match that of a lazy compiler.

Chapter 11

Scheduling Eager Haskell on a Multiprocessor

Eager Haskell was designed with parallelism in mind. It is difficult to expose parallelism using either strict or lazy evaluation, as they both impose strong requirements on program ordering. For example, imagine a tail-recursive loop. Strict evaluation must perform exactly one loop iteration at a time; a parallelizing compiler would need to prove that there are no data dependencies between loop iterations in order to parallelize the loop.

There has been only limited success in using program analysis to extract parallelism from lazy programs. In order to preserve lazy semantics, none of the work performed in parallel can be speculative. Consequently, any parallelizing transformation must first prove that the results it computes are guaranteed to be used. Such analyses tend to give very local results—we discover a value is certain to be used because its consumer is near its producer in the program. In such situations there is rarely any benefit to multiprocessor parallelism, since the result is demanded too soon after its computation is spawned [134]—the parallelism is useful, but too fine-grained.

In an eager language the situation is reversed—computations are assumed to occur speculatively in parallel unless analysis proves that they can productively be serialized. By creating tasks as lazily as possible and using a work-stealing scheduler, we exploit the coarsest-grain parallelism and mitigate the overhead of task creation. Using these techniques we hope to make thread scheduling less frequent in an eager language than thunk creation would be under lazy evaluation. We use fallback to mitigate the effects of useless speculation.

In order to parallelize Eager Haskell, the run-time system described in Chapter 5 must be modi-

fied to identify work which can be performed in parallel, and to schedule that work across multiple processors. Fortunately, much of the necessary mechanism is already in place. During fallback, computation is suspended in a systematic way, yielding thunks. When computation is resumed, work is only forced as needed. We can obtain parallelism by forcing these thunks in parallel rather than waiting for them to be demanded.

In this chapter we elaborate this mechanism for running Eager Haskell programs in parallel on a shared-memory multiprocessor (SMP), and identify possible hurdles which may prevent us from extracting the parallelism we desire. We also describe a general technique, the principle of monotonicity, for handling a shared heap without locking. Our parallelization strategy relies on this technique to efficiently manage shared data.

11.1 Indolent task creation

When fallback occurs, two types of data structure are created to represent the state of outstanding computation:

- A thunk is created for every procedure call.
- A suspension is created when a running procedure requires data which has not yet been computed.

Every suspension which is created as a result of fallback will be dependent either upon another suspension or upon a thunk; thus each such suspension will be transitively dependent on a thunk. Any attempt to force such a suspension must therefore force a created thunk. Thus, when they are created, suspensions cannot possibly embody useful parallelism.

We therefore focus on the thunks. It is simple for the run-time system to collect thunks as they are created. This is a form of *indolent* task creation [127]: no explicit representation exists for parallel work until and unless fallback occurs.

Note that a suspension may *eventually* embody useful parallelism. This is true if the suspension will eventually be forced, but the thunk it depends upon is first forced by some other computation. If insufficient parallelism can be extracted by forcing thunks, then forcing suspensions whose dependencies are available is the only alternative way to expose additional parallelism.

11.2 Scheduling Strategy

Recall from Section 5.4 that thunk creation occurs starting at the leaves of the call tree and working toward the root. A coarse-grained thunk is one which is itself the root of a large call tree. There is no *a priori* way to identify the size of this tree. However, it is clear that the deepest subtrees must occur nearest the root. Thus, we should schedule the outermost thunks first in order to extract coarse-grained parallelism.

This is exactly the goal of the parallel work-stealing strategy used in Cilk [27]. In Cilk, every processor maintains a dequeue containing outstanding parallel work. A processor pushes and pops computations from the top of its dequeue, treating it just like a call stack. However, if any processor exhausts its available work (its dequeue becomes empty), it *steals* work from a victim processor selected uniformly at random. Work is stolen from the bottom of the victim's dequeue.

After fallback in Eager Haskell the *topmost* piece of work (whether suspension or thunk) is forced in order to guarantee that progress is made on the lazy work. In effect, the processor steals work from itself. We run Eager Haskell programs on a multiprocessor by giving each processor its own exception flag, shadow stack, and *thunk stack*. During fallback, thunks are pushed onto the thunk stack as they are created. The computations nearest the root of the call tree are uppermost in the thunk stack. Execution restarts from the root of the local call tree as before. Thieves steal work by popping thunks off the thunk stack.

On a uniprocessor, the Eager Haskell run-time system is responsible for tracking resource bounds and deciding when an exception should be signaled. This could lead to poor resource usage on a multiprocessor, as a single thread can run for long periods before exhausting its resource bounds, while the remaining processors sit idle. Thus, an idle processor may raise an exception on any other processor and thereby force the creation of additional work.

The protocol for work stealing in Eager Haskell is therefore structured as follows:

- Thief completes its local work.
- Thief selects a victim uniformly at random.
- Victim's work stack is popped, unless it is empty.
- The work thus obtained is forced. If it has already been evaluated, this has no effect and more work will be sought.
- If the victim's stack was empty, set the victim's exception flag and seek more work.

11.3 Scheduling in the presence of useless computation

Meanwhile, a little more work is required when an exception occurs. Naturally, newly-created thunks must be pushed onto the thunk stack. It is, however, unclear what should be done with thunks that are *already* on the thunk stack. There are two cases of particular interest. First, there may be a thunk on the thunk stack which represents data which will be required, but will not be needed for a long time. Such a thunk is ripe for parallel execution, and should be retained in favor of more short-lived thunks. Second, there may be a thunk on the thunk stack which represents the useless tail of an infinitely-growing computation. Such a thunk is actively dangerous: it does not represent useful work, and will result in wasted memory and processor time if it is forced.

Unfortunately, there is no simple way to distinguish these cases. In the following example, *all* the thunks created when *bigtree* suspends represent infinite computations:

$$\begin{array}{lll} \textit{bigtree} & :: & \textit{Integer} \rightarrow \textit{Tree Integer} \\ \textit{bigtree } n & = & \textit{Node } n\ (\textit{bigtree}\ (n * 2))\ (\textit{bigtree}\ (n * 2 + 1)) \end{array}$$

We have already observed that infinite data structures will cause sequential execution to consume excessive resources. This problem can be exacerbated in a parallel setting—if a processor obtains the thunk of an infinite computation, it can continue consuming memory without bound.

The most promising solution to this problem is to permit the garbage collector to manage parallelism. The thunk stack should be considered a *weak reference*—objects which are reachable only from the thunk stack should be garbage collected and removed. This gives a minimal liveness guarantee for stealable thunks.

However, there remains a nettlesome problem: if *bigtree* is stolen, the thief will run nothing but calls to *bigtree*. There needs to be some mechanism to throttle such computations before they swamp the system. This means that fallback cannot be a purely local operation; occasionally non-root computations should seek work elsewhere in favor of resuming their current computation. This is at odds with any sort of locality, so it should not happen too frequently. Unfortunately, the garbage collector is much less helpful in this instance. Stealable thunks reside in shared memory, and so the root of computation on every processor is always reachable from tenured space.

Solving this problem will require coordination between global garbage collection, fallback, and thunk stealing. One simple algorithm would signal an exception on every processor when tenured space becomes full. When a processor completes fallback, it garbage collects. With an empty stack, the new nursery will be empty; only the global roots need be traced. When collection is complete,

dead thunks are discarded and work stealing begins anew. However, this effectively imposes a global barrier each time tenured space fills, sacrificing much of the parallelism possible with independent garbage collection.

11.4 Memory Structure: The Principle of Monotonicity

In addition to the coarse-grained problem of identifying and scheduling parallel work, the Eager Haskell implementation must address the fine-grained problem of coordinating parallel access to shared data. The data structures of Eager Haskell have been designed to permit lock-free parallel manipulation of objects in shared memory. In this section we articulate a general principle, the *Principle of Monotonicity*, which can be used to structure programs with lock-free synchronization, and then show how the principle is applied to Eager Haskell.

The fundamental idea behind the Principle of Monotonicity is simple: as long as a shared object may only be updated by a single processor, and is becoming "more defined" according to some ordering, it can be read and updated using simple loads and stores; there is no need to resort to atomic memory operations such as compare and swap. Such operations are required only for non-monotonic updates.

The Eager Haskell realization of this principle is shown in Figure 11-1. The natural ordering for Eager Haskell is the one imposed by evaluation: objects start out empty; they are computed exactly once, at which point the empty object is overwritten with the computed value. Because there is only a single writer, this store can take place without the use of atomic memory operations such as compare and swap. If the computation suspends, the empty object is instead overwritten with a thunk or a suspension. When a thunk is forced or a suspension is resumed, it must be *atomically* emptied. This guarantees that exactly one thread runs the suspended computation.

The presence of indirections complicates this simple structure somewhat. Barrier indirections exist to safely encapsulate nursery references. The garbage collector of the owning processor is responsible for promoting the pointed-to data; the barrier indirection is then transformed into an ordinary indirection as indicated in the figure.

Recall that we can shortcut indirections, as shown in Figure 5-6. Unlike the other transitions shown in Figure 11-1, barrier shortcutting changes the state of an object *reference*, and not of the object itself. This means that any reference anywhere in the heap which points to an indirection can instead be made to point to the destination of that indirection.

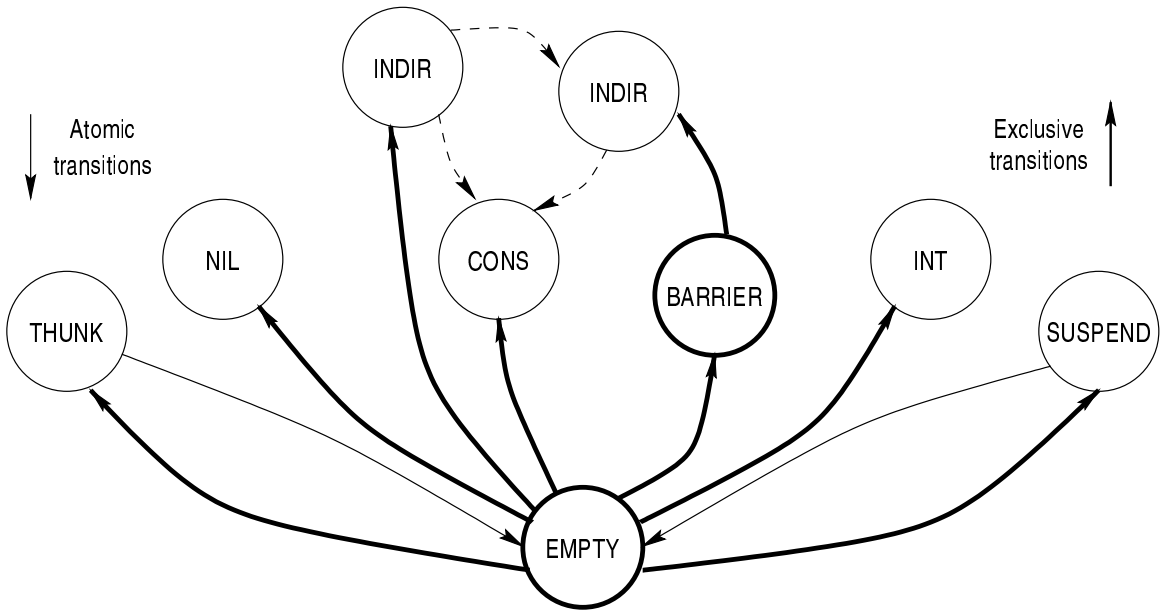


Figure 11-1: Monotonic update of Eager Haskell objects. Dashed arrows indicate permissible transitions for an object *reference*, rather than for the actual object itself.

Note that an ordering has also been imposed on object references: those most distant from a non-indirection are considered to be "greatest", with the each indirection in a chain being progressively "less". In order to guarantee that shortcutting strictly decreases the length of an indirection chain, each shortcutting step must be made atomically. Consider a chain of indirections $a \rightarrow b \rightarrow c \rightarrow d$. Two threads might be competing to shortcut a; thread 1 wishes to shortcut $a \rightarrow c$, thread 2 wishes to shortcut $a \rightarrow d$. If thread 2 succeeds, then thread 1 will actually be *undoing* some shortcutting. This can be dangerous if a reference has been made to point directly at an object such as the *cons* cell shown, and is then moved back to point to an indirection.

The rules for reading and writing according to the principle of monotonicity are fairly simple. Every object has a *state*, and a series of *permitted transitions* from that state. An object in a particular state may contain an arbitrary (but well-defined) number of data fields. However, each pair of adjacent states must be distinguishable by a single datum, which we refer to as the tag (it *is* the object tag in Eager Haskell) which can be read and written atomically. If any non-monotonic transitions are permitted, then it must be possible to do an atomic compare-and-swap operation on the tag as well.

The principle of monotonicity rests on our ability to identify certain states as having an *owner*

(these are the heavy states, Empty and Barrier, in Figure 11-1. The owning thread is permitted to make *monotonic updates* to the state (heavy lines). The protocol for monotonic update is simple:

1. Non-tag data is written normally.
2. This data is committed to shared memory [121].
3. A write/write fence is performed on the written data and tag.
4. The tag data is written in a single operation.
5. The tag is committed to shared memory.

The fields updated during a monotonic transition must be disjoint from those fields which are valid in the current state. It must be possible for another thread to read a consistent snapshot of the object even when it is in the process of being updated.

Non-monotonic transitions can be performed by any thread. As a result, they require the use of an atomic memory operation (such as compare and swap or load lock / store conditional) in order to perform the state transition. In addition, non-monotonic transitions *may not* update any fields. This means that the valid fields after a non-monotonic transition must be a subset of the fields valid before the transition. Only the tag changes value.

Note that if a monotonic transition is possible from a particular state, the same transition must be possible from any states reachable through a non-monotonic transition. Thus, in an exclusive state a non-monotonic transition must move to an exclusive state with the same owner. Only a monotonic transition can make an object's state non-exclusive. From a non-exclusive state, of course, no monotonic updates are possible; therefore all transitions are atomic and non-monotonic.

There are two ways to read the fields of a state, depending on the nature of that state. If an object is in an exclusive state, and non-monotonic transitions do not invalidate any of the fields being read, the owner can read object fields freely. If the object is not owned by the reading thread, or non-monotonic transitions may invalidate fields being read, the following protocol must be obeyed:

1. The tag is read to determine the object state.
2. A read/read barrier is performed on the tag and the fields to be read.
3. The field data is reconciled with main memory [121].
4. All required fields are read.

5. A read/read barrier is performed on the fields read and on the tag.

6. The tag is reconciled with main memory.

7. The tag is read. If the state is unchanged, the data read is valid, otherwise the read must be re-tried starting from the beginning.

If the initial state has no outgoing transitions then we can read the fields freely, and the second tag check is not required. Similarly, if the accessed fields are valid in any reachable state, the check can be skipped. In effect, we can think of a read as an atomic operation on the whole data structure, but one which might possibly fail.

We must ordinarily be careful in allocating and deallocating shared objects. When a thread performs an allocation, the resulting object is in an exclusive state and owned by the allocating thread. Ordinarily the object does not contain useful data upon allocation; we can imagine an infinite number of "just allocated" states in which the object fields contain garbage.

The act of storing a reference into a shared data structure (a so-called *broadcasting store* [105, 72]) counts as a monotonic transition. This means that the object's tag must be initialized, and that a commit and a write/write barrier are required between this initialization and the broadcasting store.

De-allocating a shared object requires knowing that it will not be accessed from any thread again. The principle of monotonicity provides no direct evidence of this fact; it must be encoded in the protocol or established in some other fashion. In Eager Haskell the garbage collector must establish the necessary invariants.

Other techniques for non-blocking management of shared data can be expressed using the principle of monotonicity. For example, we can express *trylock* (the non-blocking locking primitive) and *unlock* very easily using a boolean tag: *trylock* attempts to atomically set the tag, returning *True* if the attempt succeeds and *False* if the tag is already set. If *trylock* returns *True* the protected data can be accessed freely; access is relinquished using *unlock*, which simply writes *False* into the tag. As this example should make clear, the principle of monotonicity is not a magic bullet to avoid mutual exclusion. Instead, it is a design principle to permit shared data to be manipulated in a disciplined way without blocking.

Chapter 12

Compiling *pH* Programs Using the Eager Haskell Compiler

The Eager Haskell compiler exploits the absence of side effects and barriers in order to enable aggressive program optimizations such as full laziness. Nonetheless, both Eager Haskell and *pH* share a common compiler infrastructure. By selectively disabling various program transformations and re-enabling barrier syntax, the Eager Haskell compiler can be turned back into a compiler for *pH*. However, barriers require compiler and run-time system support which is not provided by the system described in this thesis. In this chapter we present an efficient design for implementing barriers within the Eager Haskell compiler and run-time system.

12.1 What is a barrier?

A barrier is used to detect termination of a region of code, which we will call the *pre-region* of the barrier. When the pre-region has terminated, execution of the *post-region* may begin. In *pH* the pre-region and post-region of a barrier are collections of bindings in a large **letrec** block. Barrier regions can be nested arbitrarily.

A region of code has terminated when *all* computations which occur dynamically in that region have successfully yielded values—even if those values are never subsequently used. For example, the pre-region may contain a large recursive function call. Every value in every invocation of that large recursive call must be fully evaluated before the pre-region is considered to have terminated. The λ_S calculus [17] gives a detailed semantics for barriers.

12.2 Barriers in the *pH* compiler

The *pH* compiler targets an abstract machine called SMT [16]. Part of the SMT machine state is the *current barrier*. This tracks the termination state of ongoing computations. The core of every barrier is a simple counter. This counter is atomically incremented before every thread spawn, and atomically decremented once again when a thread terminates. When the barrier count reaches zero, all computations in the pre-region have completed and the barrier *discharges*. The post-region is guarded by *touching* the barrier. No post-region thread will be run until the barrier has discharged.

Note that when multiple threads are being run, these threads may be in the pre-regions of different barriers. As a result, every thread has an associated barrier. This means that every suspension in the system must record the current barrier. When a suspension is run, it must be run in the context of the barrier which was in effect when it was created.

Nesting of barriers means that a binding may occur in the pre-region of many barriers at once. In practice, we track only the innermost barrier region. Every barrier has an associated *parent*—the barrier region in effect on entry to its pre-region. Creating the new barrier increments the parent barrier, and discharging the barrier decrements the parent barrier. In this way, computations need only update the state of a single barrier as they execute.

12.2.1 A lazier barrier

The SMT implementation of barriers has a major drawback: it imposes the run-time cost of modifying the barrier count on every single thread creation and termination. Moreover, barriers are shared among threads which are potentially executing on multiple processors. Thus, barrier counters are shared data, must reside in memory, and must be manipulated using expensive atomic memory operations.

The overhead of atomic memory operations can be mitigated by tracking barrier counts locally, but this merely decreases overheads rather than eliminating them. Instead, we shift the overhead of barriers onto the suspension path. Recall that Eager Haskell does not explicitly spawn threads. Instead, threads run until they are forced to suspend; when a thread suspends, or when it completes execution, its successor thread is immediately run. Each time a thread suspends, the successor thread becomes a new thread of execution. Thus, for a barrier region with n outstanding suspensions there will be n suspended threads, plus a possible single running thread.

Thus, we can avoid tracking barriers when execution continues normally. We start with a barrier

count of one (the initial thread of execution). If no computation suspends, we eventually reach the end of the barrier region and decrement the barrier count. It is now zero, and execution can proceed normally. If suspension occurs and there is a successor thread then we increment the barrier count. When a suspended thread is re-scheduled and executes to completion, the barrier count is decremented once again.

Because successor threads are encoded in the execution order of the compiled program, there is no easy way to check whether a given thread has a successor. However, the compilation scheme for *pH* guarantees the existence of a successor thread if a thread has never suspended before. Thus, we increment the barrier count every time a suspension is created. When control returns to the run-time system after running a previously-created suspension, we decrement the barrier count once again.

12.2.2 Reducing state saving

In *pH* every computation takes place in the context of *some* barrier. Conceptually, a global barrier surrounds the entire program execution and indicates when the program has terminated. If the system runs out of runnable work before this barrier has discharged, then deadlock has occurred.

Most programs execute almost entirely in the context of this global barrier. Nonetheless, every suspension in *pH* includes a field to record the current barrier. This adds a word of overhead to every suspension in the system. We would like to represent suspensions in a uniform way, yet still be able to save or restore barrier state when required.

A simple trick can be used to accomplish this: make the code to restore the barrier look like a suspension. If we suspend in the pre-region of a barrier (except the global barrier), we create a regular suspension as usual. We then create a "barrier restoration" suspension. This special suspension contains a reference to the current barrier and to the original suspension. The run-time system resumes this suspension in exactly the same way as an ordinary suspension. Instead of regular *pH* code, the suspension runs a stub which restores the barrier state and then jumps to the original suspension. The use of barrier restoration suspensions adds a good deal of overhead—an entire suspension—when suspension occurs in the pre-region of a barrier. However, the vast majority of suspensions are smaller. We add overhead for barriers only when they are actually used.

Note that the global barrier need not be treated in the same way as local barriers. If the state of the global barrier is shared among many processors, then updates become a major source of contention. Instead, suspensions can be tracked locally on each processor. There is no need to check the global state of the barrier until all other execution has ceased.

12.3 Barriers in Eager Haskell

Unlike *pH*, Eager Haskell does not have a notion of a "global barrier" at all. Indeed, if the value of a suspended computation is not needed, the suspension is simply ignored and will be cleaned up by the garbage collector. If we wish to allow Eager Haskell to generate barrier code, we must make sure that the run-time system does not lose any of the work in a barrier region. In this section we explore how to add barriers to Eager Haskell while still maintaining demand-driven suspension.

We take the low-overhead barrier described in the previous section as a starting point. Because suspensions will only be run if they are demanded, we explicitly track all pre-region suspensions and demand them. This has much higher overhead than simply counting suspensions. An additional complication is that the Eager Haskell code generator has no provisions for representing barrier constructs. Instead, we add explicit functions for creating and synchronizing on barriers. A separate phase of compilation replaces barriers with a mixture of touch operations and calls to these special constructs. Placing these functions in a user-level library will allow future experiments with different sorts of program synchronization (such as an efficient construct for hyperstrict evaluation).

We do not propose adding termination detection at the outermost level of Eager Haskell programs. The barrier implementation we describe in this section sacrifices many of the economies of the Eager Haskell execution strategy. In particular, there is no way to discard disused computations and still maintain the termination guarantees which *pH* provides. At the top level these guarantees simply allow us to detect deadlock; the Eager Haskell run-time system uses an entirely different set of techniques (with different semantics) for deadlock detection. Most notably, Eager Haskell allows deadlock in computations whose results are never used. No deadlock of any kind is permitted in *pH* programs.

12.3.1 Tracking the work

Tracking all the work associated with the pre-region of a barrier is not difficult. Every barrier includes an associated work pool. When suspension occurs in a barrier context, the suspended work is added to the work pool of the current barrier. Before we can exit the current barrier region, we must remove work from the work pool and force it (again in the context of the barrier) until the work pool is empty.

The chief complication is deciding what should be done when a pre-region computation forces a preexisting suspension. The run-time system must save the active barrier and force the suspension

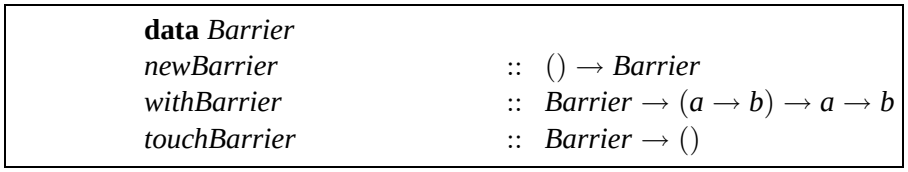


Figure 12-1: Types and functions for representing barriers

in a context with no active barrier. If the suspension being forced itself occurred in a barrier context, then it will be a barrier restoration suspension and the appropriate barrier state will be restored.

12.3.2 Run-time system changes

We can represent the work pool of a barrier using a signal pool, as described in Section 9.1. A signal pool gives a simple, language-level way to say "synchronize on all outstanding pre-region computations"—simply touch the signal pool. In our array implementation, the signal pool was constructed explicitly by user code. Since *any* function may be called from within the pre-region of a barrier, we do not want to do this in Eager Haskell. Instead, the run-time system adds suspensions to the pool as they are created.

This means that the signal pool used to represent barrier regions can expand dynamically in an unstructured way at the same time as it is being forced. Contrast this to the signal pool for an array, which is fixed by the structure of calls to the *seq* function and its relatives. This problem has a simple solution; when the signal pool has been forced, it is simply forced again. If it is empty, forcing can stop; otherwise it has grown and should be forced again. The signal pool can be kept in an M-structure and updated atomically. We thus represent a barrier as an M-structure containing a signal pool.

12.3.3 Compiler changes

In order to allow the Eager Haskell code generator to compile *pH* programs, we add functions which explicitly create a barrier region and synchronize on it. The abstract interface to barriers can be found in Figure 12-1. The *newBarrier* function creates a new barrier object. The *withBarrier* function takes a barrier object, a function, and an argument. It tells the run-time system that the barrier is currently in force, then applies the function to its argument. Any suspension which occurs during the function application will be captured in the signal pool of the barrier. Finally, *touchBarrier*

$$
\begin{array}{lcl}
\mathcal{B}[\![x = p_k\ \vec{P}_k]\!]\ \beta\ \tau & = & x = \tau\ \text{`seq`}\ p_k\ \vec{P}_k \\
\mathcal{B}[\![x = C_k\ \vec{x}_k]\!]\ \beta\ \tau & = & x = \tau\ \text{`seq`}\ C_k\ \vec{x}_k \\
\mathcal{B}[\![x = f\ \vec{x}_k]\!]\ \beta\ \tau & = & x = \mathit{withBarrier}\ \beta\ (f\ \vec{x}_{k-1})\ x_k \\
\mathcal{B}[\![x = \textbf{case}\ y\ \textbf{of}\ D]\!]\ \beta\ \tau & = & x = \mathit{withBarrier}\ \beta\ (\lambda\_ \rightarrow \textbf{case}\ y\ \textbf{of}\ D)\ () \\
\mathcal{B}[\![B_1\ ;\ B_2]\!]\ \beta\ \tau & = & \mathcal{B}[\![B_1]\!]\ \beta\ \tau \\
 & & \mathcal{B}[\![B_2]\!]\ \beta\ \tau \\
 & & \\
\mathcal{S}[\![x = p_k\ \vec{P}_k]\!]\ \tau & = & x\ \text{`seq`}\ \tau \\
\mathcal{S}[\![x = C_k\ \vec{x}_k]\!]\ \tau & = & \tau \\
\mathcal{S}[\![x = f\ \vec{x}_k]\!]\ \tau & = & \tau \\
\mathcal{S}[\![x = \textbf{case}\ E\ \textbf{of}\ D]\!]\ \tau & = & \tau \\
\mathcal{S}[\![B_1\ ;\ B_2]\!]\ \tau & = & \mathcal{S}[\![B_1]\!]\ \mathcal{S}[\![B_2]\!]\ \tau \\
 & & \\
\mathcal{P}[\![x = E]\!]\ \tau & = & x = \tau\ \text{`seq`}\ E \\
\mathcal{P}[\![B_1\ ;\ B_2]\!]\ \tau & = & \mathcal{P}[\![B_1]\!]\ \tau \\
 & & \mathcal{P}[\![B_2]\!]\ \tau \\
\mathcal{P}[\![B_1 \ggg B_2]\!]\ \tau & = & \beta = \mathit{newBarrier}\ \tau \\
 & & \mathcal{B}[\![B_1]\!]\ \beta\ \tau \\
 & & \tau_1 = \mathcal{S}[\![B_1]\!]\ (\mathit{touchBarrier}\ \beta) \\
 & & \mathcal{P}[\![B_2]\!]\ \tau_1 \\
 & & \\
\mathcal{T}[\![B_1 \ggg B_2]\!] & = & \mathcal{P}[\![B_1 \ggg B_2]\!]\ () \\
\end{array}
$$

Figure 12-2: Translation replacing barrier constructs with barrier functions and explicit synchronization.

returns a value only when the signal pool of the provided barrier is empty.

We remove barriers after program optimization is complete, but before closure conversion occurs. At this point, all subexpressions are properly named and **let** expressions have been un-nested. The barrier removal algorithm is given by $\mathcal{T}[\![B_1 \ggg B_2]\!]$ in Figure 12-2. Here B_1 are the bindings in the pre-region of the barrier and B_2 are the bindings in the post-region. The helper function $\mathcal{B}[\![\cdot]\!]$ surrounds function calls and case expressions with a call to *withBarrier* β. We explicitly synchronize simple local computations in the pre-region using $\mathcal{S}[\![\cdot]\!]$ in order to avoid transforming those computations into functions as is done with $\mathcal{B}[\![x = \textbf{case}\ldots]\!]$. In effect, we create as much of the signal tree as possible statically, and then insert calls to *withBarrier* when this is not possible.

12.3.4 A synchronization library

The functions in Figure 12-1 can also be provided to the user in a Barrier library. We can also provide a combinator which applies a function to an argument and does not return until the resulting

computation has terminated:

```
applyTerminate          :: (a → b) → a → b
applyTerminate f x      =  touchBarrier b `seq` r
    where b             =  newBarrier ()
          r             =  withBarrier b f x
```

Many parallel dialects of Haskell make use of classes whose sole task is to perform deep sequencing on large data structures—*i.e.* , to traverse those data structures and ensure that they have been fully computed. If we instead require the producer of such structures to terminate, we obtain deep sequencing without the cost of traversing the data structure. The barrier's thread pool encapsulates the "interesting" parts of such a traversal—the places where further evaluation is required.

Chapter 13

Conclusion

Chapter 11 presented a technique for running Eager Haskell programs on a multiprocessor, and Chapter 12 detailed how *pH* programs might be compiled using the Eager Haskell compiler. However, eager evaluation shows great promise in everyday Haskell compilation. In this chapter we outline future directions for work in eager evaluation, and discuss improvements which must be made to the Eager Haskell implementation to turn it into an production-quality tool. We conclude with a brief survey of the accomplishments so far and the most promising future directions.

13.1 Semantics

In Chapter 3 we presented a semantics for λ_C which contained numerous extensional equivalences. These equivalences are included with an eye to simplifying the equational proofs found elsewhere in the thesis. However, we have played somewhat fast and loose in the rules we have added. Several of the rules can be derived from one another; this redundancy could, with care, be eliminated. Meanwhile, the consistency of most of the expansion rules has not been formally established. This requires proofs of contextual equivalence—rarely a simple task in practice.

From a practical standpoint, there is still no standard way to discuss the semantics of functional programming languages. Numerous semantic styles exist with varying tradeoffs, and within those styles subtle differences can have dramatic semantic impact. Naturally, these particular styles are driven by the applications for which they are used. A common framework is required before these varying approaches can comfortably be compared.

Such a framework needs to be built up starting with small building blocks. An inclusive small-step semantics, such as the one presented for λ_C, is a good starting point. From there, other varieties

of semantics are a matter of imposing a structure on permitted reductions. For example, the reduction strategies in Chapter 4 are expressed by restricting the reduction system, collapsing multiple reductions (as in β_{var}), and imposing a structure upon the term being reduced. Big-step reduction encodes strategy and reduction rules together, and can be justified from the small-step semantics in a similar fashion. A well-designed core semantics will allow the correspondence between different semantic styles to be justified in a purely mechanical fashion.

13.2 Eagerness

As we showed in Chapter 4, the Eager Haskell compiler is simply one point on a vast spectrum of possible hybrid strategies. Restricting our attention to hybrid strategies whose semantics match those of Haskell still leaves tremendous potential for new research. For example, it might be worth using eager evaluation only for recursive function calls. Type information or termination analyses could be used to guide transitions between laziness and eagerness. Limited eagerness for provably terminating expressions has shown promise [36], but has not seen widespread use. Two particularly productive avenues of exploration for mainly-eager evaluation appear to exist. First is to incrementally improve the fallback model used in Eager Haskell. The second is to dispense with fallback entirely and explore new eager execution models.

13.2.1 Fast stack unwinding

The purpose of the fallback process is to unwind the execution stack of the running program and to transform the computations on that stack into demand-driven thunks. However, this need not happen a frame at a time, nor are there particular limits on how much of the stack must be unwound (except that we must always eventually unwind to the outermost useful computation). In fact, fallback is really a bulk continuation capture mechanism, and any technique useful for continuation capture can potentially be applied. With that in mind, we can structure the system so that fallback never involves compiled code at all.

One technique for accomplishing this is *bulk fallback*. Compiled code must place markers on the stack at call points. Every return point becomes an entry point. When fallback occurs, the runtime system traverses the stack, making use of the information contained in the markers to transform each frame into an appropriate thunk. Once the stack has been copied to the heap in this manner, a non-local exit branches to the outermost level of computation, and the program is restarted in a

demand-driven fashion as usual.

Bulk fallback has obvious optimizations. A sufficiently clever implementation could perform fallback without copying by simply transforming the markers directly into a linked heap structure (or having each caller structure the frame so this is automatically the case). Execution would resume on a new stack, and the old stack would become part of the heap, where it can be garbage-collected as usual. Furthermore, the markers can serve double duty by storing garbage collection information even in the absence of fallback, thus permitting pointer and non-pointer data to be freely intermixed on the stack. A marker technique is already used for garbage collecting the stack in GHC [75].

Bulk stack unwinding requires care to identify strictly dependent frames. When a caller immediately synchronizes on the result of its callee, the callee should be run first, and the caller can then be run. This happens naturally with the current fallback mechanism (the caller will suspend). When the stack is processed in bulk, the markers must indicate such strict dependencies if stack growth during forcing is to be controlled.

It should also be noted that bulk stack techniques will increase the complexity of generated code. Every function call must push a marker on the stack; the cost of a single store at each call site adds up, and may not be worthwhile for infrequently used information. In addition, every return point also becomes a function entry point. This is likely to increase the number of entry points overall, and most particularly will add a second entry point to every single-entry non-leaf function. An off-the-cuff experiment early in compiler development indicated substantial additional overhead just for marker pushing in Eager Haskell programs. If we compile to native code (or mangle the C compiler output as in GHC), mapping techniques involving the return address can be used to eliminate these overheads.

13.2.2 Hybrid evaluation without fallback

A more ambitious exploration of hybrid evaluation would throw away the idea of fallback and explore an entirely different mechanism for mediating the transition between eagerness and laziness. For example, it is possible to devise hybrid execution strategies in which every function call is guaranteed to return a value. This would restore an invariant useful for both strict and lazy compilation, and make it easy to use techniques such as unboxing to optimize program code. However, such an approach could not rely on the fallback mechanism described in this thesis; fallback requires that any function call be able to suspend at any time. The resulting language would likely require a mixture of both lazy and eager evaluation in compiled code, rather than relegating laziness to the

run time system.

13.3 Improving the quality of generated code

There are numerous places where the present Eager Haskell compiler and run-time system can be improved. The simplest and most direct way to improve the run time of Eager Haskell programs is to reduce the administrative overhead of the evaluation mechanism itself. This means a good deal of performance tuning and optimization of the run-time system and garbage collector. In this section we focus on improvements which can be made to the portions of the Eager Haskell implementation (code generator and runtime) detailed in this thesis. In Section 13.4 we turn our attention to higher-level compiler improvements.

13.3.1 Garbage Collection

Given the large garbage collection overheads associated with many of our programs (see Figure 10-3), any improvement in the garbage collector will improve the run time of every Eager Haskell program. Nursery collection is already fairly efficient (witness the relatively low GC overheads for fib and queens); most of the optimization possible for nursery objects consists of inlining and loop unrolling in the main collector loop. Tenured collection is substantially more expensive.

Simply decreasing the number of allocated and live objects will benefit any garbage collection strategy. As noted in Section 7.3, the current code generator is aggressive about batching allocations, and this often causes empty objects to live across function calls and garbage collection points. Increasing the number of allocation points increases the number of suspension points in the program; the corresponding increase in code size (even if that code is rarely executed) may slow program execution. We hope to make this up by reducing the load on the garbage collector and dramatically decreasing the number of write barrier checks which must be performed.

Garbage collector performance is often determined simply by the tuning of various collector heuristics. For example, increasing the nursery size can dramatically increase memory footprint and cause TLB thrashing or paging during collection. However, it decreases the rate at which objects are copied, and the rate at which they are aged and subsequently tenured. Similar tradeoffs apply to tenured space. In practice, no one set of parameters appears to work well for every program. Nonetheless, it should be possible to improve the present parameters, which were fixed fairly early in development based on a small subset of the final benchmarks.

The tenured collector sweeps eagerly and allocates very large bitmaps for marking. It should be a simple matter to separate BiBoP chunk sweeping from large-object sweeping. Sweeping a BiBoP chunk can be a simple matter of replacing its allocation bitmap with the mark bitmap. BiBoP chunks which are allocated during marking can share a single mark bitmap, which is simply discarded during sweeping. Large-object sweeping can be done using per-object mark bits; the extra word of memory required for marking is more space-efficient than allocating mark bitmaps for the entire heap. The chief challenge is devising a mechanism for mapping a pointer to the appropriate chunk bitmap. At the moment chunk descriptors are stored in a separately-allocated data structure; placing them in the chunk itself would make them easier to find, but might reduce the benefit of having pointer-free object pages which would otherwise be untouched by the collector.

Limiting write barriers to tenured indirections would further simplify the implementation of write barrier code (and reduce overhead). This would necessitate changes to the code generator. Potentially suspensive back edges would allocate enough space for a tenured indirection. When the corresponding binding is compiled, the empty slot can be overwritten with an indirection to the newly allocated object. In this way, the results of computations are always allocated in the nursery and filled in with initializing stores. The write barrier routine which introduces tenured indirections into already-allocated objects can be eliminated.

Our current collector uses only two generations, a nursery and a tenured space. It is possible to use more sophisticated non-moving generational schemes within tenured space. One simple example would be to separate truly tenured objects from those which are merely long-lived. It should only be necessary to trace objects reachable from roots once, when the root is written; subsequently this memory should not be scanned again. This effectively creates a third generation.

13.3.2 Reducing synchronization

The chief overhead of non-strictness during ordinary evaluation is the cost of checking for various exceptional conditions. In the present Eager Haskell implementation, this takes two forms: exception checks at every function entry point, and full/empty checks when data is used.

There are numerous opportunities to eliminate full/empty checks from Eager Haskell programs. For example, in a uniprocessor Haskell implementation we are not particularly concerned about the loss of eagerness (and consequent loss of parallelism) caused by hoisting synchronization past a function call. Permitting such hoisting will require minor tweaks to the array internals; the signal pool is considered to be strict, and this can cause the compiler to introduce deadlocks along signal

pool paths when synchronization is hoisted past a non-strict array initialization.

Many compilers give special treatment to “leaf functions”, where it is possible to avoid the overhead of allocating a stack frame. Similarly, it is unlikely to be worth suspending and resuming leaf calls: they represent a bounded (and generally small) amount of computation. We can therefore eliminate stack checks from leaf functions. In fact, only recursive function calls can lead directly to unbounded computation. We can therefore restrict stack checks to recursively bound functions and unknown function applications (which may prove to be recursive). However, preliminary experiments indicate that neither technique may be useful: functions are likely to suspend on unavailable data anyway, and the cost of suspension is higher than the cost of thunk creation.

At the moment no interprocedural computedness information is used by the compiler. Because synchronization is introduced after lambda lifting, obvious computedness information based on lexical context is simply thrown away. It should be simple for the compiler to annotate programs in order to preserve this information. Such an annotation can be used as an initial context during synchronization introduction.

We can also use interprocedural analysis to provide more precise computedness information. As noted in Section 6.10.4, strictness information is not necessarily a useful tool for reducing the amount of synchronization in Eager Haskell programs, as it moves synchronization operations from callee to caller, risking a dramatic increase in program size. In that section, we described computedness analysis, which propagates information on computed arguments from caller to callee. It remains to be seen whether computedness can be as effective as strictness in eliminating synchronization.

Simple computedness is not always sufficient to eliminate checking. We are careful to remove indirections in the local frame so that transitive dependency can be used for local synchronization. However, we can pass an indirection as a strict function argument. The callee will eliminate the indirection, but this will not affect the caller. Thus, even though the function may be known to be strict, it is still not safe to eliminate the indirection check from the callee.

There are several possible answers to this problem. The code generator does not currently distinguish between emptiness checks and indirection checks. These have the same overhead in the common case, but the fallback code for emptiness is considerably more complicated as it requires saving and restoring live data. Thus, even if the check is still necessary, it can be made cheaper.

Many values can never be indirections. At the moment, the compiler makes no use of this fact. When a value cannot be an indirection, we can use transitive synchronization information

with complete impunity—even if the transitive dependencies might themselves yield an indirection. Determining which computations may yield an indirection is a straightforward static analysis problem given the code generation rules. However, the precision of results will depend (as with computedness, strictness, and many other analysis problems in functional programming) on precise higher-order control flow and heap analysis. As with strictness, precision is probably unnecessary in practice; it should be enough to record which functions may return indirections. This can be captured using a simple boolean value.

13.3.3 Better representations for empty objects

The data structures used by Eager Haskell to represent empty proxies could be better chosen. The most glaring example is suspensions (whose complexity is evident from Figure 5-4). As seen in Figure 10-13, it is very rare for multiple values to result from a single suspension. We might therefore use two representations for suspensions: the first suspended-upon variable would hold the frame and dependency information; remaining suspended-upon variables would refer to this first suspension as in the present scheme. However, the benefits of an additional scheme must be weighed against the difficulty of maintaining two separate suspension representations.

We might instead take the opposite tack and attempt to unify the thunk mechanism and the suspension mechanism. This has the potential to dramatically simplify the run-time system by allowing all suspended computations to be treated in a uniform manner. Its chief drawback is the relative complexity of a suspension (which requires a descriptor and two dependency fields) versus a thunk (which only requires a function closure). One possible technique is to track the strict variables in the frame rather than storing the direct dependency explicitly. Every descriptor and every closure would include information indicating which frame entries are required for execution. This has an additional advantage: when the run-time system resumes a suspension or a thunk, it is guaranteed to make progress and will not re-suspend.

A few particular kinds of empty objects may deserve special treatment. At the moment, the Eager Haskell compiler does not give any special treatment to *projections*. A projection is a binding which fetches a single field from a constructor:

$$\begin{array}{l}\textbf{let } \mathit{head} = \textbf{case } \mathit{xs} \textbf{ of} \\ \qquad (x : \_) \rightarrow x \\ \qquad \_ \rightarrow \mathit{error} \ldots\end{array}$$

If *head* suspends, we will create a suspension which incorporates *xs*. If *xs* is later evaluated, this

can result in a space leak: *head* will retain the tail of *xs* unnecessarily.

Two techniques exist which eagerly evaluate *head* when *xs* has been computed. Both use special *projection thunks*. In the simplest technique, the garbage collector performs projection when possible [140]. When multiple projections of a single data structure occur statically (due to the use of lazy pattern matching), a second technique causes all these projections to be evaluated when any one of them is forced [123]. Either technique could prove useful in improving the space performance of Eager Haskell programs: projections will still be performed eagerly by default, but when they suspend they will generate a projection thunk rather than an ordinary suspension.

A similar optimization may be applied to touch-like operations whose discriminant is discarded. Such expressions result from the *seq* operation in Haskell and from the various operators proposed in Chapter 9 for implementing signal pools. Using a specialized representation for such operations would be a first step to fixing some of their problems.

13.3.4 Object Tagging

It is worth revisiting our decision to represent a tag as a simple pair of integers. It is unclear whether using a descriptor word in an object header actually slows code down noticeably. If it does not, the descriptor approach gives far more flexibility in our choice of object representation and is preferable to the more tightly constrained approach of using explicit tags. Use of descriptors is particularly necessary for unboxing.

13.3.5 Unboxing

In GHC the use of unboxed values is an important part of efficient code generation [97]. The Eager Haskell compiler does not give direct access to unboxed values. There are a number of daunting technical problems which must be overcome in order to permit the use of unboxing. In most functional languages, when a function application occurs it is guaranteed that a value will be returned; in such a setting, returning an unboxed value is as simple as establishing a convention for where the value will be placed (on the stack, in a register, etc.). In Eager Haskell there is no guarantee that a function will return a value—any call may exhaust resources and suspend (consider, for example, passing a large list to an unboxed *length* function). Thus, we need some way to create a suspension for an unboxed value. Any such mechanism effectively duplicates the behavior of boxing.

In the presence of a native code generator and bulk fallback, returning unboxed values may be simpler—whole segments of the stack can be restored in one go, and the usual function return mechanism will suffice. In the absence of a native code generator (or another mechanism to save and restore large pieces of the computation state) various workarounds are required. The simplest is to box values as they are returned. Second simplest is to return a flag indicating whether the corresponding value has been computed.

An apparently promising technique is to simply CPS-transform functions which return an unboxed value, transforming an unboxed return into an unboxed call. Unboxed values must be strictly bound [97]; thus, all calls to functions returning an unboxed value look like this:

case *unboxedFunc a b c* **of**
  unboxedValue $\rightarrow$ *e*

The continuation-passing transformation would transform such a call as follows:

unboxedFunc$'$ *a b c* (λ*unboxedValue* $\rightarrow$ *e*)

Note that a closure is created for *e* even if *unboxedFunc*$'$ does not suspend. If we knew that *unboxedFunc* could not suspend, the closure could be created on the stack; however, we could use straightforward unboxed return if we had that much information. The rest of the time we are obliged to heap-allocate a closure for *e*, even though the resulting closure will probably be short-lived. In most cases this will be markedly more expensive than simply boxing the result of *unboxedFunc*.

Even in the absence of unboxed return, permitting the use of unboxed values remains a challenge. In order to represent data structures which contain a mix of boxed and unboxed data, one of two techniques must be used. If we keep the current object representation, unboxed fields must be placed in a separate, segregated structure. The main data structure would include all the boxed fields plus a pointer to the structure containing the unboxed fields. Alternatively, we must change the tag representation to permit a mix of boxed and unboxed data. The most obvious way of doing so is to use descriptors rather than integer tags, as mentioned in the previous section. Otherwise, objects can be partitioned into a boxed part and an unboxed part; the tag would indicate the size of the respective parts.

Even establishing a calling convention for unboxed arguments is difficult in the presence of suspension. It is tempting to simply use the C calling conventions, passing unboxed arguments as C arguments. However, it is not clear how this should interact with entry points (where no additional arguments are expected). A more likely technique is to pass unboxed arguments on the shadow

stack along with regular arguments. If unboxed values are immediately popped from the stack on function entry, stack descriptors might not be necessary; some sort of descriptor will be required for unknown or curried function applications, however.

It is also difficult to mix curried function application and unboxing; the obvious expedient is to box curried arguments and use a special entrypoint to unbox them again. This is one place where the push-enter style of function application might work better than the eval-apply style: each function contains code to handle stack checking and partial application, and as a result it is easier to use specially-tailored calling conventions for every function.

13.4 Other compiler improvements

Eager Haskell is a research project, and like many other research compilers it doesn't contain the well-tuned and complete set of optimizations that one would expect from a production compiler. A number of changes to optimization passes have the potential to dramatically improve the quality of compiled code. We divide these changes into three classes. A good deal of the existing functionality needs to be tuned for performance. Some parts of the compiler (such as deforestation) have been improved upon by newer research. Finally, some important compiler passes are missing or will need to be re-written entirely in order to support desired functionality.

In addition to maintenance work, there are a number of optimizations for procedural languages which have not (to our knowledge) been tried in non-strict languages. By adopting an execution strategy that is very nearly strict, we can hope to adapt some of these optimizations to Eager Haskell. We examine one such opportunity—escape analysis.

13.4.1 Specialization

Haskell's type-based function overloading is elegant, but can be very expensive in practice. Optimizing Haskell compilers perform class specialization in order to eliminate much of this overloading at compile time [57]. However, it is often desirable to explicitly direct the compiler to produce versions of overloaded functions which are specialized to particular types. This further reduces overloading and often improves the code produced by class specialization. The Eager Haskell compiler performs class specialization, but at the moment it does not handle user-directed specialization gracefully, and inter-module specialization information is not correctly emitted.

13.4.2 Control-flow analysis

Most static analyses of higher-order languages have a precision which depends in part on precise control-flow analysis. Indeed, whole-program analysis of Haskell can potentially eliminate much of the overhead of laziness [55, 35], and enable advanced optimizations such as interprocedural register allocation [29]. Starting with the work of Shivers [122], numerous approaches to control flow analysis have been proposed with varying degrees of precision and complexity. Most of the relevant techniques can be found in the book by Nielson *et. al.* [83].

At the moment, no formal control-flow analysis is performed by the Eager Haskell compiler. An exhaustive analysis would require whole-program compilation. For many program analyses, however, a crude approximation to control-flow information is more than sufficient. Given the importance of control flow information to program analysis, it would be worthwhile to perform a separate control flow analysis during compilation. The resulting information could then be used to guide subsequent compiler phases. The whole compiler will then benefit if the analysis is refined.

13.4.3 Loop optimization

At the moment very little effort is made to perform loop optimizations on Eager Haskell programs. Full laziness takes care of invariant hoisting, but there are no loop-specific optimizations. Worse still, the worker/wrapper transformation, which is the cornerstone of loop optimization in GHC, is severely lacking in Eager Haskell: it is only run on top-level functions with precise type information. As explained in Section 6.10.4 it will not be nearly as useful in Eager Haskell in any case.

Instead, a separate pass must be written to perform loop optimizations, replacing the current worker-wrapper phase. We envision a system which splits functions into three pieces (one or more of which can often be omitted): a wrapper, a header, and a body. The wrapper is inlined; it destructures tuples and passes them to the header. The header is run on loop entry; it can contain any synchronization required before the loop is run, and is a logical resting place for expressions hoisted by full laziness. Finally, the body contains the actual loop recursion.

Because Eager Haskell loops are expressed using recursion, the loop-carried variables are passed as arguments from iteration to iteration. It is often beneficial to eliminate invariant arguments. Observations of compiled code indicate that this exposes opportunities for constant propagation that otherwise would have required more complex static analysis to discover. Variables which are not eliminated will subsequently be re-introduced by lambda lifting.

13.4.4 Improving inlining

Aggressive inlining can make Haskell programs dramatically more efficient. The Eager Haskell compiler has always allowed the programmer to specify that certain functions must be inlined. In addition, bindings which are used at most once are always inlined. Until comparatively recently (November 2001) these were the primary forms of inlining in the compiler.

The current version of the compiler includes a phase (run several times during compilation) which identifies candidates for inlining. A simple cost model is used to measure the size of the IR for a binding. If the cost is below one inlining threshold, it will be inlined unconditionally; if it is below a second, larger threshold then the binding is conditionally inlined. A new annotation was added to indicate that particular functions should *not* be inlined. This annotation allows small run-time support routines to be carefully coded so that only commonly-executed code is inlined.

However, there are a number of infelicities in the compiler's inlining decisions. The inlining decisions made by the compiler are *top-down*: we decide whether to inline a function before we know the impact that inlining will have on the function being inlined. This can be particularly unfortunate when a fairly small function, which will be inlined unconditionally, is made up primarily of calls to other unconditionally-inlined functions. We have in effect severely underestimated the function's size and the result is unintentional code explosion.

It is better to make inlining decisions *bottom-up*. In a bottom-up approach to inlining, the function is fully optimized (including all necessary inlining) before the inlining decision is made. In this way, inlining decisions are based on the actual optimized function size. The bottom-up approach is unfortunately more complex than the top-down approach: inlining decisions must be integrated with program transformations, rather than being performed as a separate pass.

The cost model used for making inlining decisions is not actually based on any sort of measurement of program performance, but instead relies upon a good initial guess at how programs should behave. The accounting used when making conditional inlining decisions is shoddy at best, and in practice conditional inlining is very rare. By contrast, the inliner in GHC assigns "discounts" to function arguments when making conditional inlining decisions; these discounts reflect the fact that the inlined code will shrink when particular arguments are known to be constant [100].

One consequence of the late arrival of compiler-directed inlining is that much of the Eager Haskell prelude is explicitly marked for inlining. It would doubtless be better to conditionally inline many of the larger functions which are currently being inlined unconditionally. However, a detailed

audit of prelude code will require a considerable amount of time and effort.

13.4.5 Range-based optimizations

At the moment, constant propagation in the compiler is fairly effective. However, many variables have not one value, but a range of correct values. Because Eager Haskell arrays are bounds-checked, optimizations based on variable ranges can have a strong impact on array performance. Though range-based optimizations of Eager Haskell programs have been proposed on numerous occasions, shortage of manpower and time has prevented any of these plans from coming to fruition. The biggest challenge is the analysis required to determine variable ranges. The simple intraprocedural cases are already handled by the compiler code; a simple abstract interpretation should expose most of the remaining opportunities [83]. As with most other analyses of higher-order languages, a precise analysis requires whole-program control-flow analysis.

13.4.6 Constructed products

A Haskell function can return multiple values by constructing a tuple and returning that tuple. Frequently the caller immediately fetches the fields from the newly-constructed tuple and discards it. In both Id [49] and Haskell [21] it has proven beneficial to eliminate the tupling operations and instead have such functions return multiple values. In GHC this optimization is expressed using unboxed tuples.

Attempts to express such a transformation in Eager Haskell using continuation-passing ran into a number of obstacles; most notably, lazy matching of the constructed result does not work at all. We are currently brainstorming possible ways to express the transformation effectively in the framework of λ_C; adding explicit multiple-value bindings to the IR will complicate the compiler unnecessarily. The simplest technique may be to express lazy projection in a manner that can be exploited by the code generator to generate code which uses multiple-value return.

13.4.7 Better deforestation

The Eager Haskell compiler performs deforestation in a single pass [71, 70]. Opportunities for deforestation are exposed mainly by aggressively inlining prelude code. This has several notable failings. If inlined code fails to deforest, program size has increased (often dramatically) to no avail. Because deforestation happens in a single pass, opportunities which would be exposed by program

simplification are missed—this accounts for the poor performance of the wavefront benchmark. Finally, the need to prevent unintended code explosion during deforestation (particularly across list append operations) is frequently at odds with the need to inline nested list traversals.

GHC has an innovative solution to deforestation—rewrite rules [94]. Rules allow the sophisticated user to specify additional equational transformations to be used by the compiler. This allows optimizations such as foldr/build [38] to be specified in prelude code. Just as important, it allows prelude functions to be selectively inlined exactly when they would be candidates for deforestation. The GHC prelude currently contains three versions of each deforestable prelude function. The first is a wrapper used by the rewrite rules. The second is an efficient, but non-deforestable, version. The third is the version which is used when deforestation is possible.

13.4.8 Escape analysis

At its simplest, escape analysis attempts to identify which heap objects do not escape the function call in which they are allocated. In a strict language, such objects can be allocated on the stack. Stack-allocated objects do not require garbage collection; they are deallocated as a natural consequence of function return. Intraprocedural escape analysis is performed in most functional languages in the guise of constant propagation; it is interprocedural escape analysis which is an interesting and hard problem. Lazy languages cripple interprocedural escape analysis; thunks are stored into data structures and then returned, causing function arguments to escape.

Most of the time Eager Haskell programs run strictly, however. If we perform a standard strict escape analysis, we run into only one problem: what becomes of stack-allocated data during fallback? We propose to *promote* such data to the heap. This will require an additional invariant: no data structure in the heap may point to the stack. If this invariant is respected during normal execution, it is not hard to enforce it during fallback as well. Using bulk fallback might permit even this restriction to be lifted.

13.5 Envoi

Hybrid evaluation shows great promise as a standard execution strategy for Haskell. The Eager Haskell implementation demonstrates that the performance of eagerness is comparable with that of laziness, and can be noticeably better on some problems. With care, we believe the remaining shortcomings of the hybrid strategy can be addressed at the implementation level. By using Hybrid

evaluation, programmers should be able to run Haskell programs efficiently without an intimate knowledge of the workings of the evaluation mechanism and without resorting to program annotations. It is unacceptable for programs to fail unpredictably in the face of such common and simple idioms as tail recursion.

The hybrid strategy presented in this thesis is just a start. It is encouraging to observe the tremendous scope for future work. The present Eager Haskell can be improved dramatically, bringing concomitant improvements in the quality of compiled code. Because the hybrid execution strategy shares the same semantics as a lazy strategy, it benefits from many of the same optimizations: compiler innovations on either side can be carried over to the other. Meanwhile, many of the most interesting experiments in eager evaluation can occur with little or no change to the compiler. New fallback mechanisms and heuristics are a matter of changes to the run-time system. Parallelization is a sticky design problem, but again the necessary mechanisms can be cleanly separated from compilation concerns. In short, Eager Haskell is an excellent substrate for future research in functional programming.

Bibliography

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Boston, 1986.

[2] Shail Aditya, Arvind, Lennart Augustsson, Jan-Willem Maessen, and Rishiyur S. Nikhil. Semantics of pH: A Parallel Dialect of Haskell. In *Proceedings of the Haskell Workshop*, FPCA 95, La Jolla, CA, June 1995.

[3] Shail Aditya, Arvind, and Joseph Stoy. Semantics of barriers in a non-strict, implicitly-parallel language. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture*, La Jolla, CA, June 1995. ACM.

[4] Shail Aditya, Christine H. Flood, and James E. Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *LISP and Functional Programming*, pages 12–23, 1994.

[5] Andrew Appel. Runtime Tags Aren't Necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[6] Andrew W. Appel. Garbage Collection can be Faster than Stack Allocation. *Information Processing Letters*, 25(4), January 1987.

[7] Andrew W. Appel. *Compiling With Continuations*. Cambridge University Press, 1992.

[8] Andrew W. Appel. *Modern Compiler Implmentation in ML*. Cambridge University Press, 1997.

[9] Andrew W. Appel and Zhong Shao. An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures. Technical Report CS-TR-450-94, Department of Computer Science, Princeton University, March 1994.

[10] Z. Ariola and S. Blom. Cyclic lambda calculi. In *International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, September 1997.

[11] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 233–246. ACM, 1995. Full version in [14]; see also [73, 74].

[12] Zena M. Ariola and Arvind. A Syntactic Approach to Program Transformations. In *Proceedings of the Symposium on Partial Evaluation and Semantics Based Program Manipulation*, Yale University, New Haven, CT, June 1991. Also MIT Computation Structures Group Memo322.

[13] Zena M. Ariola and Stefan Blom. Lambda calculus plus letrec: graphs as terms and terms as graphs. Technical Report DRAFT, Dept. of Computer and Information Sciences, Univ. of Oregon, Eugene OR, USA, October 1996.

[14] Zena M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, May 1997. Full version of [11].

[15] Zena M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. Technical Report CIS-TR-96-04, Dept. of Computer and Information Sciences, Univ. of Oregon, Eugene OR, USA, 1996.

[16] Arvind, Alejandro Caro, Jan-Willem Maessen, and Shail Aditya. A multithreaded substrate and compilation model for the implicitly parallel language pH. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[17] Arvind, Jan-Willem Maessen, Rishiyur Sivaswami Nikhil, and Joseph E. Stoy. λ_S: An implicitly parallel λ-calculus with letrec, synchronization and side-effects. *Electronic Notes in Theoretical Computer Science*, 16(3), September 1998.

[18] Arvind, Jan-Willem Maessen, Rishiyur Sivaswami Nikhil, and Joseph E. Stoy. λ_S: An implicitly parallel λ-calculus with letrec, synchronization and side-effects. Technical Report 393-2, MIT Computation Structures Group Memo, October 1998. (Full version of [17] with proofs).

[19] Lennart Augustsson. Implementing Haskell overloading. In *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 65–73. ACM, 1993.

[20] Lennart Augustsson and et al. *The HBC Compiler*. Chalmers University of Technology, 0.9999.4 edition.

[21] Clem Baker-Finch, Kevin Glynn, and Simon L. Peyton Jones. Constructed product result analysis for Haskell. Technical report, University of Melbourne, 2000.

[22] Henk P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[23] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[24] Joel F. Bartlett. SCHEME$-$ $>$ C: A Portable Scheme-to-C Compiler. Technical Report DEC-WRL-89-1, Digital Equipment Corporation, Western Research Laboratory, 1989.

[25] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, Nov 2000.

[26] Richard Bird, Geraint Jones, and Oege De Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(5):541–547, September 1997.

[27] Robert D. Blumofe, Christopher F. Joerg, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded run-time system. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 132–141, Montreal, Canada, 17–19 June 1998. ACM, SIGPLAN Notices.

[28] H.J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[29] Urban Boquist. Interprocedural Register Allocation for Lazy Functional Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, La Jolla, California, June 1995.

[30] Luca Cardelli. Basic Polymorphic Typechecking. *Science of Computer Programming*, 8, 1987.

[31] Alejandro Caro. *Generating Multithreaded Code from Parallel Haskell for Symmetric Multiprocessors*. PhD thesis, MIT, January 1999.

[32] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *ACM SIGPLAN-SIGSOFTWorkshop on Program Analysis for Software Tools and Engineering*, pages 21–31, September 1999.

[33] Satyan Coorg. Partitioning non-strict languages for multi-threaded code generation. Master's thesis, MIT, May 1994.

[34] Satyan Coorg. Partitioning non-strict languages for multi-threaded code generation. In *Static Analysis Symposium*, Sept 1995.

[35] Faxén, Karl-Filip. Optimizing lazy functional programs using flow-inference. In *Static Analysis Symposium*, Sept 1995.

[36] Faxén, Karl-Filip. Cheap eagerness: speculative evaluation in a lazy functional language. In *Proceedings of the fifth ACM SIGPLAN ACM SIGPLAN International Conference on Functional Programming*, pages 150–161, Montreal, September 2000. ACM.

[37] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Canada, 17–19 June 1998. ACM, SIGPLAN Notices.

[38] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, La Jolla, CA, June 1995. ACM.

[39] Andrew Gill and Simon L Peyton Jones. Cheap deforestation in practice: An optimiser for Haskell. In *Proceedings of the Glasgow Functional Programming Workshop*, 1995.

[40] Andrew J. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, University of Glasgow, January 1996.

[41] Seth C. Goldstein, Klaus E. Schauser, and Dave E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1), August 1996.

[42] Seth Copen Goldstein. The Implementation of a Threaded Abstract Machine. Report UCB/CSD 94-818, Computer Science Division (EECS), University of California, Berkeley, May 1994.

[43] Andrew Gordon, Kevin Hammond, and Andy Gill, et al. The definition of monadic I/O for Haskell 1.3. Incorporated into [46]., 1994.

[44] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, 1993.

[45] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[46] Kevin Hammond, et al. Report on the programming language Haskell, version 1.3. Released at FPCA '95. Supersedes [51], superseded by [95]., 1995.

[47] John Hannan and Patrick Hicks. Higher-order UnCurrying. In *Proceedings of the Symposium on Principles of Programming Languages*, San Diego, CA, January 1998.

[48] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Compiling Logic Programs to C Using GNU C as a Portable Assembler. In *ILPS'95 Postconference Workshop on Sequential Implementation Technologies for Programming Languages*, December 1995.

[49] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind. Performance Studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributed Computing*, 18(3):273–300, July 1993.

[50] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press, New York, 2000.

[51] Paul Hudak, Simon L Peyton Jones, and Philip Wadler, eds., et al. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language, Version 1.2. *SIGPLAN Notices*, 27(5), May 1992. Superseded by [46].

[52] C. Barry Jay and N. Ghani. The virtues of η-expansion. *Journal of Functional Programming*, 5(2):135–154, April 1995.

[53] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architecture*, 1985.

[54] Thomas Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Göteborg, 1987.

[55] Thomas Johnsson. Analysing heap contents in a graph reduction intermediate language. In *Proceedings of the Glasgow Functional Programming Workshop*, Ullapool 1990, August 1991.

[56] Thomas Johnsson. Efficient graph algorithms using lazy monolithic arrays. *Journal of Functional Programming*, 8(4):323–333, July 1998.

[57] Mark P. Jones. Partial evaluation for dictionary-free overloading. Technical Report RR-959, Yale University Department of Computer Science, New Haven, CT, 1993.

[58] Mark P. Jones. The implementation of the Gofer functional programming system. Technical Report RR-1030, Yale University Department of Computer Science, New Haven, CT, May 1994.

[59] Mark P. Jones, Alastair Reid, the Yale Haskell Group, and the OGI School of Science & Engineering. *The Hugs 98 User Manual*.

[60] Richard Jones and Rafael Lins. *Garbage Collection*. John Wiley & Sons, 1996.

[61] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[62] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. In *SIGPLAN Notices (Proceedings of the SIGPLAN '86 Symposium on Compiler Construction)*, July 1986.

[63] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 24–35, Orlando, FL, June 1994. ACM, SIGPLAN Notices. Superseded by [98].

[64] Xavier Leroy. Efficient Data Representation in Polymorphic Languages. Rapports de Recherche 1264, INRIA-Rocquencourt, August 1990.

[65] Xavier Leroy. The ZINC Experiment: An Economical Implementation of the ML Language. Rapports Techniques 117, INRIA-Rocquencourt, February 1990.

[66] Xavier Leroy. The Objective Caml system release 3.00, documentation and user's manual. http://caml.inria.fr/ocaml/htmlman/index.html, 2000.

[67] John R. Levine. *Linkers and Loaders*. Morgan Kaufman, Inc., Oct 1999.

[68] Bil Lewis, Don LaLiberte, Richard Stallman, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual*. Free Software Foundation, 2.5 edition, Nov 1998.

[69] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 333–343. ACM, 1995.

[70] Jan-Willem Maessen. Eliminating intermediate lists in pH using local transformations. Master's thesis, MIT, May 1994.

[71] Jan-Willem Maessen. Simplifying parallel list traversal. Technical Report 370, MIT Computation Structures Group Memo, January 1995.

[72] Jan-Willem Maessen, Arvind, and Xiaowei Shen. Improving the Java memory model using CRF. In *Proceedings of the 15th AnnualConference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–12, Minneapolis, MN, Oct 2000. ACM SIGPLAN. Also available as MIT LCS Computation Structures Group Memo 428.

[73] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. Technical report, Fakultat fur Informatik, Universitat Karlsruhe, and Department of Computing Science, University of Glasgow, October 1994. Superseded by [74].

[74] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998. Revision of [73]; see also [11].

[75] Simon Marlow and Simon L. Peyton Jones. The new GHC/Hugs runtime system. Available from http://research.microsoft.com/Users/simonpj/Papers/new-rts.htm, Aug 1998.

[76] Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John H. Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 274–285, 2001.

[77] James S. Miller and Guillermo J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report AIM-1462, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1994.

[78] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge MA, USA, 990. Superseded by [79].

[79] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML*. MIT Press, revised edition, 1997. Revises [78].

[80] J. C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[81] Andrew Moran, Søren B. Lassen, and Simon L. Peyton Jones. Imprecise exceptions, co-inductively. In Andrew Gordon and Andrew Pitts, editors, *Electronic Notes in Theoretical Computer Science*, volume 26. Elsevier Science Publishers, 2000.

[82] Greg Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 1995.

[83] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[84] Rishiyur S. Nikhil. A Multithreaded Implementation of Id using P-RISC Graphs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lectures Notes in Computer Science, Portland, OR, August 1993. Springer Verlag.

[85] Rishiyur S. Nikhil and Arvind. *Implicit Parallel Programming in* pH. Morgan Kaufman, Inc., 2001.

[86] Rishiyur S Nikhil, Arvind, and James Hicks, et al. pH language reference manual, version 1.0—preliminary. Technical Report 369, MIT Computation Structures Group Memo, January 1995. Working document describing pH extensions to Haskell.

[87] Rishiyur Sivaswami Nikhil. Id (Version 90.1) Language Reference Manual. Technical Report CSG Memo 284-2, MIT Computation Structures Group Memo, 545 Technology Square, Cambridge MA 02139, USA, July 1991.

[88] Rishiyur Sivaswami Nikhil. An Overview of the Parallel Language Id (a foundation for pH, a parallel dialect of Haskell). Technical Report Draft, Digital Equipment Corp., Cambridge Research Laboratory, September 1993.

[89] Will Partain. The nofib benchmark suite of Haskell programs. In J Launchbury and PM Sansom, editors, *Functional Programming, Glasgow 1992*, pages 195–202. Springer-Verlag, 1992.

[90] John Peterson and Mark P. Jones. Implementing type classes. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–236, 1993.

[91] Simon L. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. presented at the 2000 Marktoberdorf Summer School.

[92] Simon L Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[93] Simon L. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), April 1992.

[94] Simon L. Peyton Jones, C. A. R. Hoare, and Andrew Tolmach. Playing by the rules: rewriting as a practical optimisation technique. In *Proceedings of the Haskell Workshop*, 2001.

[95] Simon L Peyton Jones and John Hughes. Haskell 98, a non-strict, purely functional language. http://www.haskell.org/definition/, February 1999.

[96] Simon L Peyton Jones and John Hughes. Standard libraries for Haskell 98. http://www.haskell.org/definition/, February 1999.

[97] Simon L. Peyton Jones and John Launchbury. Unboxed Values as First Class Citizens in a Non-strict Functional Language. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, Cambridge, MA, September 1991.

[98] Simon L. Peyton Jones and John Launchbury. State in Haskell. *Journal of LISP and Symbolic Computation*, 8(4):293–341, December 1995. Elaboration of [63].

[99] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages: A Tutorial*. Prentice-Hall, Englewood Cliffs, N.J., 1992.

[100] Simon L. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell compiler inliner. *Journal of Functional Programming*, to appear.

[101] Simon L. Peyton Jones and Will Partain. Measuring the Effectiveness of a Simple Strictness Analyser. In K. Hammond and J.T. O'Donnell, editors, *Proceedings of the Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer Verlag, 1993.

[102] Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 25–36. ACM, SIGPLAN Notices, May 1999.

[103] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

[104] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[105] William Pugh. Fixing the Java memory model. In *Proceedings of the ACM Java Grande Conference*, June 1999.

[106] Niklas Röjemo. *Garbage Collection and Memory Efficiency in Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1995. Full version of [107] and other papers.

[107] Niklas Röjemo. Highlights from nhc—a space-efficient Haskell compiler. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture*, La Jolla, CA, June 1995. ACM. Expanded in [106].

[108] Amr Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, January 1998.

[109] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. In *Functional Programming Languages and Computer Architecture*, pages 106–116, 1993.

[110] André Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, 1995.

[111] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.

[112] K.E. Schauser, D.E. Culler, and S.C. Goldstein. Separation constraint partitioning: A new algorithm for partitioning non-strict programs into sequential threads. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, 1995.

[113] Klaus Schauser. Personal communication, June 1995.

[114] Klaus E. Schauser. *Compiling Lenient Languages for Parallel Asynchronous Execution*. PhD thesis, University of California, Berkeley, May 1994.

[115] Klaus E. Schauser, David E. Culler, and Thorsten von Eicken. Compiler-controlled Multithreading for Lenient Parallel Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*. Springer Verlag, August 1991.

[116] Klaus E. Schauser and Seth C. Goldstein. How Much Non-strictness do Lenient Programs Require? In *Functional Programming and Computer Architecture*, San Diego, CA, June 1995.

[117] Jacob B. Schwartz. Eliminating intermediate lists in pH. Master's thesis, MIT, 2000.

[118] Peter Sestoft. The garbage collector used in caml light. archived email message, October 1994.

[119] Zhong Shao and Andrew W. Appel. Space-Efficient Closure Representation. In *Proceedings of the ACM Conference on Lisp and Functioanl Programming*, June 1994.

[120] Andrew Shaw. *Compiling for Parallel Multithreaded Computation on Symmetric Multiprocessors*. PhD thesis, MIT, October 1997.

[121] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999. ACM.

[122] Olin Shivers. *The semantics of scheme control-flow analysis*. PhD thesis, Carnegie Mellon University, May 1991. Technical Report CMU-CS-91-145, School of Computer Science.

[123] Jan Sparud. Fixing some space leaks without a garbage collector. In *Functional Programming Languages and Computer Architecture*, pages 117–124, 1993.

[124] Guy L. Steele. RABBIT: a Compiler for Scheme. Technical Report MIT/AI/TR 474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1978.

[125] Guy L Steele Jr. Building interpreters by composing monads. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 472–492. ACM, 1994.

[126] Joseph E. Stoy. The Semantics of Id. In *A Classical Mind: Essays in Honor of C.A.R.Hoare (A.W.Roscoe, ed.)*, pages 379–404. Prentice Hall, New York, 1994.

[127] Volker Strumpen. Indolent closure creation. Technical Report 580, MIT Laboratory for Computer Science Technical Memo, June 1998.

[128] Xinan Tang, Jian Wang, Kevin B. Theobald, and Guang R. Gao. Thread Partitioning and Scheduling Based On Cost Model. In *Proceedings of the 9^{th} Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*, Newport, Rhode Island, June 1997.

[129] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. Til: a type-directed optimizing compiler for ml. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 181–192. ACM, SIGPLAN Notices, May 1996.

[130] The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 5.02*.

[131] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[132] Christiana V. Toutet. An analysis for partitioning multithreaded programs into sequential threads. Master's thesis, MIT, May 1998.

[133] Ken R. Traub. *Sequential Implementation of Lenient Programming Languages*. PhD thesis, MIT, September 1988.

[134] Guy Tremblay and Guang R. Gao. The Impact of Laziness on Parallelism and the Limits of Strictness Analysis. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, April 1995.

[135] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 45–52. ACM, 1984.

[136] Philip Wadler. Theorems for free! In *Proceedings of the 4th Conference on Functional Programming Languages and Computer Architecture*, September 1989.

[137] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1991.

[138] Philip Wadler. The essence of functional programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM, 1992.

[139] Philip Wadler. A HOT opportunity. *Journal of Functional Programming*, 7(2):127–128, March 1997.

[140] Philip L. Wadler. Fixing some space leaks with a garbage collector. *Software Practice and Experience*, 17(9):595–609, 1987.

[141] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4), July 1984.

[142] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in LNCS, Saint-Malo (France), 1992. Springer-Verlag.

[143] Paul R. Wilson. Uniprocessor garbage collection techniques. *ACM Computing Surveys*, to appear. Revised version of [142].

[144] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

Appendix A

The Defer List Strategy for λ_C

Section 4.6 briefly sketched the defer list strategy used in implementing Id and *pH*. In Figure A-1 we formalize the defer list strategy in the context of λ_C. In practice, creating and destroying threads of execution can be expensive, even when they are extremely lightweight as in the fully eager strategy of Section 4.4.1. In addition, it is possible to flood the system with vast amounts of blocked work. Finally, the amount of parallelism that can actually be exploited in a real system is usually bounded by the number of available processors.

The most notable feature of the strategy is that every binding in a thread is accompanied by an initially empty binding group known as its *defer list*. We refer to a binding and its accompanying defer list as *work*. The defer list itself contains work—all of it *pending* until the binding has completed execution. The rules IStore and indirect behave much as the analogous store and indirect rules in the fully eager strategy (Figure 4-5). In addition, however, they take the work on the defer list and *resume* it, *i.e.* schedule it for execution.[1] The defer rule tells us that if the active term $x = S[y]$ requires a variable y which is still being computed—that is, it is sitting on the stack of any thread or on any defer list—then the current work $(x = S[y]\,;\ b_0)$ should be added to the defer list b_1 for y.

The rules also describe the *pH* approach to *work stealing*. Evaluation begins with a fixed number of threads (which correspond in this strategy to the processors of a multiprocessor). All but one of these threads is initially empty. When a thread is empty, it must *steal*: a *victim* is chosen at random from the threads with non-empty stacks, and the bottommost piece of work is removed. This work becomes the new stack for the *thief*. Thus, threads do not maintain a stack, but rather

[1] Note that there is a policy decision here: the work is added to the *top* of the work stack. Other placements of the work are possible, including auxiliary structures to hold pending resumptions. The strategy given here is the one used in the original *pH* implementation.

$$
\begin{array}{lcll}
h \bullet \langle (x = v\,;\, b)\,;\, k\rangle\ r \parallel t & \equiv & x = v,\ h \bullet \langle b\rangle\ \langle k\rangle\ r \parallel t & \text{(IStore)} \\
h \bullet \langle (x = y\,;\, b)\,;\, k\rangle\ r \parallel t & \equiv & x = y,\ h \bullet \langle b\rangle\ \langle k\rangle\ r \parallel t & \text{(indirect)} \\
 & & & \\
\langle (x = \textbf{letrec}\ b_0\ \textbf{in}\ e_1\,;\, b_1)\,;\, k\rangle & \longrightarrow & \langle b_0\,;\, (x = e_1\,;\, b_1)\,;\, k\rangle & \tau_f\ \text{(spawn)} \\
 & & & \\
h \bullet \langle (x = S[y]\,;\, b_0)\,;\, B[y = e\,;\, b_1]\rangle\ r \parallel t & & & \\
\qquad\qquad \equiv & & h \bullet \langle B[y = e\,;\, (x = S[y]\,;\, b_0),\ b_1]\rangle\ r \parallel t & \text{(defer)} \\
 & & & \\
h \bullet \langle (x = S[y]\,;\, b_0)\,;\, k\rangle\ B[y = e\,;\, b_1] \parallel t & & & \\
\qquad\qquad \equiv & & h \bullet \langle k\rangle\ B[y = e\,;\, (x = S[y]\,;\, b_0),\ b_1] \parallel t & \text{(defer)} \\
 & & & \\
h \bullet \langle (x = S[y]\,;\, b_0)\,;\, k\rangle\ r \parallel B[y = e\,;\, b_1] & & & \\
\qquad\qquad \equiv & & h \bullet \langle k\rangle\ r \parallel B[y = e\,;\, (x = S[y]\,;\, b_0),\ b_1] & \text{(defer)} \\
 & & & \\
h \bullet \epsilon \parallel r\langle k\rangle \parallel t & \equiv & h \bullet \langle k\rangle \parallel r \parallel t & \text{(steal)}
\end{array}
$$

Figure A-1: Eagerness using defer lists and work stealing

a dequeue: local work is handled stack-fashion, but steals are handled queue-fashion. This work-stealing strategy is borrowed from the multithreaded language Cilk [27].

Note also that the spawn rule in Figure A-1 is virtually identical to the enter block rule in the strict strategy (Figure 4-4). This highlights the similarity of the two strategies: by default, bindings are evaluated immediately in program order. If that is not possible, the strict strategy fails; the eager strategy suspends.

The indirect rule is optional in an eager calculus. Given a binding $x = y$ we can instead attempt to instantiate y and suspend if the attempt fails. This was the approach used in the implementation of Id. Its advantage is that no indirection mechanism is necessary. The disadvantage (beyond the cost of keeping two copies of y) is that trivial computations $x = y$ find their way onto defer lists that would otherwise have been empty. In effect, instead of creating indirections the system creates deferred "copying work". This subtly alters the termination semantics of the language in the presence of barriers [18].