Trace-based Just-in-time Compilation for Lazy Functional Programming Languages

Thomas Schilling

School of Computing

University of Kent at Canterbury

A thesis submitted for the degree of

Doctor of Philosophy

April 2013

Abstract

This thesis investigates the viability of trace-based just-in-time (JIT) compilation for optimising programs written in the lazy functional programming language Haskell. A trace-based JIT compiler optimises only execution paths through the program, which is in contrast to method-based compilers that optimise complete functions at a time. The potential advantages of this approach are shorter compilation times and more natural interprocedural optimisation.

Trace-based JIT compilers have previously been used successfully to optimise programs written in dynamically typed languages such as JavaScript, Python, or Lua, but also statically typed languages like Java or the Common Language Runtime (CLR). Lazy evaluation poses implementation challenges similar to those of dynamic languages, so trace-based JIT compilation promises to be a viable approach. In this thesis we focus on program performance, but having a JIT compiler available can simplify the implementation of features like runtime inspection and mobile code.

We implemented *Lambdachine*, a trace-based JIT compiler which implements most of the pure subset of Haskell. We evaluate Lambdachine's performance using a set of micro-benchmarks and a set of larger benchmarks from the "spectral" category of the Nofib benchmark suite. Lambdachine's performance (excluding garbage collection overheads) is generally about 10 to 20 percent slower than GHC on statically optimised code. We identify the two main causes for this slow-down: trace selection and impeded heap allocation optimisations due to unnecessary thunk updates.

Acknowledgements

I would like to thank my supervisor Simon Thompson for his advice and support throughout this work. Our meetings helped clarify my thoughts and helped direct my work.

I would like to thank in particular David Peixotto for his valuable discussions about trace-based compilation and his help on an earlier prototype of Lambdachine.

I would also like to thank my fellow Ph.D students who made my time as a Ph.D student a great experince, notably Edward Barrett, Matthew Mole, Edward Robbins and Martin Ellis.

Many thanks also to the anonymous reviewers of my papers for their comments and feedback.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis describes an implementation of the lazy functional programming language Haskell.

Haskell emphasises program correctness and succinctness through a combination of features and language design decisions. Haskell separates *pure* computations from *effectful*, or *impure*, computations. A pure computation does not have any side effects and its result depends only on its inputs. Side effects include reading or writing files, writing to the terminal or reading or modifying mutable state. A pure computation will produce the same result no matter when or how often it is executed. Due to these properties pure portions of a program are generally easier to test and can be made to execute in parallel.

Haskell also uses *non-strict* evaluation by default: arguments to functions are only evaluated if (and when) their value is needed. This can be used, for instance, to define custom control flow combinators:

```
myif :: Bool -> a -> a -> a
myif True  e1 e2 = e1
myif False e1 e2 = e2
```

This function always needs to evaluate its first argument in order to determine whether it is `True` or `False`, but it only evaluates one of `e1` and `e2`, never both. We say that `myif` is *strict* in its first argument.

While the Haskell Standard defines it as a non-strict language, most Haskell implementations actually implement *call-by-need* which is also known as *lazy* *evalualion.* With lazy evaluation a variable is only evaluated once and the result is shared between all uses of the same variable. For example:

```
f b x = myif b 42 (x + x * 3)
test b = f b (nthPrime 1000000)
```

The variable `x` is used twice and its value is only needed if `b` evaluates to `False`. Nevertheless, in a call-by-need implementation of Haskell the expression `nthPrime 1000000` is evaluated at most once.

Apart from making it quite easy to implement custom control structures, laziness can also enable better modularity as demonstrated by Hughes [1989]. Lazy evaluation allows separating the data consumer from the producer. The producer is lazy and only produces data when demanded by a consumer. A basic example of this is the following:

```
f n = sum (takeWhile (\x -> x < 142) (filter odd [n..n+100]))
```

The `sum` function takes a list of numbers as its argument and consumes them to produce their sum. This in turn requires that `takeWhile` produces some values which in turn it gets from `filter` which in turn gets it from the enumeration expression `[n..n+100]` which lazily produces the numbers from `n` to `n+100`. As soon as any of the consumers stops requesting values or any of the producers stops producing values, the computation finishes.

This style of composition is very flexible and leads to many reusable combinators. It also gives rise to interesting optimisation challenges for implementers of Haskell compilers.

1.1 Deforestation

A naïve implementation of lazy evaluation unfortunately is rather inefficient. For example, the following expression will allocate $3 \times$ `n` cons cells, one for each step in the composition:[1]

[1] The expression `(+1)` is a function that adds one to its arguments; the function `(*2)` doubles its argument.

```
f1 n = sum (map (+1) (map (*2) [1..n]))
```

In this example, each composition step performs very little computation, so the memory management overhead will impact performance significantly. We would like a compiler to optimise this program into something similar to this less elegant but more efficient program, which does not perform any allocation of lists:

```
f2 n = loop 1 0
  where
    loop i res =
      if i > n
        then res
        else loop (i + 1) (res + (i*2) + 1)
```

The process of removing such short-lived intermediate data structures is called *deforestation* (Wadler [1990]). The most widely used Haskell implementation, the Glasgow Haskell Compiler (GHC), supports a number of deforestation techniques.

First, however, let us look at a technique that does *not* work well. The two key optimisations that a Haskell compiler performs are simplification and inlining. The problem is that `sum`, `map`, and `[`n`..`m`]` are all recursive functions. Inlining a recursive function will give rise to another call to the same recursive function. The question then becomes when to stop inlining and how to replace the recursive function call to a recursive call to an optimised version of the original function. A technique called *supercompilation* tackles these issues (Bolingbroke and Peyton Jones [2010]; Mitchell [2010]), but so far no implementation has been proven to be practical for everyday use.

Current practical implementations of deforestation rely on GHC's support for custom rewrite rules. A rewrite rule is an instruction to the compiler to replace certain expressions with another (presumably more efficient) equivalent expression. The rule

```
{-# RULES
  "map/map" forall f g xs.
     map f (map g xs) = map (f . g) xs
  #-}
```

will instruct the compiler to replace any program expression that matches the pattern to the left of the equation to be replaced by the expression on the right hand side. Some variables are quantified via `forall` and will match any expression. This rule will apply to our example as follows:

```
   sum (map (+1) (map (*2) [1..n]))        [apply map/map]
⇒ sum (map ((+1).(*2)) [1..n])            [inline (.)]
⇒ sum (map (\x -> (+1) ((*2) x)) [1..n])  [simplify]
⇒ sum (map (\x -> (x * 2) + 1) [1..n])
```

The application of this rule has removed one intermediate list. We might be tempted to add more rules to handle other combinations of functions, but that will lead to an increasing number of rules for each new function we add.

The short cut deforestation (Gill et al. [1993]) and the stream fusion (Coutts et al. [2007]) frameworks address this for functions operating on lists. Short cut deforestation expresses all list consumers in terms of the function `foldr` and all producers in terms of the function `build`. A single rewrite rule then performs the deforestation step:

```
{-# RULES
  "foldr/build" forall f c g. foldr c f (build g) = g c f
  #-}
```

Stream fusion expresses all list producers in terms of streams, which store a current state and a function that given the current state computes the next element in the sequence and a new state. Stream functions are no longer recursive, so GHC's inliner can optimise streams like any other code.

1.2 Fragile optimisations

Each of these optimisation frameworks relies on the compiler to inline enough definitions to expose the optimisation potential. This is nothing new and it is a problem shared by all static optimisers. A concern more specific to Haskell is that the performance difference between the optimised and the unoptimised version can be an order of magnitude.[1]

If for some reason the optimiser did not inline an important definition, then the user has to either rewrite the program to the expected version or diagnose why the compiler did not produce the expected efficient version and somehow modify the program so that the compiler will optimise it. Neither choice is desirable.

Rewriting the program into a hand-optimised version loses most of the benefits of using a high-level language like Haskell. Other programming languages like C also provide better support for low-level optimisations, but at the cost of generally higher implementation effort.

Analyzing why the compiler did not optimise the program as expected requires a good understanding of the inner workings of the compiler, its intermediate languages and patience. Once the reason is located the compiler somehow has to be instructed to perform the desired optimisation step. Usually this is done in the form of a rewrite rule or an annotation to the compiler (a "pragma") to inline a certain function more aggressively. Such instructions can be made part of a library, though care must be taken to avoid negative side effects for some users of the library; e.g., too much inlining can increase the size of the resulting program and the time it takes to compile it. Other times the user must rewrite the program to help the compiler perform the necessary optimisations. In any case, all of these methods are fragile and may break with small changes to the source program or with different versions of the compiler.

Another complication is that rewrite rules can be sensitive to the exact shape of the program. For example, the `map/map` rule will not be applied if the program looks as follows:

```
f ys = map (*2) (let z = length ys in map (+z) ys)
```

In this case, the compiler can push the outer `map` call inside the `let` binding, and then the rule will apply. This does not work, however, if there is a function call in between the nested call to `map`, then no simple local transformation can get the rule to apply.

This thesis investigates one approach that has the potential to reduce this fragility. Instead of only optimising the program at compile time, we also employ

[1]We show this in Chapter 6 for the benchmark *SumStream* in Table 2 on page 185.

just-in-time (JIT) compilation to optimise the program at runtime. Not everything is JIT compiled, only the parts that are executed frequently. These are also the parts of the program where good performance is most important.

A JIT compiler operates at runtime which allows it to gather statistics about the program's behaviour at runtime. This profiling data can be used to inform inlining decisions which in turn will enable further optimisations. A JIT compiler is thus less likely to miss optimisation opportunities where they are most important. The profiling data can furthermore be used to create specialised code and the JIT compiler can perform speculative optimisations that would be impractical for a static compiler because a static compiler would have difficulty guessing which speculative optimisations are worthwhile.

We cannot expect a JIT compiler to solve all performance problems. Nevertheless, the runtime statistics collected by the JIT compiler can be used to give insights into where optimisation efforts should be spent. The availability of a JIT compiler also enables features of a runtime system that are difficult to impossible to achieve with only a static compiler, as discussed in the next section.

1.3 A virtual machine for Haskell

This thesis describes Lambdachine: a virtual machine for Haskell. It optimises Haskell programs in two stages. First, the program is optimised statically using GHC. While GHC would then translate the optimised program into machine code, we instead generate a bytecode instructions. This bytecode is then loaded at runtime and interpreted. The interpreter collects certain runtime statistics which are then used to optimise the program at runtime.

Because the JIT compiler runs alongside the program it must be very efficient. Any time the compiler spends optimising the program is time not spent executing the user's program. Before the JIT compiler optimises the program, the program is also executed in a slower execution mode. Commonly this is done either through an interpreter or a simple non-optimising code generator. Profiling must be used to identify which parts of the program will benefit from being optimised by the JIT compiler, that is, for which parts of the program the JIT compilation overhead is likely to pay off over the total execution time of the program.

There are two widely used styles of JIT compilers used to implement programming languages: method-based JIT compilers and trace-based JIT compilers. A method-based JIT compiler optimises one function at a time, while a trace-based JIT compiler optimises traces—straight-line sequence of instructions with no inner control flow.

Lambdachine uses a trace-based JIT compiler. This was done for a variety of reasons, most importantly because a trace-based JIT compiler is comparatively simple and it seems to be a good fit for the way Haskell programs are executed.

1.4 Trace-based Just-in-time Compilation and Non-strict Evaluation

A trace-based JIT compiler starts by detecting frequent targets of branch instructions, and records the instructions executed after execution reached such a target. The goal is to detect the inner loops of the program and then optimise them as a whole, regardless of which functions were called inside of the loop. This appears to fit well with non-strict evaluation where execution jumps around between parts of different functions, but is always driven by some consumer.

The idea is to pick the right heuristics to detect loops inside strict consumers and use the trace compilation machinery to remove the overheads introduced by non-strict evaluation. Because the trace simply follows the execution of the program it automatically optimises across function boundaries. The simple structure of traces also simplifies the implementation of compiler optimisations which can reduce the compilation time.

The downsides of a trace-based approach are that optimising across traces is more difficult or may again increase the compiler's implementation complexity. This makes a trace compiler very sensitive to the trace selection heuristic, since that will decide how much the trace compiler can optimise. Furthermore, since traces do not contain inner control flow they must duplicate code occurring after a control flow join point which can lead to excessive code duplication under some circumstances.

1.5 Thesis Statement and Research Contributions

This thesis investigates the applicability of trace-based just-in-time compilation to a call-by-need language; specifically the Haskell programming language. We focus on the performance and optimisation potiential. To this end, we implemented a prototype which supports a meaningful subset of Haskell and evaluate it using a set of micro benchmarks and application kernel benchmarks.

The research contributions of this thesis are:

- Our prototype, Lambdachine, supports integer and character types and operations on them, string literals represented as byte arrays, and user-defined data types (which includes almost all monads). Lambdachine also supports GHC's `State#` type, which is used to implement the `IO` monad, but not the C foreign function interface which is required to actually perform any input/output operations.

  We rewrote benchmarks that print their output to instead compare the benchmark result with the expected value. Lambdachine also does not yet support arrays, mutable references or floating point numbers.

  Haskell programs often use immutable tree structures and pure computations, which are supported by our subset. We do not expect I/O heavy programs to benefit (or suffer) from a JIT compiler.

- We designed a bytecode instruction set with two design goals: (1) allow the implementation of an efficient interpreter which is used before the JIT compiler is invoked, and (2) preserve enough high-level information to allow the JIT compiler to perform optimisations that take advantage of invariants provided by the language semantics. Section 5.2 describes this bytecode language.

- We implemented a compiler from GHC's Core intermediate language to this bytecode (Section 5.3) and an interpreter for the bytecode (Section 5.2.9). The bytecode interpreter is not heavily optimised because in our benchmarks only a small fraction of the execution time is spent in the interpreter.

- We implemented a trace compiler that records traces via the interpreter and compiles them to machine code (Sections 5.5–5.14). The trace compiler uses some code and many of the implementation techniques of the open source trace-based JIT compiler LuaJIT (Pall [2013]). The intermediate language used by our trace compiler, the *trace IR* (Section 5.5), is similar in format, but quite different in semantics from LuaJIT's IR. The similarities, however, allowed us to reuse large parts of LuaJIT's register allocator and code generator with few changes. The trace compiler, therefore, is very fast (typically $< 1ms$ per trace), but trace selection is a key ingredient to make the optimiser effective.

- We implemented standard trace selection heuristics from the literature, but they turned out not to be particularly effective for larger programs (Section 6.4). Many traces have a low completion ratio; i.e., execution often does not reach the end of the trace, but leaves the trace early. The trace compiler optimises based on the assumption that traces are rarely exited early, so a low completion ratio will make the optimiser less effective.

- We evaluate our prototype using six micro-benchmarks and four small application kernel benchmarks (Chapter 6). While Lambdachine can beat GHC's static optimsier on all but one micro-benchmark, Lambdachine is normally slower than GHC on larger benchmarks. We attribute this to poor trace selection and an important optimisation not being applicable at runtime.

- We identify that *updates*—the key difference between a call-by-name and a call-by-need evaluator—are causing important optimisations from occurring (Section 6.6). In particular, *unnecessary* updates are the problem and we propose a dynamic tagging scheme to enable the JIT compiler to safely determine at runtime whether an update can be omitted (Section 6.8).

A report about Lambdachine during an earlier state was published in Schilling [2012]. This report included performance estimates for micro-benchmarks, but no absolute performance numbers on actual hardware.

Lambdachine is open source software. It is available at: `https://github.com/nominolo/lambdachine`.

1.6 Other Uses of a VM-based System

This thesis focuses on the performance aspects of using a JIT compiler. There, however, are other advantages of having an environment that can optimise a program at runtime. We do not address these aspects in this thesis, but they are interesting avenues for future research.

Better Introspection GHC allows programs to be compiled in different modes, such as different profiling modes or with or without multi-threading support and others. For example, GHC version 7.0.4 supports the following "ways": `prof` (profiling support), `eventlog` (generate logs of runtime system events), `threaded` (the multi-core runtime), `debug` (selects the debug runtime), `dyn` (compiles for dynamic linking), `ndp` (enable nested data parallelism support), and other undocumented ways. Some ways may be combined with others resulting in a total of 32 possible combinations. The profiling mode changes the object layout and calling convention meaning that all dependencies must also be compiled in profiling mode, thus users often have to compile every library (at least) twice if they ever plan to use profiling with their application.

In a virtual machine it is much simpler to change the calling convention by including multiple interpreters in the runtime and changing the way that optimised code is generated. Any code duplication occurs only once in the runtime system and normally is restricted to fairly small parts. For example, there are flexible interpreter designs which allow swapping the implementation of individual instructions. The code of instruction whose behaviour is unaffected by the mode need not be duplicated.

In some cases it may also be possible to switch the mode at runtime, or select the execution mode for individual functions. This could be useful to track down issues in a running production system while avoiding performance degradation in the normal case.

Dynamic Code Loading There is some support in GHC for loading compiled code at runtime. Having a portable bytecode format combined with JIT compilation could make this simpler and more flexible. One possible use case is "hot-swapping", which is updating a running program with a new version. There are, however, other problems that are more difficult to solve, such as how to ensure that all heap objects are updated to their new format.

Dynamic linking without performance overhead. Dynamic linking of libraries has the potential to reduce the memory footprint of applications if multiple applications use the same library. Multiple applications could share the same library code at runtime, thus the library code only needs to be loaded once.

This only works if that library is compatible with each application. Compatibility is expressed using an application binary interface (ABI) version number. The ABI includes the semantic version as well as calling convention and other low-level details. If changes to the library do not require changing the ABI version, then existing programs can transparently use a new version of the library. Inlining across library boundaries, however, inhibits reusability. If an application inlines functions from a library, then the application depends no longer just depends on the interface of the library, but also on the particular implementation of the inlined function. To maximize compatibility a static compiler thus would have to choose to not inline functions across package boundaries, which may cause performance degradation due to the overhead of additional function calls and missed optimisations.

A compiler that operates at runtime does not have this issue because inlining is performed at runtime and uses the code of whichever version of the library is available at runtime.

Platform portability. Virtual machines have often been used to achieve platform independence, the famous goal being "write once, run anywhere". In practice, it is very difficult to achieve full platform independence due to the need to also abstract higher-level differences than the CPU architecture. For example, graphical user interfaces or I/O subsystems require good abstractions at the library level. Nevertheless, a virtual machine and a portable bytecode can make it easier to obtain achieve portability.

Mobile code. Many modern applications use a distributed architecture consisting of many programs running on different machines, possibly even at opposite sides of the planet. Being able to send code as data from one machine to another can be very useful. For example, consider an application that runs analyses over financial data. A user in, say, New York, may run her analysis over the stock data from, say, London's stock market. Transmitting all the stock data from London all the way across the Atlantic Ocean over to New York would be a lot more expensive than sending the analysis code to London to be executed on servers near where the data is hosted.

Being able to send across portable byte code which only later gets compiled into platform-specific machine code can make implementing this feature much easier. For example, Cloud Haskell (Epstein et al. [2011]) essentially requires users to perform manual closure conversion and run the same program binary on both machines.

1.7 Thesis Overview

The following Chapter 2 discusses some background to help understand the issues involved in implementing Haskell efficiently. Chapter 3 then describes trace-based just-in-time compilation and the design decisions that implementers of such a compiler must face. To give a better understanding of the full compilation pipeline Chapter 4 follows the transformation of a small Haskell program all the way to machine code. Chapter 5 then describes our implementation, Lambdachine, in detail. In Chapter 6 we discuss how well Lambdachine performs for a set of benchmark programs and investigate the issues involved. Chapter 7 discusses related work, and the final Chapter 8 discusses possible future work and concludes this thesis.

Chapter 2

Background

This chapter discusses some background related to the implementation and optimisation of lazy functional programs, specifically Haskell.

2.1 Graph Reduction

Haskell's execution model is based on graph reduction. Figure 2.1 shows a simple example of graph reduction for the expression $f\ (3 * 7)$ where $f = \lambda x.(x + x)$. The expression $f\ (3 * 7)$ is represented as a tree with the label "@" standing for application. Evaluation proceeds by replacing the application of f to an argument with the body of f. The formal parameter x is replaced by the argument. Since the body of f mentioned x twice, the tree has now become a graph. Evaluation of the $+$ node now requires that the expression $(3 * 7)$ is evaluated as well.

Note how the expression $(3 * 7)$ was not evaluated until its value was actually needed. It was also evaluated only once because the expression was shared by both sides of the $+$ node. This evaluation strategy is known as lazy evaluation or call-by-need.

This example is only a conceptual model. Modern implementations do not always construct the graph in memory; the exact representation will instead depend on the particular design of the implementation model.

Graph reduction has been implemented both on standard hardware as well as using custom hardware. Because graph reduction is quite different from the tra-

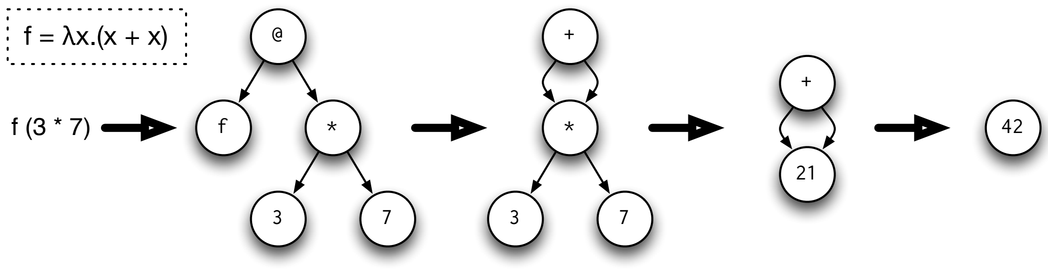


Figure 2.1: Simple example of graph reduction.

ditional von-Neumann style execution model using custom hardware is appealing and can be quite efficient (Naylor and Runciman [2008], Naylor and Runciman [2010]). Unfortunately, it is nearly impossible to keep up with the amount of money and effort spent to improve traditional hardware. Additionally, custom hardware makes it difficult to interface with existing software written for standard hardware.

Much research has gone into implementing graph reduction efficiently on standard general purpose processors. A number of abstract machines have been developed to serve as a suitable abstraction layer with a simple mapping from a call-by-need source language and an efficient mapping from the abstract machine to machine code of the target platform. These abstract machines include the G-machine (Johnsson [1987], Augustsson [1987]), the Spineless G-machine (Burn et al. [1988]), the Spineless Tagless G-machine (Peyton Jones [1992]), the ABC machine (Plasmeijer and van Eekelen [1993]), and GRIN (Boquist [1999]; Boquist and Johnsson [1996]).

The most widely used Haskell implementation, GHC, uses the *Spineless Tagless G-machine* (STG) (Peyton Jones [1992]). STG requires the use of a higher-order construct, namely the use of indirect function calls. Boquist and Johnsson [1996] used the GRIN language and whole-program analysis to translate a Haskell program into only first-order constructs such as C's `switch` statement or calls of known functions.

2.2 The Source Language

Figure 2.2 shows our input language, called CoreHaskell. CoreHaskell is a simple, untyped subset of Haskell. We do not require users to write programs in this language. Rather, it is intended to be produced as the output of some Haskell compiler front-end. In fact, CoreHaskell is the output of GHC's `CorePrep` pass with the types erased. It is designed to make allocations (**let**) and the order of evaluation (**case**) explicit.

Operationally, **let** $x = b$ **in** e causes b to be allocated on the heap and stores a reference to that heap object in x. If b is a constructor application then an object with that constructor is allocated. If b is a variable, a function with no free variables, or a constructor with no arguments then no allocation is necessary and x will simply be assigned a reference to the static object denoted by b. For any other terms an object is allocated on the heap that encodes the unevaluated expression. Section 2.4 discusses this further.

Let-expressions are recursive, thus the binding b may refer to itself through the variable x. If the **let** expression contains multiple bindings they may refer to each other. For example,[1]

```
let one = CInt 1 in
let l = Cons one l in
l
```

constructs an infinitive list of ones, whereas the expression

```
let one = CInt 1 in
let two = CInt 2 in
let l1 = Cons one l2
    l2 = Cons two l1
in l1
```

returns an infinite list alternating between one and two.

[1] We use `CInt` as the constructor for heap-allocated `Int`s and write `Cons` and `Nil` for Haskell's list constructors `(:)` and `[]`.

Variables

x, y, z, f, g

Primitives

$p \quad ::= \quad + \mid - \mid < \mid \leq \mid ...$

Constructors

$K \quad ::= \quad \mathsf{True} \mid \mathsf{False} \mid \mathsf{Cons} \mid ...$

Literals

$\ell \quad ::= \quad$ 1 | 2 | ... | `'a'` | `'b'` | ...

Atoms

a	$::=$	ℓ	Literal
	$\mid$	x	Variable reference

Terms

e	$::=$	a	Atom
	$\mid$	$f\ \overline{a}$	Function application
	$\mid$	$p\ \overline{a}$	Saturated primitive application
	$\mid$	$K\ \overline{a}$	Saturated constructor application
	$\mid$	$\textbf{let}\ \overline{x = b}\ \textbf{in}\ e$	Recursive let / allocation
	$\mid$	$\textbf{case}\ e\ \textbf{of}\ (x)\ \overline{\rho \to e}$	Pattern matching

Binders

b	$::=$	$\lambda\overline{x}.e$	Lambda abstraction
	$\mid$	e	Other expression

Patterns

ρ	$::=$	$K\ \overline{x}$	Constructor pattern
	$\mid$	ℓ	Literal pattern
	$\mid$	$\_$	Wildcard

Programs

p	$::=$	$\overline{f = e}$	Top-level definition

Figure 2.2: The CoreHaskell language.

The **case** expression is used both to evaluate expressions and for pattern matching. Pattern matching only operates on the outermost constructor. The variable after the **of** keyword is bound to the result of evaluating the expression.

Haskell supports nested patterns, but CoreHaskell only supports matching on the outermost constructor. Nested matches in Haskell can be translated into multiple **case** expressions in CoreHaskell. For instance:

```
evenLength :: [a] -> Bool   -- In Haskell
evenLength [] = True
evenLength [x] = False
evenLength (x:y:xys) = evenLength xys

-- In CoreHaskell
evenLength list =
  case list of
    [] -> True
    x:xs ->
     case xs of
       [] -> False
       y:xys -> evenLength xys
```

Applications of constructors and primitives must always be saturated, that is, the number of arguments must match exactly the number of formal parameters of the constructor or primitive. Haskell expressions that partially apply constructors or primitives must be translated into appropriate λ-terms. For example, the Haskell expression

```
consFive :: [Int] -> [Int]
consFive = (5:)
```

when translated into CoreHaskell becomes

```
five = CInt 5
consFive = λ xs -> Cons five xs
```

2.3 Non-strict Evaluation

Haskell uses *non-strict* evaluation by default. This means that arguments to a function are not evaluated before they are passed to the function. Arguments are only evaluated if the called function actually needs its value. For example, given the following definition of the function `double`

```
double :: Int -> Int
double x = x + x
```

a *strict* language would evaluate the expression `double (3 * 4)` as follows:

```
double (3 * 4)    ⇒
double 12         ⇒
12 + 12           ⇒
24
```

In a non-strict language this expression is evaluated as follows:

```
double (3 * 4)      ⇒
(3 * 4) + (3 * 4)   ⇒
12 + (3 * 4)        ⇒
12 + 12             ⇒
24
```

In this example, the expression `(3 * 4)` was evaluated twice. That is, the compiler chose to implement Haskell's non-strict semantics using *call-by-name*. If the evaluated expression does not have any side effects then the compiler can cache the result. This is known as *call-by-need* or *lazy evaluation*. The evaluation sequence becomes:

```
double (3 * 4)          ⇒
let x = 3 * 4 in x + x  ⇒
let x = 12 in x + x     ⇒
let x = 12 in 24
```

The value of `x` is no longer needed and will eventually be garbage collected.

While the Haskell standard only prescribes a non-strict evaluation strategy, almost all Haskell implementations actually implement non-strictness using call-by-need instead of call-by-name.

2.3.1 Weak Head Normal Form

Non-strictness also means that expressions are evaluated only as much as needed and no further. Evaluation stops as soon as it reaches a data constructor. Arguments of the constructor are not evaluated. For example, given a function

```
f :: a -> List a
f x = Cons x Nil
```

then evaluating the expression `f (6 * 7)` only performs a single reduction step (that of applying the function):

```
f (6 * 7) ⇒
Cons (6 * 7) Nil
```

The argument `6 * 7` is not evaluated since `Cons` already is a constructor. More formally, evaluation reduces an expression to *weak head normal form.*

Definition 1. *An expression is in* weak head normal form *(WHNF) if either of these conditions hold:*

- *It is of the form $C\ e_1\ \cdots\ e_n$ where C is a constructor of arity n, and $e_1, \cdots, e_n$ are arbitrary expressions. This includes zero-arity constructors.*
- *It is a partial application, i.e., it is of the form $f\ e_1\ \cdots\ e_k$ where f is a function of arity n, $k < n$, and $e_1, \cdots, e_n$ are arbitrary expressions.*
- *It is a literal.*

In our core language (Section 2.2), evaluation is triggered by `case` expressions which also supports pattern matching on the result.

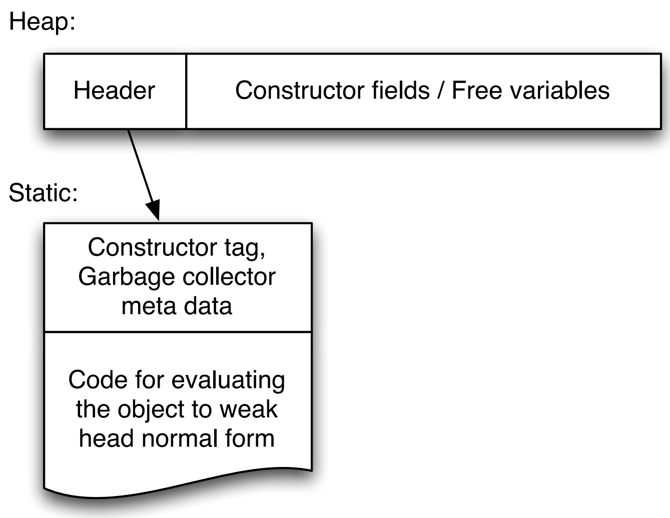

Heap:
Header
Constructor fields / Free variables
Static:
Constructor tag, Garbage collector meta data
Code for evaluating the object to weak head normal form

Figure 2.3: Possible layout of an object on the heap. The payload for thunk `x + y * 2` would consist of the values for `x` and `y`. The code for evaluating the thunk to weak head normal form is shown in Figure 2.4.

Since unevaluated expression are represented as heap-allocated objects, some operations on primitive types obey strict evaluation rules. For example, an expression of type `Int`, i.e., a heap-allocated fixed-precision integer, may be an unevaluated expression, but a value of type `Int#`, which represents an underlying machine integer, is always in normal form. These primitive types, however, are implementation-specific and are used mostly by low-level code. We discuss thes

2.4 Implementing Lazy Evaluation

Unevaluated expressions are typically implemented using heap-allocated objects called *thunks* that contain the free variables of the expression and a pointer to the code for reducing the expression to WHNF. A call-by-need implementation then overwrites the thunk with its evaluation result, while a call-by-name implementation would not.

For instance, the expression `x + y * 2` contains two free variables, thus its thunk may look as shown in Figure 2.3. The object header identifies the object

```
1   Object *x = Node[1];  // load free variable "x"
2   if (!inWHNF(x))
3     x = EVAL(x);

4   Object *y = Node[2];  // load free variable "y"
5   if (!inWHNF(y))
6     y = EVAL(y);

7   int x_raw = UNBOX_INT(x);
8   int y_raw = UNBOX_INT(y);
9   int result_raw = x_raw + y_raw * 2;

10  Object *result = BOX_INT(result_raw);
11  UPDATE(Node, result);
12  return result;
```

Figure 2.4: Evaluation code for the thunk of `x + y * 2` (in pseudo-C).

layout to the garbage collector and contains a pointer to the code that evaluates the thunk to WHNF. Figure 2.4 shows this evaluation code for our example expression in pseudo-C.

The code first loads the values of the free variables via the special `Node` variable (lines 1 and 4). The `Node` variable always points to the currently evaluated thunk. It thus serves a similar role as the `this` or `self` variable in object-oriented languages.

Both `x` and `y` may itself be thunks, so they first must be evaluated to WHNF. The implementation of `inWHNF` and `EVAL` is discussed in Section 2.4.2. Now that `x` and `y` are guaranteed to be in WHNF their payload values can be extracted to compute the expression's result.

The `UNBOX_INT` operation extracts the integer from a heap-allocated integer object (which must be in WHNF). This can be implemented in different ways. For example, in GHC heap-allocated integers are heap objects with a single field. Retrieving the integer value thus requires a memory load. Other implementations may choose to use a tagging scheme to indicate if a value is a pointer to a heap object or an integer. Retrieving the integer value would then amount to removing the tag bits.

2.4.1 Thunk Update

After computing the result, `BOX_INT` turns it back into an object reference either by tagging, or by allocating a heap object. This object (`result`) is then used to *update* the current thunk with its value (line 11).

Updating of thunks distinguishes lazy evaluation from other forms of non-strict evaluation. Note that updating is only necessary if there is a possibility that the thunk is evaluated again in the future.

Updating can be implemented either by directly overwriting the thunk object with the new object, or by using an *indirection*. Using an indirection means that we overwrite the header word of the thunk with another header that indicates that the object has been evaluated. The payload of the object (which previously contained the free variables of the expression) is then overwritten with a pointer to the new value. If the same object is evaluated again the evaluation code will immediately return the cached value.

Directly overwriting the thunk seems like a reasonable choice, however, it causes some complications:

1. The new object may be larger than the thunk. In that case we have to support a way to split objects, or also support using an indirection.

2. Even assuming the size of the object is not an issue, we would still have to copy the whole object. An indirection only requires two memory writes (one for the header, and one for the pointer to the new value).

3. In a concurrent implementation we may want to atomically replace the thunk with its new value. This also favours a solution with fewer writes.

GHC chooses to use indirections and employs some help from the garbage collector to reduce the amount of pointer-chasing required. Whenever the garbage collector traverses the heap, it automatically rewrites all pointers to an indirection to point to the indirectee instead. In the long run this removes almost all indirections. The implementation described in this thesis uses the same technique.

2.4.2 Checking for Weak Head Normal Form

There are a number of ways to check whether a value is in WHNF. One way, is to look up the meta data associated with the object using information from the header. In GHC, the header is actually a pointer to the meta data (the info table). Checking if the object is in WHNF could then be done by checking a bit in the info table:

```
bool isWHNF(Object *o) {  // inlined at each use site
  InfoTable *info = o->header;
  return info->isWHNF;
}
```

This requires following two pointers, one to load the pointer to the info table from the object, and another to then load the bit from the info table itself. Checking whether an object is in WHNF is a very frequent operation, so this can cause a lot of overhead.

The bit indicating whether an object is in normal form could be stored in the header itself, but that would require either masking that bit whenever the pointer to the info table needs to be followed (if the bit is stored with the pointer to the info table), or a larger header (if the bit is stored separately from the pointer).

For a long time GHC instead decided to never perform this check and unconditionally evaluate objects. The code for evaluating an object that is already in WHNF would return immediately and return a pointer to the object itself as the result. Evaluation still requires dereferencing two pointers, but if the object is not in WHNF then that would have been necessary anyway.

It turns out that in practice, most calls to `isWHNF` would return `true`, though. Therefore, if `isWHNF` could be implemented efficiently the performance of many programs could be improved. Marlow et al. [2007] performed these measurements and evaluated two tagging schemes to exploit this fact. Both schemes exploit the fact that on a 64 bit architecture all objects are aligned at a multiple of 8 bytes (64 bits). This implies that any pointer to such an object must be a multiple of 8 and thus all three least significant bits must be 0. Similarly, on a 32 bit architecture objects will be aligned at a 4-byte boundary, so the two least significant bits will

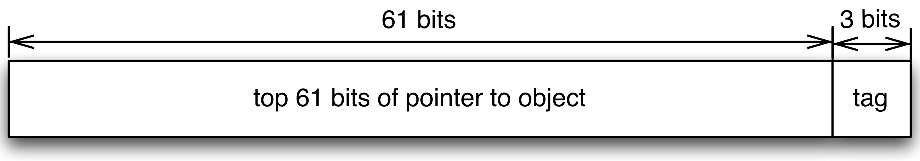


Figure 2.5: Pointer tagging. Because objects are aligned at 8 byte boundaries on a 64 bit architecture (or 4 bytes on a 32 bit architecture), the lowest 3 bits can be used to store other information (2 bits on a 32 bit architecture).

always be 0. This leaves room to store extra information in those bits provided that we ensure that this extra information is removed when dereferencing the pointer.

Marlow et al. [2007] evaluated two uses of these extra bits:

Semi-tagging. Only the least significant bit, call it t, is used to store whether the pointed-to object is in normal form. If $t = 1$, then the object is guaranteed to be in WHNF. If the $t = 0$, then the object *may* not be and we must evaluate it.

Constructor tagging. All available bits are used to encode a 2-bit or 3-bit number c. If $c > 0$ then the pointed-to object is in normal form and its tag is c. For example, for a list $c = 1$ could be the "`[]`" constructor, and $c = 2$ could mean the "`(:)`" constructor. If $c = 0$ then the object might not be in WHNF and it must be evaluated. If an object has more constructors than can be represented with c, it is always safe to use $c = 0$.

In both cases a pointer is almost always dereferenced in a context where the value of t/c is known. That means, no bit masking is necessary. To get the true address of a pointer p we simply access the address $p - t$ or $p - c$, respectively. The evaluation code always returns a tagged pointer and the garbage collector will add the right tag to each pointer to an object in the heap.

The semi-tagging scheme improved performance over an implementation without tagging by on average 7% to 9%. Marlow et al. [2007] argue that the increase in performance is mostly due to the reduced number of mispredicted branches (due to the indirect function call employed by `EVAL`).

The pointer tagging scheme performed even better, improving performance by on average 12.8% to 13.7%. Branch mispredictions have been reduced the same way as semi-tagging. The additional speedups are due to cache effects. A `case` expression normally has to look up the tag of the target from the info table (two pointer indirections). By storing the tag in the pointer, this look-up is avoided.

As Marlow et al. [2007] note, this level of improvement is indeed quite impressive because GHC was already a mature compiler at the time this dynamic optimisation was added. It could mean, that this moved GHC very close to the peak efficiency of Haskell's execution environment. It could also mean that there is more potential in using dynamic optimisation to improve performance of Haskell program.

2.5 Partial Application and Over-application

In Haskell a function `f` of type `A -> B -> C` can be implemented in two different ways. Most commonly it will be implemented as a function that requires two arguments (of types `A` and `B`) and then returns a result of type `C`. It could also be implemented as a function which expects a single argument (of type `A`) and then returns another function which expects a single argument (of type `B`) and then returns a result of type `C`. The Haskell type system does not distinguish these two cases, but an implementation of Haskell usually will due to efficiency concerns.

This ambiguity can be used to make programming in Haskell more convenient. For example, regardless of how `f` is implemented, applying it to one argument returns a function of type `B -> C`. We do not have to distinguish these two expressions:

$$\lambda\,\texttt{x -> f a x} \;\equiv\; \texttt{f a}$$

The problem is, if `f` is implemented as a function that expects two arguments, it has to be translated dynamically into something equivalent to the left-hand side of the above equality. We can avoid this overhead if the function being called is known at compile time. The compiler will look up which way `f` is implemented and generate an efficient calling sequence. For example, if `f` takes two arguments, then the compiler will supply both arguments at once. Otherwise, the compiler will translate a call like `f x y` to:

```
case f x of g -> g y
```

The compiler cannot do this if the function is a variable and the compiler does not know which function will be invoked at runtime. The compiler will then have to use a generic function call mechanism. GHC will translate this to a call to the right variant of its `apply` primitives (Marlow and Peyton Jones [2004]). The `apply` primitive, or the equivalent mechanism in the push/enter model (Marlow and Peyton Jones [2004]), must dynamically handle each of the three possible cases:

Exact Application If the function expected exactly the number of arguments that it was called with, then the function is simply called with the given number of arguments.

Partial Application If the function expected *more* arguments than were supplied, we must construct a partial application on the heap, which is referred to as a PAP. A PAP contains a pointer to the function being applied and all the arguments that have already been applied.

Over-application If the function expected *fewer* arguments than it was given, then the additional arguments must be saved and applied later. That is, the function is first called with the number of arguments it expected and once the function returns, the function's result—which must be a function—is called with the remaining arguments.

Figure 2.6 shows pseudo-code that handles all these cases for the case where a function is applied to two arguments. If the function argument is a thunk, it has to be evaluated first. The result must then be either a function or a partial application. In each case we must consider many possibilities.

In practise, a fair amount of executed function calls are calls to unknown functions (around 20%, but varying widely), but the vast majority of these calls

```c
/* Applies the function f to the two pointer arguments arg1 and arg2 */
Object *apply_pp(Object *f, Object *arg1, Object *arg2)
{
  if (!inWHNF(f))
    f = EVAL(f);

  if (isFunction(f)) {
    switch (arity(f)) {
    case 1: { /* too many args / over-application */
      Object *g = invoke(f, arg1);
      return apply_p(arg2); /* apply the remaining arg */
    }
    case 2: /* exact application */
      return invoke(f, arg1, arg2);
    default: { /* too few args / partial application */
      Object *pap = alloc_pap2(f, arg1, arg2);
      return pap;
    }
  } else { /* f is a PAP */
    switch (pap_arity(f)) {
      ...
    }
  }
}
```

Figure 2.6: Sketch of a possible implementation of a generic `apply` primitive.

are exact calls (Marlow and Peyton Jones [2004]). Nevertheless, each of these calls of an unknown function must go through the above decision process, adding overhead.

2.6 Type Classes and Overloading

Another interesting feature for Haskell implementers are *type classes* (Wadler and Blott [1989]). A type class is a mechanism for overloading functions to work over different types. For example, the type class for types whose values can be compared for equality is called `Eq` in Haskell. Two implementations of `Eq` for booleans and lists may look as follows.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)  -- default implementation

instance Eq Bool where
  False == False = True
  True  == x     = x
  _     == _     = False

instance Eq a => Eq [a] where
  []     == []     = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _      == _      = False
```

If we use the operators defined by the `Eq` type class on a concrete type, the compiler automatically figures out which implementation to use. The more interesting feature is that type classes can be used to abstract over the implementation of certain operations. For example, the function `elem` works over any type that is an instance of `Eq`.

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

Type classes usually also have associated laws that must hold for each implementation. For example, `==` should describe an equivalence relation and therefore should be reflexive, symmetric and transitive. These properties are not enforced by the Haskell compiler, but programs may break in subtle ways if such properties do not hold.

Type classes are commonly implemented using dictionaries. A *dictionary* is a record of functions (or constants) that implement the operations of the type class. The type class definition is translated by the compiler into a type declaration of the dictionary type. Type class instances are translated into values of this type. Figure 2.7 shows the result of applying this transformation to the `Eq` type class.

The dictionary type for `Eq` simply contains two functions which implement the two operations (Line 1). The type class methods take a dictionary as an additional argument, look up their implementation in that dictionary then tail-call the implementation with the appropriate arguments (Lines 2–4).

The implementation of the instance `Eq Bool` is stored in the dictionary `dictEqBool` (Line 6). The dictionary for `Eq [a]` is actually a function (`dictEqList`) because the instance declaration had the constraint `Eq a`. The intuitive reading of the instance `Eq a => Eq [a]` is: "If the type `a` supports testing for equality, then equality for lists of type `a` is defined as follows." The function `dictEqList` translates this intuition into the more operational "Given an implementation of equality for type `a`, this function provides an implementation of equality for `[a]`." The dictionary passed to `dictEqList` is indeed passed on to the implementation of the operations where it is used eventually in the implementation of `(==)` on Line 29.

2.6.1 Performance Overhead of Type Classes

In order to invoke the equality operation on lists we must construct a dictionary to pass to `(==)`. This means that dictionaries may actually be dynamically allocated (Lines 22,32). In the case of lists a new dictionary will be allocated for each

```
data DictEq a = DictEq (a -> a -> Bool) (a -> a -> Bool)

(==), (/=) :: DictEq a -> a -> a -> Bool
(==) (DictEq eq _) x y = eq x y
(/=) (DictEq _ ne) x y = ne x y

-- instance Eq Bool where ...
dictEqBool :: DictEq Bool
dictEqBool = DictEq dictEqBool_eq dictEqBool_ne

dictEqBool_eq, dictEqBool_ne :: Bool -> Bool -> Bool
dictEqBool_eq x y =
  case x of
    False -> case y of
               False -> True
               True -> False
    True -> y

dictEqBool_ne x y = not ((==) dictEqBool x y)

-- instance Eq a => Eq [a] where ...
dictEqList :: DictEq a -> DictEq [a]
dictEqList dict = DictEq (dictEqList_eq dict)
                         (dictEqList_ne dict)

dictEqList_eq, dictEqList_ne :: DictEq a -> a -> a -> Bool
dictEqList_eq dict xs ys =
  let dictEqListA = dictEqList dict in
  case xs of
    [] -> case ys of
            [] -> True
            _  -> False
    (x:xs') -> case ys of
                 [] -> False
                 (y:ys') -> (==) dict x y
                              && (==) dictEqListA xs' ys'

 dictEqList_ne dict xs ys =
   let dictEqListA = dictEqList dict in
   not ((==) dictEqListA xs ys)
```

Figure 2.7: Implementation of the type class Eq.

recursive call. Fortunately, in this example, the compiler can optimise this easily by inlining `(==)` at each use site.

Other overheads cannot be eliminated so easily. For example, consider the expression from Line 29

```
(==) dict x y
```

Inlining removes the overhead of the function call, but the resulting expression cannot be optimised any further:

```
case dict of
  DictEq eq _ -> eq x y
```

The expression `eq x y` is a call of an unknown function which is more expensive than a call to a known function. This inefficiency is simply the cost of abstraction. It is also no less efficient than calling a virtual method call in, say, the C++ programming language, which is also implemented via indirect function calls. When the dictionary is statically known the compiler can, of course, remove the pattern match and directly call the implementation. Inlining may expose further optimisation opportunities.

2.6.2 Specialisation

In order to optimise functions that abstract over type classes we need *specialisation*. The following is the `elem` function translated to use explicit dictionary arguments:

```
elem :: DictEq a -> a -> [a] -> Bool
elem dict x l =
  case l of
    [] -> False
    (y:ys) -> (==) dict x y || elem x ys
```

We can create a specialised version of this function for arguments of type `Bool`. This involves duplicating all code and substituting `dictEqBool` for the `dict` argument. After further optimisation we get this more efficient version:

```
elem_Bool :: Bool -> [Bool] -> Bool
elem_Bool x l =
  case l of
    [] -> False
    (y:ys) -> dictEqBool_eq x y || elem_Bool x ys
```

Like inlining, specialisation requires code duplication. This may enable subsequent optimisations which may eventually shrink the amount of code, but a static compiler must carefully evaluate the trade-off between better performance and size of the compiled code. Increased code size can negatively affect compilation time or even performance due to worse (instruction) cache utilization.

Automatic specialisation is difficult in practice, because a polymorphic function and all the functions that it calls must be specialised. If multiple parts of a large program require the same specialisation of a function, then it could potentially be shared, but that would often be insufficient and it would work against inlining.

C++ templates use essentially this technique. Any polymorphism is eliminated by specialising each use site to the desired monomorphic type. Sharing of specialised code fragments can reduce the total amount of generated code, but the amount of generated code can still be excessive. A common work-around is to instantiate templates to use a generic pointer type (`void *`) which can then be shared across multiple usage sites. This effectively disables specialisation and corresponds to the use of polymorphic types in Haskell.

2.7 Dynamic Optimisation

A program optimiser that executes at program run time can perform specialisation based on the program's actual behaviour. Rather than having to be prepared for every possibility, the optimiser can specialise the code based on the commonly encountered argument patterns and provide a slower fallback for the uncommon case.

The next chapter discusses the techniques used in this work to implement such a dynamic optimiser.

Chapter 3

Trace-based Just-in-time Compilation

Just-in-time (JIT) compilation is commonplace in modern implementations of programming languages and virtual machines. In contrast to ahead-of-time compilation, where a program is first compiled and then run, just-in-time compilation occurs while (or immediately before) the program is running. It is commonly employed to improve the performance of interpreter-based systems, such as, for example, the Java Virtual Machine (JVM) (Lindholm and Yellin [1999]), JavaScript runtimes (V8; Gal et al. [2009]), or binary translators such as IA-32 (Baraz et al. [2003]).

Because compilation is delayed until runtime, JIT compilers may incorporate information collected at runtime into optimisations. For example, a static Java compiler may implement virtual function calls using indirect branches. This is costly since hardware branch prediction has difficulties predicting the target such branches which in turn causes pipeline stalls. A JIT compiler could instead compile each call site to a direct branch to the method of the most commonly encountered type, preceded by a check that the object type is indeed as expected. If this check fails, the execution falls back to the indirect branch technique. It is mainly for this reason that JIT compilation is very popular for dynamically typed languages such as JavaScript where a basic operator such as "+" may do many different things depending on the types of its arguments at runtime (ECMA

International [2011]). A JIT compiler can potentially compile multiple versions, specialised to the commonly occurring argument types.

One important decision in designing a JIT compiler is to ask what constitutes a compilation unit. The traditional compilation unit of a whole file or module is generally too large. Since compilation is interleaved with program execution, compilation now counts towards the program's execution time. Compiling large sections of the program at once can cause a substantial delay in the program and would be too disruptive. Compiled code also increases the memory usage of the program, which leads to another reason to minimize the total amount of compiled code. A better strategy, then, is to compile smaller units and only those that are executed most frequently.

Common choices of compilation units for dynamic optimisation systems are:

Dynamic Basic Block A dynamic basic block (Smith and Nair [2005], Chapter 2) is a sequence of dynamically executed instructions starting from the target of a branch and ending with the next branch. A dynamic basic block may include several static basic blocks, for example if one static basic block has a "fall-through" branch to one of its successors. For example, in Figure 3.1 (middle), basic blocks `A` and `B` get merged into one dynamic basic block because they are linked by a fall-through branch. Because `B` later becomes the target of a branch, another dynamic basic block is created which only contains `B`. This technique is often used in binary translators where the program's higher-level structure is not known.

Method/Function The natural compilation unit for implementations of most programming languages is a single function (or method). Since static compilers normally also compile a single function at a time, this means that the same techniques used for static compilation can be used in the JIT compiler (assuming that they can run quickly enough). It also makes it easy to interface compiled and interpreted code if both use the same calling convention. In Figure 3.1 (left), the compiler has a choice to compile each function separately (blocks `A-G`, and blocks `H-K`), or inline the called function (`H-K`) into the outer function and compile the complete control-flow graph at once.

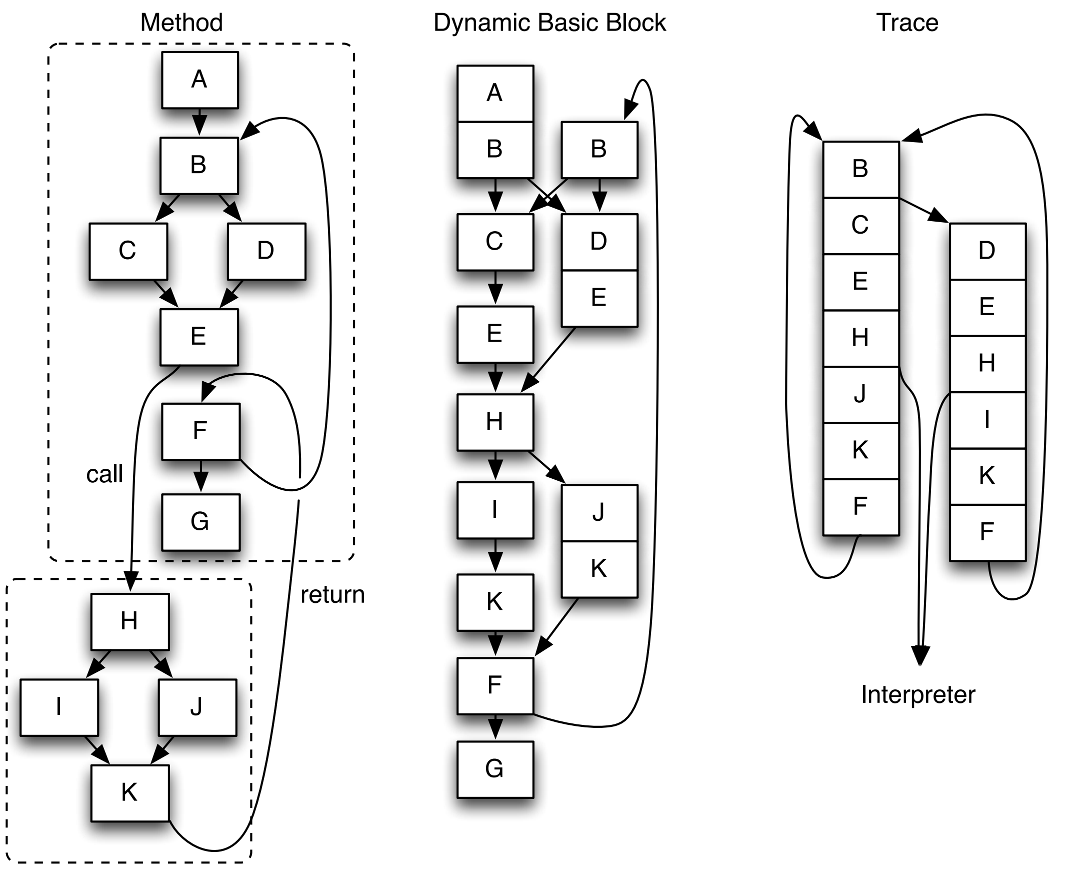


Figure 3.1: Compilation units: methods, dynamic basic blocks, and traces.

Trace A trace is a linear sequence of instructions with a single entry point and one or more exit points. A trace does not contain any inner control-flow join points—execution either continues on the trace or it exits the trace. As a special case, a trace may form a loop, i.e., the last instruction of the trace may jump back to the beginning of the same trace. Traces are not restricted to a single function, but are allowed (and often do) include parts of several functions. If there are multiple frequently executed control flow paths in a function, then these must be covered using multiple traces. As shown in Figure 3.1 (right), this can cause a fair amount of duplication, but this duplication may provide more opportunity for specialisation and removal of redundancies.

Region This is the most general compilation unit and allows basic blocks from several functions to be included in the compilation unit. All basic blocks must be related via control-flow edges, but control-flow within a region is unrestricted. In particular, a region may include several loops and several control-flow join points.

Because this thesis is concerned with implementing a language runtime rather than a binary translation system, we do not consider dynamic basic blocks since their scope for optimisation is too small.

3.1 Functions vs. Traces vs. Regions

We now discuss the trade-offs between the choices of compilation units.

Detection of hot code In order to determine which parts of a program are worth optimising, a JIT compiler needs to perform some form of profiling. The choice of profiling method determines the runtime overhead and the accuracy of the profiling data. Commonly chosen profiling methods either involve counters or statistical profiling using a timer interrupt.

A statistical profiler can be used if the compilation unit is a function (Arnold et al. [2000]). The profiler interrupts the program at a fixed time interval (e.g., whenever the thread scheduler interrupts the program) and inspects the call stack. Methods that occur frequently at or near the top of the stack are considered as hot and are compiled to machine code. The profiling data may also be used to inform inlining decisions.

Profiling using counters is applicable regardless of the choice of compilation unit. The profiler counts, e. g., the number of times a function is called, or the number of times that a particular basic block is called.

Scope for optimisation If the compilation unit is too small, there is very little room for the compiler to perform any optimisations (apart from removing the overhead of the interpreter). The compilation unit should therefore be of reasonable size and preferably span multiple functions to enable interprocedural optimisations.

A function can provide reasonable scope for optimisations if combined with inlining of smaller functions called from within the function. Since programs spend most of their time in loops, it is important that a compiler can optimise a whole loop at a time whenever possible. In imperative languages loops tend to be contained fully within a function (e.g., because they are implemented using `for` or `while` statements) and are easy to recognize. Because of the importance of loop optimisations, the compiler may choose to, e. g., inline functions more aggressively if they are called from within a loop. Even if the language implementation supports tail call optimisations, they may not be optimised to the same degree.

In functional languages, loops are implemented as (tail-)calls to recursive functions. If a loop is implemented using mutually recursive functions, detecting the loop becomes more difficult. For example, if a loop spans two mutually recursive functions f and g then the compiler has to inline g fully inside f (or vice versa) to optimise the whole loop.

In a trace or region compiler loop detection is part of the trace selection mechanism. The compilation unit therefore often is a loop to begin with. Optimisations on a trace or region are also naturally interprocedural because the selected traces often span multiple functions. The quality of the selected traces, however, is important. If a hot inner loop requires more than one trace, then the trace compiler may produce much less efficient code due to missed optimisation opportunities. Selection of high-quality traces is an active area of research (see Section 3.3).

Applicable optimisations In a method compiler, the whole function is visible to the optimiser. Methods called from within the compiled method are either inlined either fully or not at all. Having access to the whole method allows some optimisations which require proving that a certain condition cannot happen. For example, escape analysis (Choi et al. [1999]) tries to determine whether a local variable's value is stored in a place where it may be observed by other threads. A trace compiler may have to assume that a value escapes even if it does not, because the trace compiler cannot "see" the off-trace code (or chooses not to look at it in order to speed up compilation).

Some of this can be ameliorated by doing limited static analysis and embedding this information into the program. For example, a dead-code analysis may benefit from knowing the live variables at each branch point. That way, the trace compiler can determine whether a variable is live on an off-trace path. Even if conventional optimisations would be ineffective, a trace compiler can achieve the same effect via other optimisations. For example, while escape analysis cannot be applied effectively to traces, allocation sinking (Section 5.10, Ardö et al. [2012]) and forms of partial evaluation (Bolz et al. [2011]) can be used to achieve a similar effect.

Compilation overhead Since a function can contain more complex control flow, a function compiler is also more complex and thus more difficult to make fast. Optimisations in a method or region compiler work in a way very similar to static compilers. If there are control-flow join points, then analyses must be iterated until a fixed point is reached. Keeping the program in SSA form helps keep such iteration to a minimum, but it cannot be avoided in general, and establishing and maintaining SSA form during optimisations has some overhead. One of the bigger disadvantages of a method compiler is that it compiles the full function, which may include a large amount of code that is not actually executed very often. Inlined functions increase the optimisation problem even further (although some parts of the inlined function may turn out to be dead code) and thus inlining has to be rather conservative.

In contrast, a trace compiler can be extremely simple. Often a single pass of forward optimisations (e.g., constant propagation, algebraic simplification, removal of redundant operations) and a single pass of backward optimisations (e.g., dead code elimination) is enough. No iteration is necessary. Both forward and backwards passes can be implemented as a pipeline of simple optimisations. This makes a trace-based compiler both simpler and requiring less compilation overhead than a method-based compiler (assuming both generate the same amount of code) (Gal and Franz [2006]).

Higher compilation overhead may be less of a problem for long-running systems where it can be amortised over the total run time of an application. For shorter programs compilation time is more important, possibly at the expense of absolute performance. Note that a faster compiler can also more quickly adapt to changing runtime behaviour of the application.

Size of compiled code The size of the compiled code can influence the memory footprint of the application. On memory-constrained devices (e.g., mobile devices or shared servers) this can be an important factor. In a function-based compiler code size can be controlled by varying the degree to which aggressive inlining is performed.

Because a trace cannot contain any joining control flow nodes (except at the entry) any code occurring after such a merge point is duplicated (e.g., blocks `E`, `H`, `K`, and `F` in Figure 3.1). This is known as *tail duplication.* Tail duplication can enable additional optimisation by propagating information from before the diverging control flow paths. A function compiler will have to assume that code after the merge point could have been reached through either path. On the other hand, tail duplication can cause large amounts of duplicated code for certain extreme cases. Figure 3.2 shows how tail duplication can increase code size. In this example, the basic block E has been duplicated four times. A trace compiler has to protect against excessive duplication, but in small amounts it can be beneficial if it enables additional code specialisation.

Integration of compiled and interpreted code A method compiler only needs to be able to call and be called from interpreted functions. A more difficult problem in a method compiler is the technique by which a running function is replaced with an optimised version of the same function, called *on-stack replacement* (OSR) (Soman and Krintz [2006]).

A trace compiler needs to be able to switch back to the interpreter at any exit point. The integration with the interpreter is tighter, which means that the design decisions of the virtual machine can have a larger effect on the performance of the whole program. For example, a trace compiler cannot change the size of a function's activation frame if execution may fall back to the interpreter to execute some other part of the function.

A region-based compiler avoids the tail-duplication problems of a trace-based compiler and also avoids compiling unused code. However, it is also more complicated than either trace-based or method-based compilers since it must be able to deal both with control flow joins and also integrate well with the interpreter in case execution leaves the compiled region.

We did not consider implementing a region-based compiler because of the higher implementation cost. Instead we chose to implement a trace-based JIT compiler for Lambdachine. There were two reasons for this:

Implementation Effort A trace compiler is simpler overall to implement than a method compiler. Of course, the trade-offs change if we can reuse an existing implementation. There are a number of open source projects that can be used. A commonly used library for JIT compilers, particularly method-based compilers, is LLVM (Lattner and Adve [2004]).

LLVM is certainly very powerful and can generate high-quality code, but it is relatively slow as a JIT compiler. LLVM performs optimisations using a sequence of simple optimisation passes, some of which may be run multiple times. This makes it easier to implement new optimisation passes, but optimisation passses that combine multiple optimisations are typically faster (yet harder to maintain).

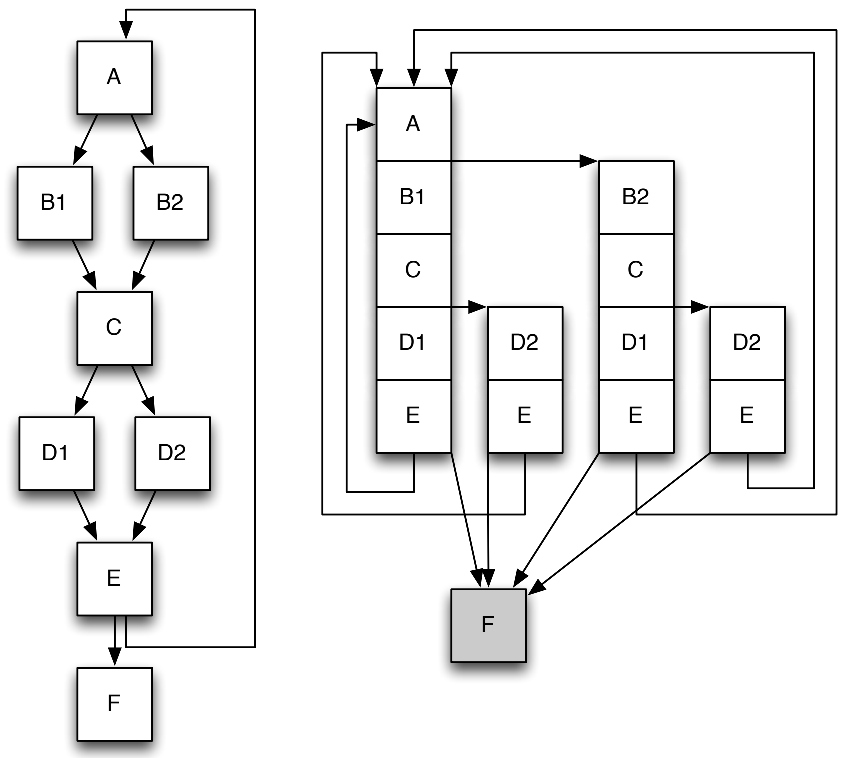


Figure 3.2: Excessive code duplication due to tail duplication. Because traces cannot contain merging control flow, blocks occurring after control flow merge points are duplicated. For certain control flow patterns that can lead to some basic blocks being duplicated many times.

Fortunately, there is a high-quality open source trace-based JIT compiler available, namely LuaJIT (Pall [2013]). We therefore chose to base our implementation on LuaJIT as we estimated the cost of using LuaJIT to be not much higher than LLVM. LuaJIT's code generator, in turn, takes advantage of the special structure of traces and has been heavily optimised with high compilation speed in mind.

Optimisation/Effort Trade-off The choice of compilation unit affects which optimisations can be performed efficiently or at all. For example, a trace-based JIT compiler automatically performs partial inlining which is hard to implement in a pure method compiler. Similarly, a trace-based JIT cannot on its own decide if a variable is live-out (but it can take advantage of liveness information provided by the bytecode compiler).

We felt that the optimisations that can be implemented easily and effectively in a trace-based JIT compiler are sufficient.

The alternative of porting GHC's (essentially method-based) optimisation passes to a lower-level intermediate representation did not promise to enable significantly more powerful optimisations and would be much more difficult to implement. Support for speculative optimisations only possible in a JIT compiler would have required additional effort.

Looking back, we still consider this analysis as valid. Since we did not implement a method-based compiler it is difficult to compare the implementation effort involved, but we have not discovered any issues that would be significantly easier to solve in a method-based compiler. The biggest issue specific to trace-based compiler is the design of trace selection strategy which is discussed further in Section 3.3. While this is not needed in a method-based compiler, we expect inlining decisions to be a related non-trivial design aspect of a method-based compiler.

We now explore in more detail the design decisions involved in the implementation of a trace-based JIT compiler. Throughout this chapter we remark which of the possible choices were made in Lambdachine.

3.2 Trace JIT Compiler Overview

A trace-based JIT compilation system typically has three modes:

Interpret. Execute bytecode using interpreter.

Record. Execute bytecode using interpreter and additionally record each executed instruction.

Native. Execute compiled machine code.

An optimised trace compiled to machine code is called a *fragment* and is stored in the *fragment cache*. The high-level behaviour, shown in Figure 3.3, consists of the following phases:

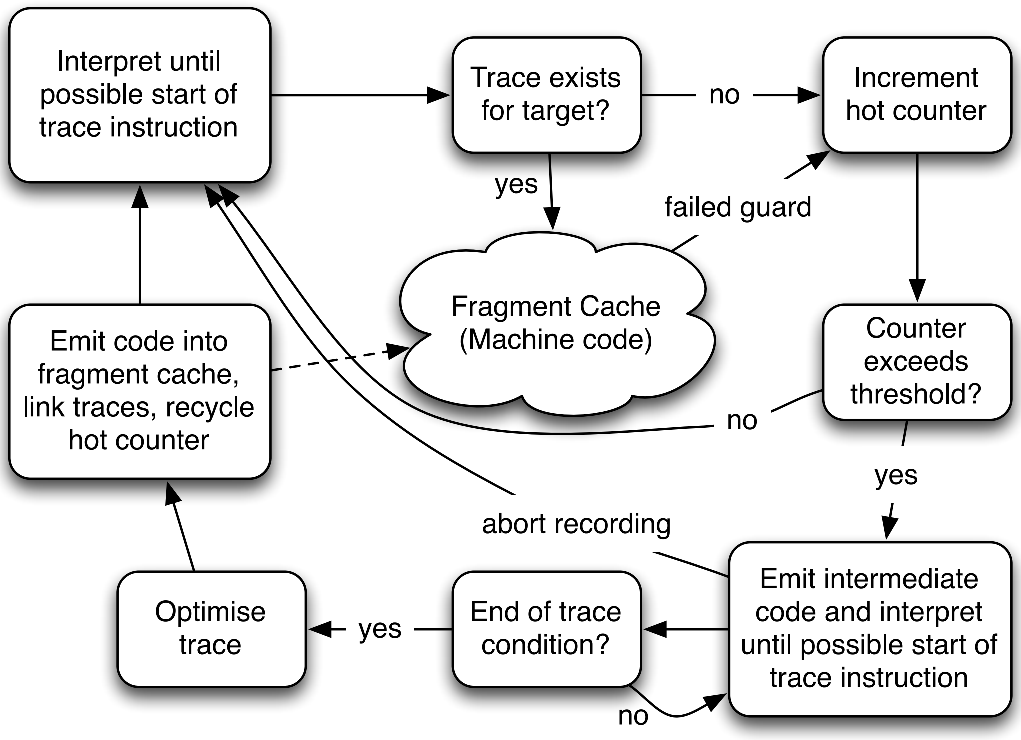

Interpret until possible start of trace instruction
Trace exists for target?
no
Increment hot counter
yes
failed guard
Fragment Cache (Machine code)
Emit code into fragment cache, link traces, recycle hot counter
Counter exceeds threshold?
no
yes
abort recording
Emit intermediate code and interpret until possible start of trace instruction
End of trace condition?
yes
Optimise trace
no

Figure 3.3: Overview of trace-based JIT compiler execution modes.

1. Execution starts in the interpreter (top left in Figure 3.3). Certain instructions may be potential *trace head*, that is, a trace may start at that instruction. If the interpreter encounters such an instruction it first checks the fragment cache to ascertain whether a trace for that trace head already exists. If so, execution is transfered to the machine code of the respective fragment until the machine code falls back to the interpreter.

   If no trace exists for a given trace head, the hotness counter associated with that trace head is incremented. If the hotness counter exceeds the hotness threshold, the interpreter is switched into recording mode (phase 2 below). Otherwise, execution continues normally in the interpreter. If execution switches from compiled code back to the interpreter, another hot counter is incremented and may trigger a switch to recording mode. Section 3.3 discusses different techniques for discovering and selecting traces.

2. When the interpreter is in recording mode it continues to interpret instruction, but also records the executed instructions. This is usually done by emitting instructions of an intermediate representation into a compilation buffer.

   Under some circumstances recording may fail. Since recording is speculative it could happen that the program takes an execution path that does not produce a suitable trace, e. g., an exception is thrown by the program, or garbage collection is triggered. In this case the recorded code is discarded and the interpreter switches back to normal execution.

   Recording normally finishes when a loop is detected. A trace may also be completed when the interpreter reaches the start of an existing traces. The decision about when to stop recording can influence the performance of the generated code as well as the total amount of code generated. Section 3.4.3 discusses these trade-offs.

3. If trace recording completes successfully, the recorded intermediate code is optimised and compiled to machine code. The machine code is placed in the fragment cache and execution continues in the interpreter. If the compiled trace starts at the exit of another trace, the *parent trace*, then

the two traces may be linked together so that if the parent trace exits, execution is transferred directly to the child trace. Section 3.6 discusses multiple techniques for trace linking.

Execution need not start in an interpreter. Some systems eschew an interpreter in favour of a fast baseline compiler. A baseline compiler makes sense if implementing a fast interpreter would be difficult. For example, implementing a fast interpreter for x86 instructions is difficult due to the cost of decoding x86 instructions. By translating a basic block at a time with minimal optimisations, the baseline compiler can achieve good performance at the cost of using additional memory.

However, if the design of the interpreter is under our control it is possible to write a very efficient interpreter, which is ultimately more flexible. In a direct-threaded interpreter (Ertl and Gregg [2003]) switching execution mode is as easy as replacing the pointer to the dispatch table. The dispatch table is an array of pointers to the implementation of each opcode. A baseline compiler usually needs to generate new code for each execution mode instead and thus has higher memory overhead.

Lambdachine uses an interpreter instead of a baseline compiler. This is for a number of reasons. An interpreter is easier to implement and test; it has lower memory overhead since it does not need to store any generated machine code. The bytecode is register based which reduces the dispatch overhead of the interpreter (Section 5.2.1, Section 7.6). Lambdachine's interpreter is implemented in C++. Section 7.10 discusses systems that use a baseline compiler instead of an interpreter.

3.3 Trace Selection

A key to good performance of a trace-based JIT compiler is the quality of the selected traces. If the selected traces are too short, the scope for optimisations is too small. Longer traces are preferable, but they will only pay off the recording and compilation time if they are executed frequently and if execution remains on the trace most of the time.

Another problem is that traces must be created speculatively. Traces must be created before the program actually spends a lot of time actually executing that particular code path. If the wrong traces are selected the overhead of invoking the JIT compiler will not pay off. For this reason, trace selection schemes focus on detecting hot loops as any long running program must execute some kind of loop.

A metric for estimating the quality of selected trace is how many traces are required to account for X% of the program's execution time. The *X% cover set* is defined as the smallest set of regions that comprise at least X% of program execution percentage (Bala et al. [1999]). A commonly used metric therefore is the size of the 90% cover set (Bala et al. [1999]; Hiniker et al. [2005]).

Finally, the mechanism to detect traces must not cause too much profiling overhead. Most implementations use simple counter-based profiling. Method-based JIT compilers sometimes use periodic stack sampling to detect which methods are called frequently, but that does not work for detecting traces.

Trace-based JIT compilers therefore use counter-based profiling. Certain program points are identified as potential *trace heads*, that is places where a trace may start, and each has an associated hotness counter. If a trace is recorded and compiled, the counter for its trace head may be recycled since execution will be diverted into the fragment cache from now on.

3.3.1 Selecting Trace Heads

We now discuss different mechanisms for identifying trace heads:

Next executed tail (NET) Every loop must contain at least one backwards branch. The next executed tail heuristic considers every target of a backwards branch a potential trace head (Bala et al. [1999, 2000]; Duesterwald and Bala [2000]). A backwards branch is any branch to a lower address than the current program counter. This implies that where traces may start depends on some arbitrary aspects, such as the static layout of the program code in memory. It also means that every function call (recursive or not) gives rise to at least one potential trace head. Either the function itself is considered a trace head, or the return point after the call site is a candidate.

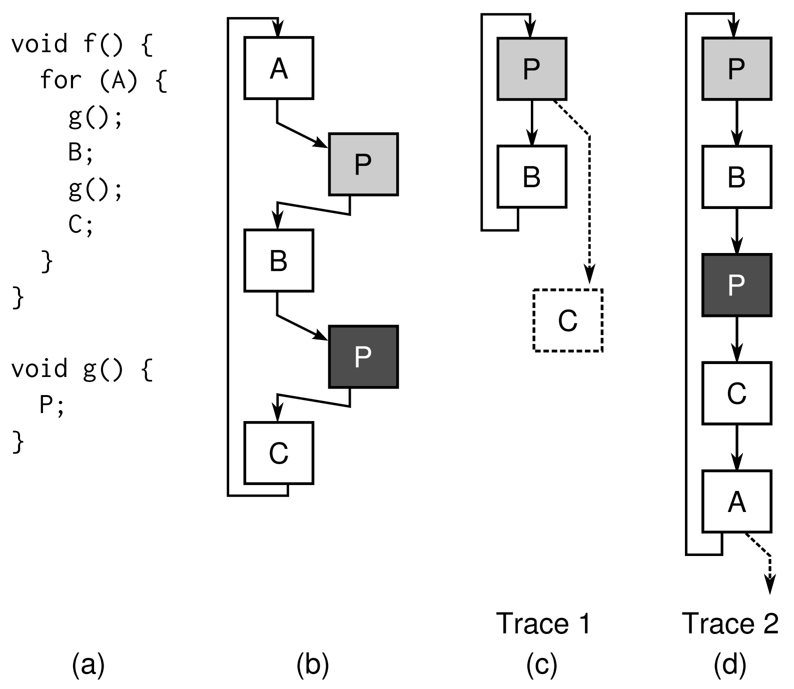
While inaccurate and somewhat random, NET is useful where we do not have higher-level knowledge of the program structure. It is therefore quite popular for binary translation systems (e.g., Bala et al. [2000]; Bruening et al. [2003]).

Natural loop first (NLF) If more knowledge about the higher-level structure of the program is available, it makes sense to have special instructions that mark the beginning of a loop. For example, a `for` or `while` loop can be translated to a special bytecode instruction (Pall [2009]). Recursive functions or gotos (if the language supports them) may also give rise to loops, so a strategy must be in place to detect those, for example, by using NLF for these types of branches.

The Maxpath VM (Bebenita et al. [2010b]) for Java does not use a special bytecode instruction to mark loop heads, but uses static analysis during bytecode verification to detect loops.

NLF has the advantage of being more predictable about where traces start and it reduces the number of counters required (since fewer program points are considered potential loop headers). Traces are more likely to start in natural places, especially in programs that rely mainly on loops and less on recursion.

Last Executed Iteration (LEI) Another way to detect loops is to keep track of the last n branch targets in a branch history buffer and only consider branches to a target in this buffer. Like NET only targets of backwards branches are considered potential trace head. LEI is thus a refinement of NET. While LEI has higher runtime overhead due to the need to maintain the buffer and hash table, it does need fewer counters since fewer targets are potential trace heads. LEI was proposed by Hiniker et al. [2005], who showed that on the SPECint2000 benchmarks it requires fewer traces to achieve 90% coverage and fewer region transitions than NET.



```
void f() {
  for (A) {
    g();
    B;
    g();
    C;
  }
}

void g() {
  P;
}
```

Figure 3.4: False loop filtering: (a) shows the imperative source program; (b) is the control flow graph corresponding to the source program; (c) shows the selected trace using NET or LEI without without false loop filtering (assuming P occurs at an address lower than A); (d) is the selected trace with false loop filtering.

> The branch history buffer may provide enough information to build a trace directly without interleaving interpretation and trace recording.

All trace selection schemes also consider side exits as potential trace heads. This ensures that all paths through a hot loop are compiled into traces and not just the one path that happened to be detected first.

Lambdachine uses NET, because it is simple and we do not have a special bytecode for loops—all loops are implemented via tail-calls. We do some false loop filtering (see below), but the resulting traces are not very good (Section 6.4).

3.3.2 False Loop Filtering

Both NET and LEI may detect *false loops* that can arise due to function calls. Consider the example code of Figure 3.4a. The loop inside function `f` contains two calls to the function `g`. If the code for `g` occurs at a lower address than the code for `f`, then each call is a backwards branch and both NET and LEI will consider the entry point of `g` as a potential trace head. Furthermore, since each iteration of the loop in `f` contains two calls to `g`, the hotness counter for `g` reaches the hotness threshold first. The trace recorder then starts recording the basic block `P`, followed by a return to, say, basic block `B`. Following `B` execution will again enter `g`, so the trace recorder has found a loop and recording concludes (Figure 3.4c).

Unfortunately, this trace will execute at most one iteration of its loop. The second iteration will always leave since the correct return point is the basic block `C`, not `B`. The problem is that the trace represents a *false loop*: the second basic block `P` occurred in a different context than the first, that is, the return address on the call stack was different.

False loop filtering (Hayashizaki et al. [2011]) prevents the trace recorder from stopping at such false loops by keeping track of the relevant calling context during trace recording. The calling context consists of the top part of the stack frame. Hayashizaki et al. [2011] also describe various ways how the relevant parts of the stack frame can be collected on the fly. They also ensure that proper (non-tail) recursive calls are correctly detected as true loops.

The trace selected with false loop filtering applied is shown in Figure 3.4d. It contains the same number of basic blocks as the original loop (Figure 3.4b), provides better optimisation potential, and only contains one trace exit.

Note that since the trace starts at `P`, other calls to function `g` may cause execution to enter the Trace 2, but will then quickly bail out because, again, the calling context was different. Avoiding this requires a mechanism to have multiple traces for the same program point and a means to distinguish them based on calling context. For example, the trace starting at `P` could be annotated with the return address, namely `B`. When the interpreter reaches the start of `B`, then instead of always entering the same trace, it would have to examine the current return address and pick a trace to be entered (or record a new one).

The root problem in this example is that the trace does not start at a natural loop. A potentially better strategy would abort recording the trace starting at `P` once the trace recorder detects that the edge from `C` to `A` is part of primitive loop (the `for` statement). The trace recorder could then continue recording, but using `A` as the trace head. We are, however, not aware of any trace-based JIT compiler that implements this strategy.

LuaJIT reduces the need for false loop filtering by giving preferred treatment to natural loops.[1] Even though `P` will get hot first and LuaJIT will create a trace containing just `P`, but LuaJIT is reluctant to create a trace that returns outside of the current stack frame. Soon after the second trace will be created starting at the `for` loop and inline both calls to `g`. LuaJIT does create traces that increase or decrease the stack size, but does with effectively higher hotness thresholds than natural loops. These heuristics rely on the source language being used a certain way. It is not necessarily applicable to trace-based compilers for a different source language.

3.4 Trace Recording

Once a trace head reaches the hotness threshold a popular strategy is to simply switch the VM into trace recording mode and continue executing. This strategy is speculative in that there is no guarantee that the recorded trace is actually the most frequently executed trace starting at that trace head. If, say, a loop has two traces A and B and the loop executes trace A 90% of the time and trace B for the remaining 10%, then with 90% probability this strategy will select trace A and later on trace B will become a side trace of trace A. If we are unlucky and record trace B first then we may lose some optimisation opportunities in the common path since trace A will now become a side trace of B.

[1]Conversation on the `luajit` mailing list: `http://www.freelists.org/post/luajit/Tracing-of-returns,1`

3.4.1 Multi-stage Trace Recording

A common strategy to protect against accidentally selecting a suboptimal trace first is to have two hotness thresholds. The first threshold triggers a lightweight form of recording to gather statistics about the trace. In this mode we could, for example, record just one bit for each conditional branch and a compressed address for indirect branches (Hiniker et al. [2005]). Another approach would be to collect all traces observed in this mode into a trace tree structure (see Section 3.6) annotated with the execution frequency of each branch.

The second hotness threshold then triggers actual compilation to machine code. The trace compiler can now use the information from the first phase to choose which trace to compile or which way to arrange multiple recorded traces. Of course, since the lightweight recording phase only observes a small number of traces there is still a chance that a suboptimal trace is selected.

The downside of such an approach is that a profiled execution mode adds overhead, which has to pay off in improved trace selection. Hiniker et al. [2005] did not determine the performance overhead of the additional time spent in recording mode. Bebenita et al. [2010b] recompile a complete trace tree (a collection of traces, see Section 3.6) each time a new trace is added to it. Collecting several traces together and compiling them together later may be beneficial in that setting.

Lambdachine does not use multi-stage recording. Any successfully recorded trace is compiled to machine code. Our low trace completion ratio (Section 6.4) suggests that the added complexity may be worth it. Any changes to the trace selection heuristics will likely shift the trade-offs, so it is difficult to consider both independently.

3.4.2 Hotness Thresholds

The choice of hotness threshold can be an important aspect of the performance of a tracing JIT compiler. If the hotness threshold is too low, the selected traces may not actually be executed very frequently and the compilation overhead will never pay off. If the hotness threshold is too high, then too much time will be spent in the slower execution mode.

In general, the best threshold will be highly dependent upon the executed program. A trace compiler has to guess the future behaviour of the program, based on a very short profiling period. Even with a profiling phase there is always a chance that recorded code is no longer needed shortly after the JIT compiler has compiled it.

One advantage of tracing JIT compilers is that due to the simple structure of traces, optimising the trace and emitting machine code can be implemented very efficiently. A low compilation overhead will repay itself more quickly, thus the compilation threshold can be relatively low. JIT compilers with slower compilers generally choose higher thresholds which will manifest itself in longer warm-up periods, i.e., the virtual machine will take longer to before the program is running at peak performance. For example, the Java Hotspot VM has two modes, client mode (Kotzmann et al. [2008]) and server mode (Dimpsey et al. [2000]). Server mode is optimised for best peak performance and chooses more expensive JIT compilation and high hotness thresholds but suffers from longer start-up times. This mode makes most sense for long-running applications such as servers. Client mode is meant for applications that prioritise quicker startup times over peak performance: for instance, applications with a graphical interface.

Commonly used hotness thresholds for tracing JIT compilers are quite low. Dynamo (Bala et al. [2000]) settled on a hotness threshold of 50 after evaluating a range of possible values between 15 and 100 (Bala et al. [1999]). LuaJIT uses multiple (user-configurable) thresholds depending on the kind of trace: root traces have a default hotness threshold of 56 and side traces have a hotness threshold of 10. Gal et al. [2009] chose a very low threshold of 2 for their JavaScript JIT compiler, because JavaScript programs typically run for a very short time.

Lambdachine sticks with standard of around 50, but most benchmarks worked well with any value between 10 and 100 (Section 6.5).

3.4.3 End of trace Condition

Recording continues until either it is aborted or an end-of-trace condition is encountered. A possible reason for aborting a trace is that the program throws an exception, which is usually a rare occurrence. This indicates that the recorded trace is unlikely to be the common case, so many trace compilers do not support traces that include throwing exceptions. Trace recording is also aborted if the trace becomes too long. This is to guard against cases where the trace selection heuristic has failed. For instance, execution has likely left the loop that initiated recording of the trace.

It is possible that trace recording will fail each time it attempts to record a certain trace (e.g., due to some failure of the trace selection heuristics, or due to an unsupported feature in the trace compiler). Since execution in recording mode is more expensive than normal execution, such repeated trace recording attempts can be detrimental to overall performance. To prevent such slow-downs trace compilers employ a mechanism that bounds the number of recording attempts per potential trace entry point (e.g., Gal et al. [2009]). If recording a trace from a particular program location fails too many times, that program location is black listed.

Trace recording may complete successfully if any of the following conditions is encountered:

Loop back to entry This is the ideal case. The recorded trace loops back to the point where recording was started. The trace now represents a proper loop and can be fully optimised, including important optimisations such as loop unrolling or loop invariant code motion.

Loop back to parent The recorded trace must be a side trace and execution has reached the entry point of the root trace. It is harder to optimise side traces well unless we are willing to modify the root trace (and possibly other side traces) as well.

Inner loop False loop filtering (Section 3.3.2) detects if a trace contains an inner loop and treats it specially. Assuming the loop is not a false loop, the trace compiler will cut off the trace *before* the inner loop. The cut-off trace will immediately fall back to the interpreter upon completion, but this will quickly trigger the creation of another trace. Hopefully this trace will now form a proper loop and can be optimised fully.

Start of existing trace During recording we have encountered the start of an existing trace. For their Dynamo system, Bala et al. [2000] decided to stop recording at that point at the last instruction of the compiled trace would jump directly to the existing trace. Wu et al. [2011] argue that this reduces optimisation potential, and instead suggest that tracing should continue in this situation. This leads to longer traces, but combined with false loop filtering, trace truncation and trace path profiling it leads to overall better performance with relatively small increase in code size.

Lambdachine stops at all of these conditions. If an inner loop is detected, we do not cut off the trace at the beginning of the inner loop, but at the end. This is purely to simplify the implementation; the compiler would need to maintain additional meta data to be able to revert the trace recorder state to an earlier point in the trace.

3.5 Trace Exits

When a guard fails and execution leaves the trace, execution must switch back to the interpreter (or to unoptimised machine code). This requires that the virtual machine is left in a suitable state for the interpreter to continue. This mainly concerns the contents of the stack, whose values will normally be held in registers throughout the trace. On exiting the trace the values in registers must now quickly be written to their respective stack locations.

Many implementations use *exit stubs*, short pieces of code consisting mainly of store instructions that perform the necessary writes. The guard on the trace is implemented as a conditional branch to the exit stub. The end of the exit stub then consists of a jump to a VM routine that transfers control back to the interpreter. Exit stubs are often located at the end of the trace, but they may be located anywhere in jump range of the trace.[1] Exit stubs are used by Dynamo (Bala et al. [1999]) and DynamoRIO (Bruening [2004]).

Exit stubs are an efficient way to restore the VM state, but they take up space and may cause fragmentation of the machine code area. If a side exit gets hot then the exit stub is no longer needed and its memory can be reclaimed.

[1]On some architectures conditional branches have a limited jump distance.

Exit stubs, however, have various sizes, so this will lead to some fragmentation. Bruening [2004] (Chapter 6) describes the complexities involved in managing exit stubs.

Bruening [2004] also showed that, in DynamoRIO, exit stubs could take up on average about 20% of the compiled machine code. Storing exit stubs in a separately managed area of the heap enabled a reduction of over more than half of the required memory space by allowing the memory of no longer needed exit stubs to be recycled.

Another approach is to simply save the contents of all registers on trace exit and then use meta-data stored with the trace to reconstruct the VM state. For example, this meta-data could simply be a mapping from stack location to the name of the register that holds its value (if any). This approach is used by LuaJIT (Pall [2009]). We can think of trace exits stubs as a compiled version (or partial evaluation) of such a meta-data interpreter. The interpreted method is slower but avoids the need to generate extra code.

Note that trace exits should be a relatively rare event since frequently taken side exits will be turned into side traces. The number of times an exit stub is executed thus depends directly on the hotness threshold for starting side traces. Compiled side exits are hence only worthwhile if this threshold is sufficiently large or the cost of restoring the VM state is comparatively high.

3.6 Trace Trees and Trace Linking

A single trace only covers one path through the control flow graph. If multiple paths through the same graph are hot then multiple traces are needed to cover all the hot paths. Transitioning from one trace to the other must be efficient, too.

Of particular importance is the case of attaching a side trace to another trace's hot exit (due to a guard that fails a lot). We will call the trace to which the side trace is attached the *parent trace*.

One option is to overwrite the last branch of the exit stub to jump directly to the start of the side trace. The downside of this approach is that all values held in registers in the parent trace are first written back to the stack only to be read back from the stack in the new trace. It would be better (i.e., require fewer instructions and less memory traffic) if the side trace could read values directly from the register locations of the parent trace.

One strategy is to use *trace trees* instead of single traces as the compilation unit (Bebenita et al. [2010b]; Gal and Franz [2006]; Gal et al. [2009]). A trace tree consists of a root trace and all its attached side traces. When a new side trace is attached the whole trace tree is recompiled. The advantage of this method is that optimisations can see the whole trace tree and switching from the root trace to a side trace has little overhead. In particular, because the register allocator can make sure that matching register assignments are used throughout each side trace. The downside of this approach is that the compilation overhead increases because the whole trace tree is recompiled. Staged recording can mitigate this overhead: instead of recompiling the trace tree each time a new side exit becomes hot, the trace recorder simply waits a bit longer to see if other side exits become hot, too, and then adds them all to the trace tree. Another option is to wait before the first trace is compiled and then compile the whole trace tree only once.

A simpler approach is *trace coalescing*. The trace compiler keeps track of the mapping between registers and stack slots, i.e., the same meta-data that is needed to restore the VM state on a trace exit. This mapping is then used in the compiler for the side trace. Instead of emitting a load from a stack slot, the trace compiler emits a read from the register that held the contents of that stack slot in the parent trace. This technique is used by LuaJIT (Pall [2013]). LuaJIT's register allocator works last-instruction-to-first. This means, the register chosen by the parent trace may not be available and a move instruction must be inserted. Still, move instructions are fairly cheap on modern out-of-order processors, and trace coalescing still avoids both unnecessary memory traffic and the complexities of trace trees.

Lambdachine uses trace coalescing, partly for simplicity, and partly because much of the trace compiler code is adapted from LuaJIT. Changing the code to support trace trees would have required non-trivial changes. Given that trace trees are not clearly better (due to repeated recompilation), we did not pursue this possibility.

3.7 Compiling Traces

Due to the lack of inner control flow in traces, it is very simple to perform optimisations. Trace compilers commonly structure most optimisations as a simple pipeline through which each instruction is passed. Commonly implemented optimisations are:

- Constant propagation
- Constant folding
- Strength reduction / algebraic simplifications
- Copy propagation
- Redundant guard removal
- Common sub-expression elimination
- Redundant load removal. If a load is preceded by a load from (or store to) the same location, and the loaded (stored) value is still in a register, then the load is replaced by a reference to that register.

Not all of these optimisations are applicable in all settings. For instance, Bala et al. [1999] classifies some of these optimisations as *aggressive* because they may change the expected behaviour of the source program under some circumstances (e.g., volatile memory reads or writes).

Bebenita et al. [2010a] also perform more elaborate optimisations on guards:

- Guard implication. Remove a later guard if it is implied by an earlier guard.
- Guard hoisting. Replace an earlier guard by a later guard if the later guard implies the earlier guard. The later guard can then be removed (due to guard implication). This causes the trace to be exited at an earlier exit than strictly necessary, but if we assume that exits are taken infrequently, then this is unlikely to be a problem.

Some optimisations cannot be performed using a simple pipeline, but are nevertheless very useful:

- Dead code elimination
- Loop invariant code motion
- Loop unrolling

Memory access optimisations like redundant load removal must insure that no other store can write to the same location in the time between the first load or store. This is done using some kind of alias analysis. In some source languages additional semantic information may be available to improve alias analysis, e.g., an object read may not alias with a read of an array element.

If we assume that execution is unlikely to leave the trace at a side exit, then performing more speculative optimisations can become beneficial. For example, *code sinking* consists of moving instructions off-trace and only perform them if the execution leaves the trace. An example of this kind of optimisation is allocation sinking (Section 5.10).

Lambdachine most of the above optimisations except for guard implication or guard hoisting. Loop unrolling was implemented in an earlier prototype, but not in the implementation described here. It was deprioritised because its usefulness is reduced due to a difficult issue in optimising lazy evaluation as described in Section 6.6.

3.8 Code Cache

The optimised code is placed into the code cache. For side traces, the exit point of the parent trace is patched to delegate to the new side trace; for root traces the relevant data structures are updated so that the interpreter will switch to the compiled code.

The fragment cache can be simply a memory pool and the trace compiler allocates chunks of memory from it and places the compiled code in that memory. The fragment cache may allow portions of memory to be freed again, or it may only allow flushing the whole cache at once. Being able to remove traces also means being able to unlink traces which requires maintaining the necessary meta data (Bruening [2004]). As discussed in Section 3.5, it can be worthwhile to manage exit stubs differently from the main trace data due their potentially shorter lifespan.

Bruening [2004] showed that the amount of fragment cache memory needed varies significantly based on the application. Standard benchmarks required between 30 KB and 700 KB; complete applications required multiple megabytes. Imposing a size limit on the code cache can seriously reduce performance if the limit is too low. Bruening [2004] also developed a heuristic that automatically resizes the fragment cache. The fragment cache would evict old fragments in FIFO order, but keep a history of recently evicted traces. If new traces occur frequently in this history then the fragment cache is likely too small and must be increased.

Lambdachine currently has an unbounded cache which did not cause issues, because our benchmarks are small. This issue needs to be revisited once the implementation matures.

3.9 Reacting to Workload Changes

A trace is compiled and optimised for the behaviour that was observed when the trace was recorded. If the behaviour of the program changes later on, it might be worthwhile to re-optimise the program for the new behaviour. If the program runs through several phases then the code generated for the previous phase may no longer be needed and can be discarded.

Dynamo (Bala et al. [2000]) used a simple heuristic to detect such phase changes: if a phase changes, then new parts of the program will become hot and will cause an increase of the rate at which new traces are be created. If a the trace creation rate would suddenly increase Dynamo would then flush the code cache, i.e., discard *all* compiled code. The reasoning is that much of that code will probably be no longer needed. Since the trace compiler is very efficient, any code that was discarded erroneously will quickly be re-compiled.

3.10 Tiered Compilation

Some just-in-time compilers like the Java HotSpot VM allow users to specify one of two compilation modes. The server mode trades off start-up performance for better peak performance. As the name suggests, it is intended for long-running applications such as web servers. The client mode in turn prioritises short compilation times over peak performance (Kotzmann et al. [2008]).

Lambdachine at this point does not support any optimisations where spending more time in the JIT compiler could lead to better overall performance. Compilation times are very short, thus start-up perfomance is not an issue, either. This could change if the compiler becomes more complex in the future.

3.11 Reusing GHC Bytecode

GHC contains both a static code generator as well as an interpreter. Could this existing interpreter be extended with a JIT compiler? The answer is: yes, but we chose not to.

The GHC interpreter is stack-based and not particularly optimised for best interpreter performance. It also does not support certain features such as unboxed tuples (which are used to implement the `IO` monad, for example). While it supports most other language features, it is meant to instead integrate well with compiled code. This introduces some complications. For example, the calling convention of interpreted code is different from compiled code, thus special adapter stack frames are needed to interface one with the other. To avoid these design constraints, and to be able to obtain a very high-perfomance interpreter, we decided to use a custom bytecode instruction set and interpreter.

3.12 Summary

This chapter introduced the basic working principles of a trace-based JIT compiler and discussed many of the interrelated design aspects of an implementation. Lambdachine is based on LuaJIT and thus adopts many of its design decisions:

- We use simple traces rather than trace trees. Trace linking is done using trace coalescing.
- We do not use exit stubs and instead use low hotness thresholds for side traces.
- To detect loops we use a variant of NET, together with false loop filtering and a simple form of trace truncation.
- The trace optimisation engine works similarly to LuaJIT's. It consists of a rule-based rewrite engine for constant folding, algebraic optimisations, constant propagation, redundant load removal, and redundant guard removal. Dead code elimination is performed as part of code generation.
- The size of the trace cache is currently unlimited.

Several implementation details had to be adapted to the idiosyncrasies of lazy evaluation. These and further details of our implementation are described in Chapter 5.

Chapter 4

An Example

To give a better big picture idea of how our trace compiler for Haskell works, we now follow the execution of one program from its Haskell source code to the compiled machine code for the x86-64 architecture. Our example program simply computes the sum of a sequence of numbers:

```
upto :: Int -> Int -> [Int]
upto lo hi =
  if lo > hi then Nil
             else Cons lo (upto (lo + 1) hi))

sumAux :: Num a => a -> [a] -> a
sumAux acc l =
  case l of
    Nil       -> acc
    Cons x xs -> let !acc’ = x + acc in sumAux acc’ xs

test = sumAux 0 (upto 1 100000)
```

This chapter follows the transformation of the program by the Lambdachine tool chain.

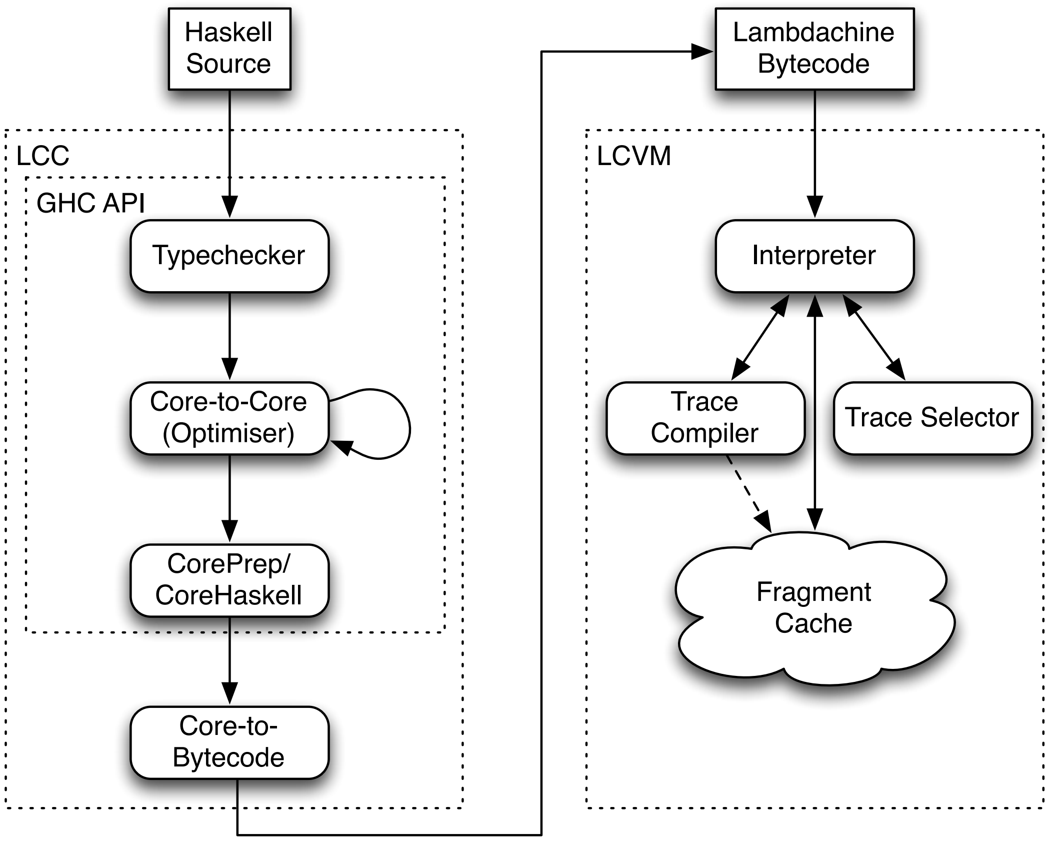


Figure 4.1: Overview of the Lambdachine infrastructure.

4.1 Core Haskell

The code is sent to GHC where it is type checked and translated into a subset of Haskell called Core. Most of the static optimisations are performed on programs in this representation. The program is then sent through the CorePrep pass which establishes certain invariants.

CorePrep corresponds to our CoreHaskell with explicit type information. The type information is only needed to derive information for the garbage collector, so we omit it in this chapter.

The invariants provided by CoreHaskell are in particular that all allocation is made explicit through `let`-expressions and all evaluation is made explicit through

```
upto = λ (from :: Int) (to :: Int) ->
  case from of from1 { I# x ->
  case to of to1 { I# y ->
  case x ># y of _ {
    False ->
      let ys =
           case +# x 1 of x' { _ ->
           let x'' = I# x' in
           upto x'' to1 }
      in
        from1 : ys
    True -> []  } } }

sumAux = λ (dNum :: Num a) (acc :: a) (l :: [a]) ->
   case l of _ {
     [] -> acc
     : z zs ->
       case (+) dNum z acc of acc' { _ ->
       sumAux dNum acc' zs } }

test4 = I# 0
test3 = I# 1
test2 = I# 100000
test1 = upto test3 test2
test = sumAux fNumInt test4 test1
```

Figure 4.2: The example program in CoreHaskell form.

case-expressions. For our example, the program after this phase is shown in Figure 4.2.

The top-level name `upto` is bound to a function with two arguments. The first `case` expression evaluates `from` to WHNF, binds the evaluated result to `from1` and then pattern matches on the result to extract the unboxed integer `x`. The `I#` constructor is the constructor for heap-allocated integers, i.e., values of type `Int`. The hash ("#") character is not special, it is treated like a regular character by GHC. By convention, names of primitive operations and types contain "#" in their names, and so do a few constructors for basic types. For example, `Int#` is the name for the type of unboxed machine integers. The type `Int` is defined simply as:

```
data Int = I# Int#
```

Variable `x` from Figure 4.2 therefore has type `Int#`. The second `case` expression then evaluates the second argument and extracts the unboxed integer `y` from it. Using `>#`, the greater-than operator for comparing unboxed integers, the code then makes a control-flow decision between the two cases. If the comparison returned `True`, `upto` simply returns a reference to the `[]` closure. Otherwise, `upto` first allocates a thunk `ys` on the heap. The thunk will contain the current values of the two free variables `x` and `to1` of the expression on the right hand side. Finally, another heap object is allocated with constructor "`:`" and the two fields `from1` and `ys`.

The second function, `sumAux`, used the `Num` type class, so it receives an extra argument which is the type class dictionary (`dNum`) for `Num`. This dictionary is then used to look up the implementation of addition using `(+)`. Note that `(+)` only requires one argument, but it will return a function that expects two arguments. GHC Core does not distinguish between such a function and a function that expects three arguments and then returns a value. The `case` expression ensures that the result of the addition is evaluated to normal form and then bound to the variable `acc'`. Finally, `sumAux` calls itself recursively.

The three closures `test2`, `test3`, and `test4` are simply statically allocated `Ints`. The two closures `test` and `test1` are *constant applicative forms* (CAFs), that is, they are top-level thunks. CAFs are allocated on the heap when the program is first loaded into memory. However, they are not immutable since, like any other thunk, a CAF must be updated with its value after it has been evaluated.

4.2 Bytecode

The program in CoreHaskell form is then translated to Lambdachine's own bytecode format. The translation is relatively straight-forward. A `case` expression translates to an `EVAL` instruction potentially followed by a `CASE` and `LOADF` ("load field") instructions to extract fields out of a heap-allocated object. Constants are stored in a literal table and referred to by an index by the `LOADK` instruction ("load constant"). A `let` expression is translated into a corresponding allocation instruction. The `ALLOC1` instruction allocates objects with exactly one field, and `ALLOC` allocates objects with 2 or more fields. Both instructions use a different encoding, hence the difference in the opcode name.

Each thunk is translated into its own info table. In our example, the thunk for the expression stored in variable `ys` is called `cl_upto_ys`. The `LOADFV` ("load free variable") instruction is used to load a value from the currently evaluated object into a register.

There are two call instructions, `CALL` and `CALLT`. The `CALL` instruction is evaluated in a new stack frame and the result of the call can be loaded into a register via `MOV_RES`. The `CALLT` instruction denotes a tail call which can be implemented as a jump. The arguments to `CALLT` must be in registers `r0` through `r`n. This invariant is enforced by the register allocator by inserting any necessary `MOVE` instructions. Both instructions are used by the bytecode for `sumAux` as shown in Figure 4.4

Every allocation instruction, `EVAL`, and every call instruction is also annotated with a list of live pointer registers. These are registers that contain pointers into the heap and whose value may be used later during execution. This information will be used by the garbage collector to detect which heap objects may still be needed by the program — any other objects can be discarded.

Since `sumAux` uses the `Num` class to perform addition, Figure 4.5 shows the relevant bytecode from its implementation. Both are straightforward; `(+)` simply loads the second field of its argument (the `Num` dictionary) and returns it; `plusInt` evaluates both its arguments (of type `Int`), extracts the unboxed integers, adds them, and re-boxes the result of the addition and returns it.

The Lambdachine bytecode compiler, `lcc`, stores this bytecode in a file on disk which is later loaded into memory by the Lambdachine runtime, `lcvm`.

4.3 Trace Compiler Intermediate Representation

After the program's bytecode is loaded into the memory, the runtime starts evaluating the entry closure. In this example we would specify `test` as the entry closure. Certain branch instructions such as `CALLT` or `EVAL` cause a hotness counter

```
upto:
  Function Payload=[]
  literals:
    0: info 0x10024d790 (cl_upto_ys)
    1: info 0x100208d10 (:)
    2: clos 0x1002006a0 ([])
  code (arity=2/[**])  function takes two arguments, both pointers
     0: FUNC    r4             the function uses 4 registers
     1: EVAL    r0             evaluate first argument (from)
     3: MOV_RES r0, 0          store result of evaluation in r0
     4: EVAL    r1             evaluate second argument (from)
     6: MOV_RES r1, 0
     7: LOADF   r2, r0, 1      unpack integer x from from
     8: LOADF   r3, r1, 1      unpack integer y from to
     9: ISLE    r2, r3 ->13    if x < y jump to 13
    11: LOADK   r0, 2          load pointer to empty list closure
    12: RET1    r0             return that as the result
    13: LOADK   r3, 0          load info table for thunk cl_upto_ys
    14: ALLOC   r1, r3, r2, r1   allocate the thunk
    17: LOADK   r2, 1          load info table for cons (:)
    18: ALLOC   r0, r2, r0, r1   allocate cons cell
    21: RET1    r0            return it

cl_upto_ys:
  Thunk Payload=[-*]   the thunk has two free variables, the second is a pointer
  literals:
    0: 1 (i)
    1: info 0x100208d50 (I#)
    2: clos 0x100203248 (upto)
  code (arity=0/[])
     0: FUNC    r3
     1: LOADFV  r0, 1            load the first free variable (x)
     2: LOADK   r1, 0            load the literal 1 into r1
     3: ADD     r0, r0, r1       integer addition
     4: LOADK   r1, 1            load info table for boxed integers (I#)
     5: ALLOC1  r0, r1, r0       allocate the boxed integer
     7: LOADFV  r1, 2            load the second free variable (to1)
     8: LOADK   r2, 2            load the a pointer closure for upto
     9: CALLT r2(r0*,r1*)        call upto
```

Figure 4.3: Compiled bytecode for `upto` and its local thunk `ys`

```
sumAux‘info:
  Function Payload=[]
  literals:
    0: clos 0x100202ec0 (+)
    1: clos 0x100203250 (sumAux)
  code (arity=3/[***])
     0: FUNC    r5
     1: EVAL    r2      {r0*,r1*}
     3: MOV_RES r2, 0
     4: CASE    r2 [tags 1..2]
           1: ->17
           2: -> 6
     6: LOADF   r3, r2, 1
     7: LOADK   r4, 0 ; clos 0x100202ec0 (+)
     8: CALL    r4(r0*)
     9: MOV_RES r4, 0
    10: CALL r4(r3*,r1*)
    12: MOV_RES r1, 0
    13: LOADF   r2, r2, 2
    14: LOADK   r3, 1 ; clos 0x100203250 (sumAux)
    15: CALLT r3(r0*,r1*,r2*)
    17: MOV     r0, r1
    18: EVAL    r0      {}
    20: MOV_RES r0, 0
    21: RET1    r0
```

Figure 4.4: Bytecode for the sumAux function.

associated with the target address to be decremented. If a hotness counter reaches zero, the interpreter switches to recording mode. The interpreter continues to execute the program, but before each instruction is executed an equivalent sequence of instructions in the JIT compilers internal intermediate representation (IR) is appended to a buffer, the IR buffer. In this section we will use a C-like syntax to denote IR instructions. An IR instruction usually corresponds to a single assembly instruction on a RISC machine.

The sequence of IR instructions follows very closely the implementation of the interpreter. For example, the instruction `ADD r1, r2, r3` is translated into the following IR instructions:

```
t1 = base[2]
```

```
GHC.Num.+‘info:
   Function Payload=[]
   code (arity=1/[*])
      0: FUNC    r1
      1: EVAL    r0      {}
      3: MOV_RES r0, 0
      4: LOADF   r0, r0, 2
      5: EVAL    r0      {}
      7: MOV_RES r0, 0
      8: RET1    r0

GHC.Base.plusInt‘info:
  Function Payload=[]
  literals:
    0: info 0x100208d50 (I#)
  code (arity=2/[**], frame=2)
     0: FUNC    r2
     1: EVAL    r0      {r1*}
     3: MOV_RES r0, 0
     4: EVAL    r1      {r0*}
     6: MOV_RES r1, 0
     7: LOADF   r0, r0, 1
     8: LOADF   r1, r1, 1
     9: ADD     r0, r0, r1
    10: LOADK   r1, 0 ; info 0x100208d50 (GHC.Types.I#‘con_info)
    11: ALLOC1  r0, r1, r0      {r0,r1}
    13: RET1    r0
```

Figure 4.5: Compiled bytecode for the `(+)` selector and `plusInt`.

```
t2 = base[3]
t3 = t1 + t4
base[4] = t3
```

The `base` variable points to the first item in the current stack frame. Just like in the interpreter, both operands must first be loaded from the stack frame into a register, then they are added together and the result is written back into the stack frame. Of course, if the next instruction reads from the same location, we can avoid both writing to and reading from memory by just keeping the result in a register. This kind of optimisation is performed on the fly, before any instruction is appended to the IR buffer. For instance, the bytecode sequence

```
ADD r0, r0, r1
SUB r1, r0, r2
MUL r0, r1, r0
```

is translated into the following sequence of IR instructions

```
t1 <- base[0]  ; ADD r0, r0, r1
t2 <- base[1]
t3 <- t1 + t2
t4 <- base[2]  ; SUB r1, r0, r2
t5 <- t3 - t4
t6 <- t3 * t5  ; MUL r0, r1, r0
base[0] = t6
```

If the value of register r1 is no longer needed later in the code, then we need not write its value back onto the stack and omit the write altogether.

For control-flow instructions we emit guards. Consider the execution of the instruction EVAL r0 where r0 pointed to a heap-allocated integer. We would simply emit a guard which verifies that this is still the case.

```
  ⟨assume t holds contents of r0⟩
if (info(t) ≠ &I#_info_table) goto exit1;
```

The expression info(t) extracts the pointer to the info table from the object's header. Normally, this is simply the first word of the object in the heap. The guard only compares the address of the info table against the expected address. The info table itself need not be read.

If r0 was a thunk when the trace was recorded, we will emit a guard for the info table followed by the IR instructions for evaluating the thunk.

```
  ⟨assume t holds contents of r0⟩
if (info(t) ≠ thunk1_info_table) goto exit1;
  ⟨code for evaluating thunk1⟩
```

```
entry:  ; sumAux
  ; FUNC r5
  ; EVAL r2
Object *t1 = base[2]
if (info(t1) ≠ &cl_upto_ys)
  goto exit1;
  ;   --- cl_upto_ys ---
  ;     Node = t1
  ;   FUNC r3
  ;   LOADFV r0, 1
int t2 = t1[1]
  ;   LOADK r1, 0
  ;   ADD r0, r0, r1
int t3 = t2 + 1
  ;   LOADK r1, 1
  ;   ALLOC1 r0, r1, r0
Object *t4 = new I#(t3)
  ;   LOADFV r1, 2
Object *t5 = t1[2]
  ;   LOADK r2, 2
  ;   CALLT r2(r0, r1)
  ;    no guard needed since
  ;    r2 contains a constant
  ;   --- upto ---
  ;   FUNC r4
  ;   EVAL r0
if (info(t4) ≠ I#) goto exit2
  ;   MOV_RES r0
  ;   EVAL r1
if (info(t5) ≠ I#) goto exit3
  ;   MOV_RES r1
  ;   LOADF r2, r0, 1
int t6 = t4[1]
  ;   LOADF r3, r1, 1
int t7 = t5[1]
  ;   ISLE  r2, r3 ->13
if (t6 > t7) goto exit4
  ;   LOADK r3, 0
  ;   ALLOC r1, r3, r2, r1
Object *t8 =
  new cl_upto_ys(t6, t5)
  ;   LOADK r2, 1
  ;   ALLOC r0, r2, r0, r1
Object *t9 = new Cons(t4, t8)
  ;   RET1
  ;  UPDATE
update (t1, t9)
  ; -- return to sumAux ---
  ; MOV_RES r2, 0
  ; CASE r2 [tags 1..2]
  ;    1: ->17
  ;    2: -> 6
if (info(t9) ≠ &Cons) goto exit5
  ; LOADF r3, r2, 1
Object *t10 = t9[1]
  ; LOADK r4, 0
  ; CALL r4(r0)
Object *t11 = base[0]
  ;   --- entering (+) ---
  ;   FUNC r1
  ;   EVAL r0
if (info(t11) ≠ &NumDict) goto exit6
  ;   LOADF r1, 2
t12 = t11[2]
  ;   EVAL r0
if (info(t12) ≠ &plusInt) goto exit7
  ;   MOV_RES r0
  ;   RET1 r0
  ; -- return to sumAux ---
  ; MOV_RES r4, 0
  ; CALL r4(r3, r1)
t13 = base[1]
if (info(t12) ≠ &plusInt) goto exit8
  ;   --- plusInt ---
  ;   FUNC r2
  ;   EVAL r0
if (info(t10) ≠ &I#) goto exit9
  ;   MOV_RES r0, 0
  ;   EVAL r1
if (info(t13)) ≠ &I#) goto exit10
  ;   MOV_RES r1, 0
  ;   LOADF r0, r0, 1
int t14 = t10[1]
  ;   LOADF r1, r1, 1
int t15 = t13[1]
  ;   ADD r0, r0, r1
int t16 = t14 + t15
  ;   LOADK r1, 0
  ;   ALLOC1 r0, r1, r0
Object *t17 = new I#(t16)
  ;   RET1  r0
  ; -- return to sumAux ---
  ; MOV_RES r1, 0
  ; LOADF r2, r2, 2
Object *t18 = t9[2]
  ; LOADK r3, 1
  ; CALLT r3(r0,r1,r2)
base[1] = t17
base[2] = t18
goto entry;
```

Figure 4.6: Trace IR for inner loop starting at `sumAux`.

```
entry:
  Object *t1 = base[2]
  if (info(t1) ≠ &cl_upto_ys)
    goto exit1;
  int t2 = t1[1]
  int t3 = t2 + 1
  Object *t4 = new I#(t3)
  Object *t5 = t1[2]
  ~~if (info(t4) ≠ I#) goto exit2~~
  if (info(t5) ≠ I#) goto exit3
  int t6 = t3 ~~t4[1]~~
  int t7 = t5[1]
  if (t6 > t7) goto exit4
  Object *t8 = new cl_upto_ys(t6, t5)
  Object *t9 = new Cons(t4, t8)
  update (t1, t9)
  ~~if (info(t9) ≠ &Cons) goto exit5~~
  Object *t10 = t4 ~~t9[1]~~
  Object *t11 = base[0]
  if (info(t11) ≠ &NumDict) goto exit6
  t12 = t11[2]
  if (info(t12) ≠ &plusInt) goto exit7
  t13 = base[1]
  ~~if (info(t12) ≠ &plusInt) goto exit8~~
  ~~if (info(t10) ≠ &I#) goto exit9~~
  if (info(t13)) ≠ &I#) goto exit10
  int t14 = t3 ~~t10[1]~~
  int t15 = t13[1]
  int t16 = t14 + t15
  Object *t17 = new I#(t16)
  Object *t18 = t8 ~~t9[2]~~
  base[1] = t17
  base[2] = t18
  goto entry;
```

Figure 4.7: Optimised Trace IR

Figure 4.6 shows the executed bytecode and the recorded trace IR for a single iteration of the loop induced by `sumAux`. There are a number of redundancies.

- Object `t9` was allocated on the trace (Line 44), hence the guard at Line 53 will always succeed and can be omitted. The same optimisation can be applied for `t4` (Line 17) and the guard at Line 27 and at a few other places.
- Loading a field from an object that has been allocated on the trace can be replaced by a reference to the value that was used to initialise the field. For example, `t10` (Line 55) can be replaced by `t4` since its value was stored at field 1 when `t9` was allocated (Line 44). This optimisation is always valid if the object is immutable. For mutable objects we would have to check whether there could have been any intervening write to the same location.
- The guard at Line 73 is the same as Line 66 and can be omitted.

4.4 Machine Code

Figure 4.8 shows how the IR is translated to x86-64 machine code. Three registers have predetermined purposes and are not used by the register allocator.

```
entry:
    ;   Object *t1 = base[2]
  mov rcx, rbp[16]
    ; if (info(t1) ≠ &cl_upto_ys)
    ;  goto exit1;
  mov r13, cl_upto_ys
  cmp [rcx], r13
  jne _exit1
    ;  int t2 = t1[1]
  mov rbx, [rcx + 8]
    ; int t3 = t2 + 1
  mov r8, rbx
  add r8, 1
    ; reserve 80 bytes
  lea r12, [r12 + 80]
    ; check for heap overflow
  cmp r12, [rsp]
  ja heapoverflow
    ; Object *t4 = new I#(t3)
  mov rdi, I#
  mov [r12 - 80], rdi
  mov [r12 - 72], r8
  lea r10, [r12 - 80]  ; r10 = t4
    ; Object *t5 = t1[2]
  mov rbx, [rcx + 16]
    ; if (info(t5) ≠ I#) goto exit3
  cmp [rbx], rdi  ; rdi = I#
  jne _exit3
    ; int t6 = t3
    ; int t7 = t5[1]
  mov r14, [rbx + 8]
    ; if (t6 > t7) goto exit4
  cmp r8, r14
  jg _exit4
    ; t8 = new cl_upto_ys(t6, t5)
  mov [r12 - 64], r13  ; cl_upto_ys
  mov [r12 - 56], r8
  mov [r12 - 48], rbx
  lea rbx, [r12 - 64]  ; rbx = t8
    ; Object *t9 = new Cons(t4, t8)
  mov r9, Cons
  mov [r12 - 40], r9
  mov [r12 - 32], r10
  mov [r12 - 24], rbx
  lea r9, [r12 - 40] ; r9 = t9
    ; update (t1, t9)
  mov rdx, IND
  mov [rcx], rdx
  mov [rcx + 8], r9
    ; Object *t10 = t4
    ; Object *t11 = base[0]
  mov rcx, [rbp]
    ; if (info(t11) ≠ &NumDict) ...
  mov rsi, NumDict
  cmp [rcx], rsi
  jne _exit6
    ; t12 = t11[2]
  mov rsi, [rcx + 16]
    ; if (info(t12) ≠ &plusInt) ...
  mov r11, plusInt
  cmp [rsi], r11
  jne _exit7
    ; t13 = base[1]
  mov rdx, [rbp + 8]
    ; if (info(t13)) ≠ &I#) ...
  cmp [rdx], rdi
  jne _exit10
    ; int t14 = t3
    ; int t15 = t13[1]
  mov rdx, [rdx + 8]
    ; int t16 = t14 + t15
  add rdx, r8
    ; Object *t17 = new I#(t16)
  mov [r12 - 16], rdi
  mov [r12 - 8], rdx
  lea rdx, [r12 - 16]
    ;  Object *t18 = t8
    ;  base[1] = t17
  mov [rbp + 8] = rdx
    ;  base[2] = t18
  mov [rbp + 16] = rbx
    ; goto entry;
  jmp entry
```

Figure 4.8: Trace IR compiled to machine code

- Register `rbp` is the `base` pointer. The virtual machine stack is kept separate from the C stack.

- Register `r12` is the heap pointer and points to the first byte available to the memory allocator. The heap pointer must always be less than the heap limit. Allocating memory from the heap is done simply by incrementing the heap pointer by the number of bytes needed. If that would move the heap pointer past the heap limit, then a heap overflow has occurred and execution leaves the trace. The heap overflow handler may then trigger garbage collector or select a new allocation region.

- Register `rsp` points to the top of the C stack at which some of the virtual machine's data is stored. For example, the address `[rsp]` contains the value of the current heap limit.

All other register can be freely used for storing intermediate results of the compiled trace code.

Most IR instructions map to one or two assembly instructions. Large literals, i.e., those that require more than 32 bits to encode cannot be used as an immediate in x86-64, so they must first be loaded into a register. The register allocator tries to keep constants in a register if enough registers are available.

Allocation is split into two parts: (1) a *heap check* reserving the required number of bytes in the heap, and (2) initialising that memory. The trace optimiser combines multiple heap checks into a single heap check to reserve the memory for all allocations on the trace. If execution leaves the trace at a side exit, some of that reserved memory may have to be given back to the memory allocator. The first allocation on a trace will trigger a heap check and is followed by the code to initialise the first-allocated object. All remaining allocations simply need to initialise the memory that has previously been reserved by the combined heap check.

Each object initialisation sequence in Figure 4.8 is followed by a `lea` instructions which puts a pointer to the beginning of the object into the target register. The `lea` instruction stands for "load effective address" and does not actually perform any memory access. An instruction `lea dest, [ptr + offset]` simply stores the value of the addition `ptr + offset` in register `dest`.

Chapter 5

Implementation

This chapter provides a detailed account of the implementation details of Lambdachine, our trace-based JIT compiler for Haskell. Lambdachine currently only targets the 64 bit x86 architecture (a. k. a. `x86-64` or `amd64`).[1] Its source code can be found at `https://github.com/nominolo/lambdachine`.

The organisation of this chapter largely follows the path of a program through the optimisation pipeline, which is shown in Figure 5.1.

The front-end, `lcc`, compiles GHC's core language into the bytecode format. This translation is rather simple and follows from the bytecode semantics. It is described in Section 5.3.

The bytecode format and related architectural decisions such as the layout of stack frames and heap objects is described in Section 5.2

The trace recorder is responsible for translating the semantics of a recorded instruction into the trace intermediate representation, or just trace IR, for short. The generated IR instructions are not directly written into a buffer, but are rewritten on-the-fly. The trace IR instruction set is described in Section 5.5. Section 5.11 describes how bytecode instructions are mapped to trace IR instructions. Section 5.8 explains the forward optimisation pipeline which optimises IR instructions on-the-fly before emitting them to the trace IR buffer.

Some optimisations propagate information from the end of the trace to the beginning. Such backward optimisations are done after recording completed suc-

[1]There are no fundamental limitations to support other architectures; it is simply a matter of implementation effort.

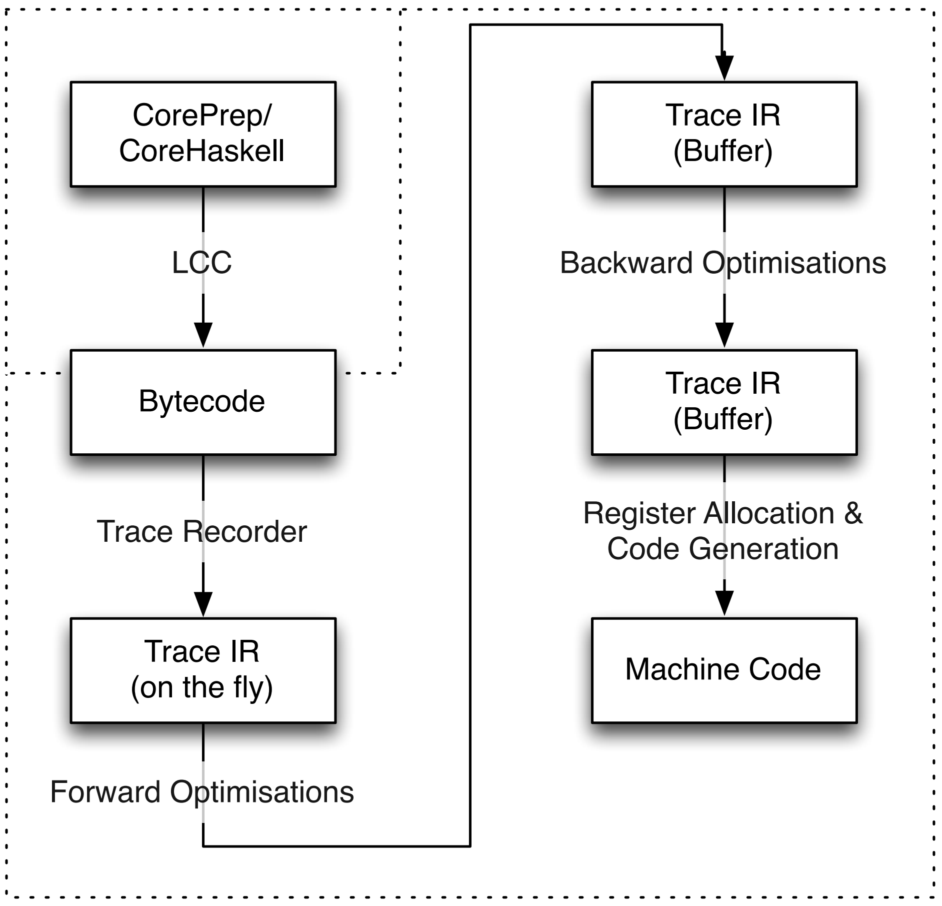


Figure 5.1: Stages and intermediate formats used in Lambdachine. We start after GHC has parsed, type checked, (optionally) optimised the Haskell program and finally translated it into the CorePrep/CoreHaskell form. From there, `lcc` translates it into Lambdachine's bytecode format. The bytecode is then loaded by the Lambdachine runtime system (`lcvm`) and interpreted. Eventually, some of the bytecode instructions are detected as hot and the trace recorder converts them into the trace intermediate representation (IR). The generated IR instructions are optimised on the fly and either eliminated or placed into the IR buffer. After recording has completed, additional optimisations may be performed on the IR buffer. Finally, the IR instructions are turned into machine code. Register allocation and dead code elimination are integrated with the machine code generation pass.

cessfully. Lambdachine currently does not use this stage. The only currently used backwards optimisation is dead code elimination, but that is integrated into code generation. The allocation sinking optimisation (Section 5.10) if combined with loop optimisation would require a backwards optimisation pass.

After all trace IR optimisations have been performed, the trace IR instructions are translated into machine code. The code generation pass operates backwards, that is, it generates the last instruction of the trace first and generates the first machine code instruction last. This is done to avoid a separate pass for determining the live ranges, needed by the register allocator. By operating backwards, register allocation can be performed immediately before the machine code instruction is written. The information used by the register allocator includes live variable information which can be used to simultaneously perform dead code elimination during register allocation.

Section 5.13 explains how switches between interpreter and compiled machine code are implemented. Finally, Section 5.14 describes how multiple traces are linked together efficiently.

5.1 Relationship between Lambdachine and LuaJIT

Lambdachine's runtime system, which includes the JIT compiler, is implemented in C++, but many architectural decisions and a certain amount of code were derived from the open source LuaJIT[1] implementation which is written in C (Pall [2009, 2013]). This section explains the relation between these two code bases.

LuaJIT is a trace-based JIT compiler for the programming language Lua (Ierusalimschy et al. [2006]). Lua is a dynamically typed language designed to be easily embeddable in other programs, for instance, as a scripting language in computer games. Lua is an imperative language, but it does support first-class functions. Its main data structure is a hash map (called "table").

[1]LuaJIT version 1 was actually a method-based compiler, and only version 2 is trace-based. In this thesis, we simply write "LuaJIT" to refer to version 2 unless otherwise stated. LuaJIT is available at `http://luajit.org/`

Even though the semantics of Lua and Haskell are very different, there are a number of components of a JIT compiler that are largely language independent. For example, aspects such as register allocation and code generation are almost independent of the source language. This also applies to the design of data structures used by the JIT compiler itself. Many JIT compiler optimisations can also be implemented in a very similar manner. In particular, the following design decisions were taken from LuaJIT:

- The in-memory layout of the bytecode instructions.
- The in-memory layout of the IR instructions as well as the internal organisation IR buffer and references to other IR instructions.
- There is some overlap in the IR instruction set semantics.
- The use of an abstract stack and snapshots.
- The code generator and register allocator are almost identical. We also use trace coalescing (see Section 3.6), which can be seen as part of the register allocator.
- Our loop optimisations are based on LuaJIT's. While recent versions of LuaJIT support allocation sinking, the implementation of Lambdachine was started earlier and Lambdachine's implementation of allocation sinking was done independently of LuaJIT's.

Of course, since Haskell and Lua are quite different languages, Lambdachine differs from LuaJIT in a number of significant ways:

- The semantics of the bytecode instruction set is very different. It is based mostly on the Spineless Tagless G-Machine and GRIN.
- The implementation of the interpreter is therefore also different.
- The layout of heap objects and the implementation of the garbage collector are completely different. Lambdachine instead uses GHC as the inspiration for these aspects.

- Lua values are tagged with their type information. Lambdachine does not use tagged values, so pointer information (required by the garbage collector) must be provided in a different way. Some pointer information is stored in the bytecode itself, other information is accessible from a heap object's header.

- The method of root trace selection is different. LuaJIT relies mainly on source-level loop constructs (like `for` loops) and has a less sophisticated mechanism for detecting hot functions. Lambdachine must detect all loops based on calls and tail call instructions since there are no loop constructs in the language. Haskell's support for partial- and over-application also interacts with trace selection.

- The semantics of the trace IR instructions differ in some aspects. Since trace IR instructions are fairly low-level, arithmetic and logic instructions are the same for both. Due to the different design of data structures in both languages, however, trace IR instructions for allocation and object access differ.

Basing the implementation of Lambdachine on LuaJIT was very helpful in achieving good JIT compilation performance with relatively little effort. It will also make it easier to support new instruction set architectures by adapting the code from LuaJIT. Nevertheless, Haskell and Lua are sufficiently different to make Lambdachine a non-trivial implementation effort. Appendix 8.7 gives a breakdown of how much of Lambdachine's implementation is shared with LuaJIT.

The VM code shares about 35% with LuaJIT, and if we include unit tests and the bytecode compiler (`lcc`) it reduces to 20%. We therefore feel confident in the claim that Lambdachine consists of a significant amount of original work.

5.2 Bytecode Instruction Set

Lambdachine's bytecode instruction set design is based on Boquist's GRIN language Boquist [1999]; Boquist and Johnsson [1996] and the Spineless Tagless

G-machine (STG) (Peyton Jones [1992]). Section 5.2.7 discusses the relation between Lambdachine's bytecode and GRIN.

Lambdachine's bytecode uses an explicit `EVAL` instruction to reduce an argument to WHNF. Unlike STG, the `CASE` statement always assumes that its argument is already in normal form. The following example translation illustrates this (assume `x` and `f` local variables in the surrounding context:

```
case x of
  Left y -> y
  Right z -> f z
```

$$\Downarrow$$

```
     EVAL x
     CASE x
       Left:  ->L1
       Right:  ->L2
L1:  LOADF y, x, 1
     RET1 y
L2:  LOADF z, x, 1
     CALLT f, z
```

An instruction `LOADF` x, p, o loads the field at offset o of the objected pointed to by p and stores the result in x. `RET1` simply returns from the function, and `CALLT` is a tail-call.

5.2.1 Register-based Bytecode

Lambdachine's bytecode is register-based rather than stack-based. In a stack-based interpreter operands are pushed onto an operand stack. Operations then pop their inputs off the operand stack and push the result back onto the stack. This leads to compact bytecode, but may requires more *administrative* instructions, that is, instructions that merely manipulate the contents of the stack and do not contribute to the program's computation. For example, in an imperative language, code such as `x = a * b + c` might be translated into:

```
push a   ;  stack = a : []
push b   ;  stack = b : a : []
mul      ;  stack = (a * b) : []
push c   ;  stack = c : (a * b) : []
add      ;  stack = ((a * b) + c) : []
store x  ;  stack = []
```

In an interpreter, the overhead of dispatching an instruction (decoding the instruction opcode and arguments and selecting the right implementation) often dominates the execution time. A register-based bytecode allows instructions to read and write operands anywhere in the stack. These stack locations are often called *virtual registers* and must be encoded with the instruction. The size of a single instruction thus increases, but the total number of executed instruction decreases. In a register-based bytecode, the above expression would be translated to:

```
; assume: a = r0, b = r1, c = r3, x = r2
mul r4, r0, r1   ;    tmp = a * b
add r2, r4, r3   ;    x = tmp + c
```

Davis et al. [2003] motivated the use for register-based bytecode and Shi et al. [2005] showed the overall performance advantages of this bytecode design.

5.2.2 Bytecode Format

Lambdachine uses the same simple instruction format as LuaJIT with a few extensions for instructions with an arbitrary number of arguments.

The size of each bytecode instruction is a multiple of 4 bytes. Most instructions are 4 bytes, but some instructions take multiple arguments and thus need more than 4 bytes to encode. Additionally, information about live pointer data is encoded into the instruction stream at call, evaluation and allocation instructions. This information is used by the garbage collector. The first 4 bytes of each instruction are always one of the two formats shown in Figure 5.2.

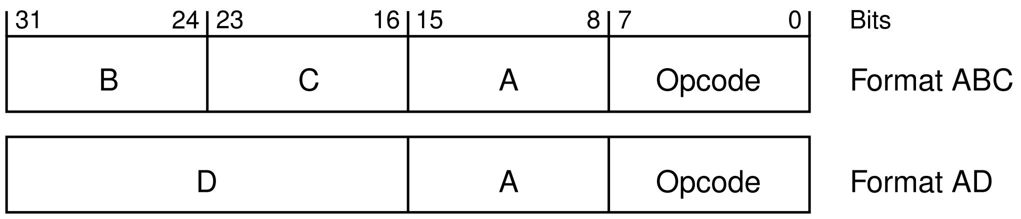


Figure 5.2: Basic bytecode instruction formats. Bit 0 is the least significant bit, bit 31 is the most significant bit.

Having fields be multiples of a byte avoids the need to use bit shifting and masking operations which helps reduce instruction decode overhead.

In our current implementation the on-disk bytecode format is the same as the in-memory format. This does not have to be the case. We expect future versions of Lambdachine to use a different format for on-disk storage which is then translated to the internal format by the bytecode loader. This will make it easier to add bytecode validation checks, improve portability and backwards compatibility.

5.2.3 Bytecode Instruction Set

Instructions can take three kinds of operands:

- `R`: A virtual register.
- `R+`: A variable number of virtual registers.
- `N`: An integer constant.
- `J`: An address encoded as an offset from the end of the current instruction.
- `P`: Pointer information encoded inline into the instruction stream.
- `J+`: A variable number of jump offsets (used only by the `CASE` instruction).

Table 5.1 shows most of Lambdachine's instructions. Only a few primitive instructions are omitted.

Table 5.1: Bytecode instruction format

Opcode	Format	Semantics
ISLT	RRJ	Branch if less than (signed integer)
ISGE	RRJ	Branch if greater than or equal (signed integer)
ISLE	RRJ	Branch if less than or equal (signed integer)
ISGT	RRJ	Branch if less than (signed integer)
ISLTU	RRJ	Branch if less than (unsigned integer)
ISGEU	RRJ	Branch if greater than or equal (unsigned integer)
ISLEU	RRJ	Branch if less than or equal (unsigned integer)
ISGTU	RRJ	Branch if greater than (unsigned integer)
ISEQ	RRJ	Branch if equal (signed or unsigned)
ISNE	RRJ	Branch if not equal (signed or unsigned)
JMP	J	Branch unconditionally
CASE	RJ+	Case analysis on constructor tag
NEG	RR	Negate (signed integer)
ADDRR	RRR	Addition (signed and unsigned integer)
SUBRR	RRR	Subtraction (signed and unsigned integer)
MULRR	RRR	Multiplication (signed and unsigned)
DIVRR	RRR	Division (signed integer)
REMRR	RRR	Modulo (signed integer)
BNOT	RR	Bitwise not (signed and unsigned integer)
BAND	RRR	Bitwise and (signed and unsigned integer)
BOR	RRR	Bitwise or (signed and unsigned integer)
BXOR	RRR	Bitwise xor (signed and unsigned integer)
BSHL	RRR	Logical bit shift left (signed and unsigned integer)
BSHR	RRR	Logical bit shift right (signed and unsigned integer)
BSHL	RRR	Arithmetic bit shift right (signed and unsigned integer)
BROL	RRR	Bit rotate left (signed and unsigned integer)
BROR	RRR	Bit rotate right (signed and unsigned integer)
MOV	RR	Register to register move (any type)
MOV_RES	RN	Load result of last function call into register

Continued on next page

Table 5.1: Bytecode instruction format

Opcode	Format	Semantics
`LOADF`	`RRN`	Load a field from a closure
`LOADFV`	`RN`	Load free variable of current closure
`LOADBH`	`R`	Load pointer to black hole into register
`LOADSLF`	`R`	Load pointer to current closure into register
`INITF`	`RRN`	Initialise field with value
`LOADK`	`RN`	Load literal into register
`ALLOC1`	`RRRP`	Allocate closure with payload size 1
`ALLOC`	`RR+P`	Allocate closure with payload size > 1
`ALLOCAP`	`RR+P`	Allocate an application thunk
`CALLT`	`RNP`	Tail call a function
`CALL`	`RR+P`	Call a function
`RET1`	`R`	Return a single result
`RETN`	`R`	Return multiple results
`EVAL`	`RP`	Evaluate to weak head normal form
`UPDATE`	`RR`	Update thunk with indirection to other closure
`FUNC`	`R`	Marks beginning of a function/thunk
`IFUNC`	`R`	Marks a function/thunk that should not be JIT compiled
`JFUNC`	`RN`	Redirects execution to a compiled trace
`IRET`	`RN`	Return, but target cannot start a new trace
`JRET`	`RN`	Return and execute the given trace
`SYNC`	–	Refresh internal interpreter state (for mode switching)
`STOP`	–	Stop the interpreter

Most instructions write their result into a register which is always the first operand.

A sequence of byte code is accompanied by a literal table which holds constants that cannot be encoded directly in the instruction stream. Such literals are mostly references to function info tables, string literals, or statically allocated closures. The `LOADK` instruction loads a literal (described by its offset in the literal table) and copies it into the specified register.

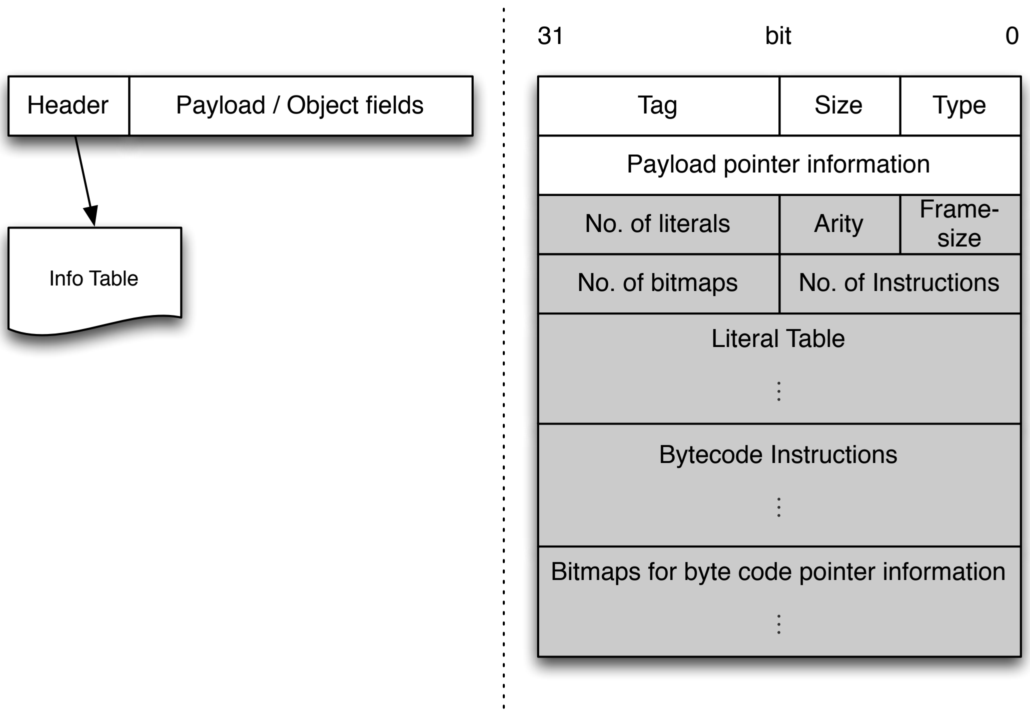


Figure 5.3: Layout of a heap object (left) and info tables (right). Grey parts are only present for object types with code attached.

Branch instructions perform a test and transfer control to the instruction at the specified object if the test succeeded. Arithmetic and bit operations work as expected.

5.2.4 Heap Object Layout

All heap objects have the same basic layout, shown in Figure 5.3, which is very similar to GHC's. Each heap object has a header which contains a pointer to a (statically allocated) info table. The rest of the object contains the payload, i.e., values of the fields of a constructor or the free variables of a thunk or closure.

The info table contains a "Type" field, which describes the type of object that it represents. Commonly used object types are:

- `CONSTR`: The object describes a constructor with fields.

- `FUN`: The object is a function. The info table will also include the function's code.
- `THUNK/CAF`: The object describes a thunk or a CAF (top-level thunk, see below). The info table will also include the code to evaluate the thunk.
- `PAP`: The object represents a partial application of a function. See Section 5.2.6.
- `IND`: The object is an indirection. When a thunk is updated with a new value the original thunk's info table is overwritten with the indirection info table and its payload is overwritten with a pointer to the value.

A CAF, short for *constant applicative form*, is a thunk whose free variables are all statically allocated. CAFs are allocated in mutable memory when the program starts and, like any other thunk, are evaluated and updated when their value is first needed[1]. Updated CAFs must be added to the garbage collector's roots because an updated CAF will contain a pointer into the program's heap.

The other fields used by all info tables are the size of an object and a bitmap describing which payload fields are pointers. For large objects, the field may just store a reference into a table of larger bitmaps. For constructor objects, the info tables also stores the constructor tag.

For objects with code attached, the info tables contains the bytecode and associated meta data. Bytecode may contain embedded pointer data (see Section 5.2.8) which is stored alongside the bytecode.

5.2.5 Stack Frame Layout

Figure 5.4 shows Lambdachine's stack frame layout. Virtual registers are accessed relative to the `Base` pointer. Register `r`i is at byte address `Base` $+ i \times$ `sizeof(Word)`. Each bytecode object specifies its *frame size*, which is the number of virtual registers used by the code.

Each frame stores the `Node` pointer, which points to the closure currently being evaluated. It is used to access free variables. A frame also stores the return

[1]CAFs do not contain any payload, so the bytecode loader reserves memory for the pointer of the indirection.

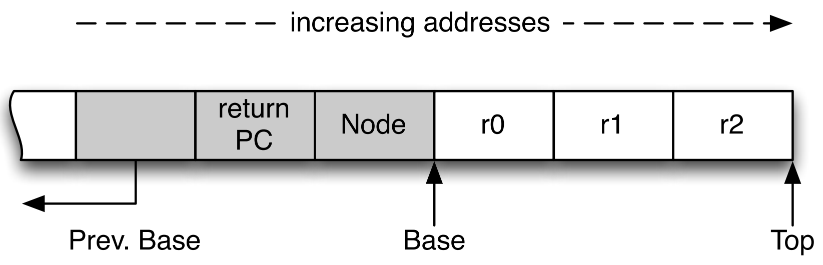


Figure 5.4: Layout of an interpreter stack frame with frame size 3.

PC, which is the address of the bytecode where execution will continue if a return instruction is executed, and a pointer to the previous stack frame.

A function of n arguments receives them in registers `r0` through `r`$(n-1)$. The `CALL` instruction takes a list of registers, creates a new stack frame and copies the contents of the registers into adjacent locations in the new frame. The `Node` pointer in the called function's frame will point to the function's closure.

The `CALLT` instruction implements tail calls which re-use the same frame. `CALLT` therefore requires all arguments to already be in registers `r0` through `r`$(n-1)$. `CALLT` sets the current node pointer to the called function and jumps to the function's body.

In fact, if the number of arguments to `CALL` and `CALLT` is not known statically, the instructions must check whether the called function actually expects the same number of parameters as there are arguments specified in the instruction. Most of the time this will be true, but not always, as discussed next.

5.2.6 Overapplication and Partial Application

Figure 5.5 shows the possible cases that may occur when calling a function and how the stack must be adjusted.

Initially, both `CALL` and `CALLT` set up the stack frame so the called function is stored in the `Node` position and all arguments are in their expected positions. We now must distinguish five cases.

- If the item is a function (closure type is "`FUN`") then we need to inspect the

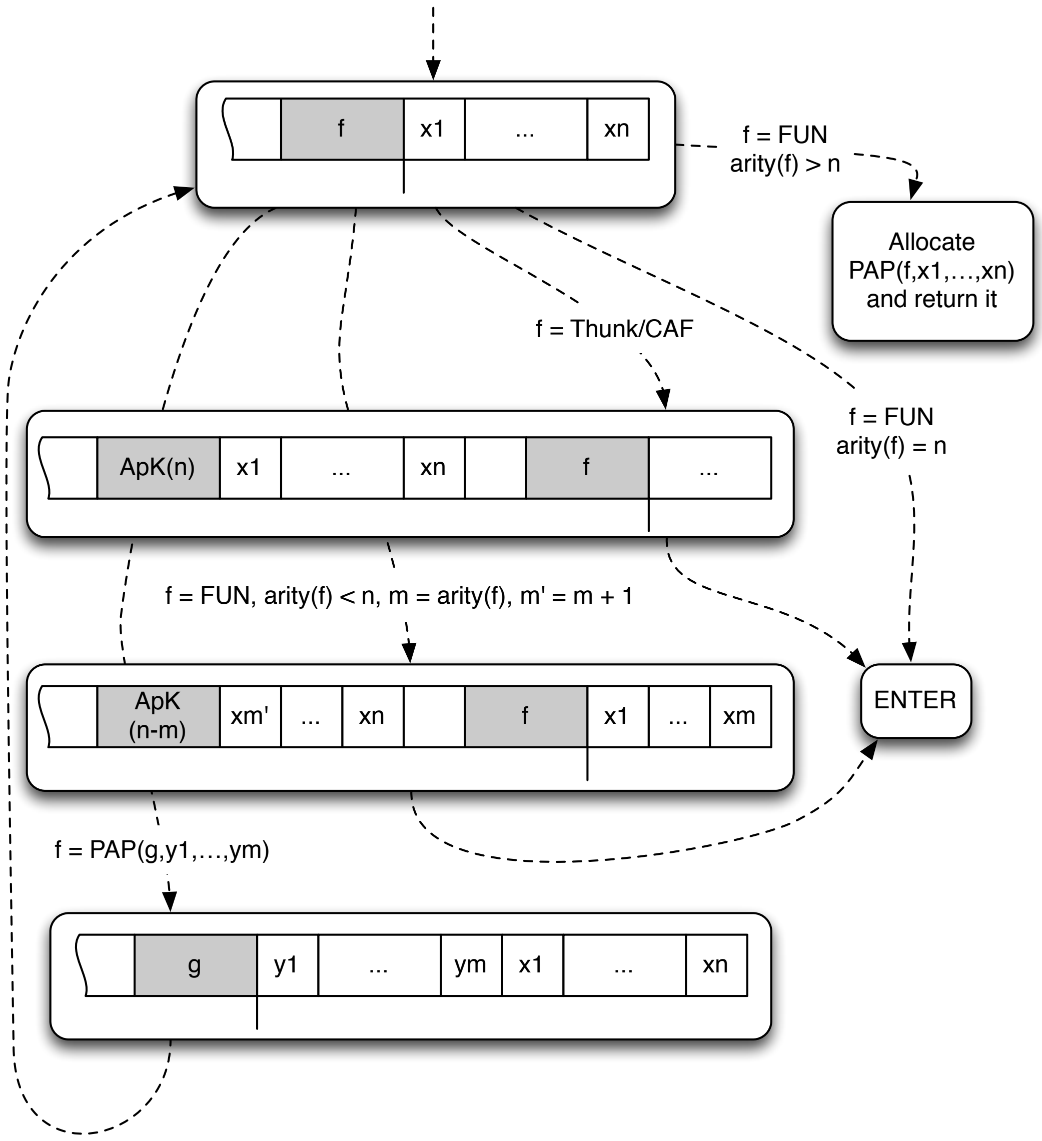


Figure 5.5: Cases that need to be handled by `CALL` and CALLT.

arity of the function.

- If the arity matches the number of arguments, we just execute its code.
- If the function expects more arguments then supplied (partial application), then we create a partial application (PAP) and return to the parent stack frame. A PAP is a heap object that holds the function and all arguments provided so far.
- If the function expects fewer arguments than supplied (overapplication), then we need to apply the function's result to the remaining arguments.. To this end we create an *application continuation* stack frame underneath the frame for `f`. A stack frame for n-ary application continuation, written `ApK(`n`)`, retrieves the result of the parent frame and applies it to n arguments.

- If the called function is a thunk or CAF (top-level thunk), then we turn the current frame into an application continuation that applies all arguments and evaluate the thunk in a new frame.
- Finally, if the function is a partial application, then we extract the function and all arguments from the PAP and add them to the current frame. We then apply the same case analysis again. Note that (as an invariant) the function field in a PAP objects must always be a FUN, thus this case is encountered at most once per function call.

The above analysis becomes more complicated in the actual implementation, because we need to ensure accurate pointer information for each argument. Every argument can come in (at least) two forms (pointer and non-pointer), so for every application continuation `ApK(`n`)` there are 2^n possible closures. Since only very few of these will ever be used in practice, we generate application closures on demand at runtime. The implementation of the two call instructions are by far the most complicated portion of the interpreter.

5.2.7 Lambdachine bytecode and GRIN

Like high-level GRIN, our bytecode uses explicit `eval` and `apply` instructions, and the bytecode's `case` instruction requires its argument to already be in normal form. The bytecode's `case` instruction also only examines the object tag. Object fields must be loaded one by one using the relevant load instructions.

GRIN (as well as GHC, Marlow and Peyton Jones [2004]) distinguishes between calls to known function and unknown functions. For calls to known functions the compiler knows the exact number of arguments and can emit a direct call to the function since no over- or partial-application can occur. For calls to unknown functions GRIN uses the generic `apply` function. Lambdachine currently does not distinguish between those cases on the level of bytecode instructions. If better interpreter performance becomes desirable, adding a new instruction type for statically known functions would be a low-hanging fruit.

Boquist's GRIN had high-level and low-level variants of many of its operations. For example, the initial translation from the core language into GRIN constructs would include polymorphic `eval`, `apply` and pattern matching with variable bindings. Further optimisation passes over the GRIN language would then turn all these constructs into first-order constructions that can be mapped easily onto a RISC machine architecture. Lambdachine's bytecode is very similar to low-level RISC except for `EVAL` and call instructions.

Boquist required program-analysis to efficiently turn polymorphic occurrences of `eval` and `apply` into first order constructs. In Lambdachine's bytecode language we keep these constructs polymorphic and optimise them later using just-in-time compilation. In the interpreter `EVAL` is implemented like in STG using an indirect call. Calling unknown functions also performs an expensive runtime case analysis on the arity of the called function. Again, we rely on the JIT compiler to later specialise the program to the cases actually occurring in practise.

5.2.8 Pointer Information

Allocation instructions are annotated with the set of pointer registers that were live *before* the instruction (the "live-ins"); call and eval instructions are annotated with the pointer registers that are live *after* the instruction (the "live-outs"). The difference stems from the different uses of this pointer information.

Garbage collection is only triggered by allocation instructions:[1], namely if the allocation instruction fails to reserve memory. The pointer information from the failing allocation instruction is then used to find the roots of the current frame. The return address of the current frame will point to an instruction following a call or an eval instruction. The pointer information from that instruction describes the roots of the corresponding frame.

5.2.9 Bytecode Interpreter

Lambdachine's interpreter is written in C++. We use the "labels as values" C extension[2] to allow easy mode switching. The basic structure is shown in Figure 5.6.

A pointer to the implementation of each interpreter instruction is stored in a global *dispatch table*. There is one dispatch table for each mode supported by the interpreter (currently "normal", "recording", and "debug"). The interpreter maintains a pointer to the currently active dispatch table.

Upon entry to each instruction, the local variable `opA` contains the contents of the `A` part of the instruction (cf. Figure 5.2), and `opC` contains the contents of the `D` part. If an instruction uses the ABC format, then it first has to decode the `B` and `C` components from the `opC`. Decoding is followed by the implementation of the instruction. Finally, the instruction must transfer control to the following instruction. To do this, it needs to extract the opcode of the next instruction (pointed to by `pc`) and pre-decode the `A` and `D` parts. Finally, we need to look up the address of the implementation of the next instruction in the active dispatch table and jump to it.

The reason for pre-decoding parts of the next instruction is to hide the delay introduced by the final indirect branch. A modern out-of-order processor can execute the these instruction in parallel with the branch execution.

LuaJIT uses the same techniques (and this bytecode format is designed specifically to allow efficient interpretation), but its interpreter is implemented in hand-

[1]In future versions of Lambdachine, a stack overflow may trigger a stack resize which in turn may trigger garbage collection.

[2]`http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html`

```
typedef void *AsmFunction;

int interpret(Capability *cap, InterpreterMode mode) {

  // Dispatch table for normal execution mode.
  static const AsmFunction dispatch_normal[] = {
     ..., &&op_ADD, ...
  };

  // Dispatch table for recording mode.
  static const AsmFunction dispatch_record[] = ...;

  const AsmFunction *dispatch;  // Active dispatch table
  Word *base;                   // base[i] = register i
  BcIns *pc;             // points to *next* instruction
  uint32_t opA, opB, opC, opcode;

  if (mode == INTERP_MODE_NORMAL)
    dispatch = dispatch_normal;

  ...

  // Implementation of integer addition (signed/unsigned)
op_ADD:
  // Decode B and C operands from pre-decoded D part
  opB = opC >> 8;
  opC = opC & 0xff;
  // Implementation of instruction:
  base[opA] = base[opB] + base[opC];
  // Dispatch next instruction:
  BcIns next = *pc;
  opcode = next.opcode;
  opA = next.a;  // Pre-decode A
  opC = next.d;  // Pre-decode D
  ++pc;
  // Jump to implementation of next instruction.
  goto *dispatch[opcode];

  ...
}
```

Figure 5.6: Basic structure of the interpreter code.

Concatenating instructions:

$$\langle i_1, \cdots, i_n \rangle \oplus \langle j_1, \cdots, j_m \rangle \triangleq \langle i_1, \cdots, i_n, j_1, \cdots, j_m \rangle$$

Bytecode objects:

bco	$::=$	$\mathsf{Function}_n(I)$	Function of arity n and code I
	$\mid$	$\mathsf{Thunk}(I)$	Thunk with evaluation code I
	$\mid$	$\mathsf{Constr}(\overline{\mathit{lit}})$	Static data structure

Global bytecode compiler environment

$$\Gamma ::= x \rightarrow \mathit{bco}$$

Figure 5.7: Bytecode compiler data structures

written assembly. This gives better performance since C/C++ compilers generally do not optimise such large dispatch tables well. For the time being, we chose to prioritise ease of interpreter evolution over absolute performance.

5.3 Compiling Haskell to Bytecode

The bytecode compiler, `lcc`, interfaces with the GHC API to parse and type check the Haskell source file. The input Haskell code is then translated into GHC's Core language and optionally optimised via the GHC API. Finally, `lcc` takes the result of this transformation and translates it into Lambdachine's bytecode. Translation to bytecode is split into two phases, (1) translation to bytecode instructions and (2) virtual register allocation. The final bytecode is then written to a bytecode file to be loaded later by the virtual machine (`lcvm`).

The bytecode compiler translates the program into a collection of bytecode objects (Figure 5.7). Bytecode objects are thunks, functions, or static data structures. Bytecode objects are later turned into info tables or static closures (or both) and then written to disk. The full implementation also computes all information required by the garbage collector, but we omit this here for clarity.

Figure 5.8 shows the translation of CoreHaskell to our bytecode. This translation assumes an infinite number of temporary variables (with the same name as in CoreHaskell) which are later mapped to virtual registers via a register allocation pass. The translation scheme for expressions

$$\begin{array}{lcl} \textsc{Result}(\mathsf{Ret}, x) & = & \mathtt{RET1}\ x \\ \textsc{Result}(\mathsf{Bind}\ y, x) & = & \mathtt{MOVE}\ y, x \end{array}$$

$$\textsc{Var}\ \frac{}{\mathcal{E}[\![x]\!], C, \Gamma \rightsquigarrow \langle \mathtt{EVAL}\ x; \mathtt{MOV\_RES}\ x; \textsc{Result}(C, x)\rangle, \Gamma}$$

$$\textsc{Lit}\ \frac{t \text{ fresh}}{\mathcal{E}[\![\ell]\!], C, \Gamma \rightsquigarrow \langle \mathtt{LOADK}\ t, \ell; \textsc{Result}(C, t)\rangle, \Gamma}$$

$$\textsc{Tailcall}\ \frac{I = \langle \mathtt{MOVE}\ \mathtt{r0}, x_1; \cdots; \mathtt{MOVE}\ \mathtt{r}(n-1), x_n; \mathtt{CALLT}\ f, n\rangle}{\mathcal{E}[\![f\ x_1, \cdots, x_n]\!], \mathsf{Ret}, \Gamma \rightsquigarrow I, \Gamma}$$

$$\textsc{Call}\ \frac{}{\mathcal{E}[\![f\ x_1, \cdots, x_n]\!], \mathsf{Bind}\ y, \Gamma \rightsquigarrow \langle \mathtt{CALL}\ f, x_1 \cdots, x_n; \mathtt{MOV\_RES}\ y\rangle, \Gamma}$$

$$\textsc{Data}\ \frac{t_1, t_2 \text{fresh} \qquad I = \langle \mathtt{LOADK}\ t_1, K; \mathtt{ALLOC}\ t_2, t_1, x_1, \cdots, x_n; \textsc{Result}(C, t_2)\rangle}{\mathcal{E}[\![K\ x_1, \cdots, x_n]\!], C, \Gamma \rightsquigarrow I, \Gamma}$$

$$\textsc{DataBind}\ \frac{\mathcal{E}[\![K\ x_1, \cdots, x_n]\!], \mathsf{Bind}\ x, \Gamma \rightsquigarrow I_1, \Gamma' \qquad \mathcal{E}[\![e]\!], C, \Gamma' \rightsquigarrow I_2, \Gamma''}{\mathcal{E}[\![\mathbf{let}\ x = K\ x_1, \cdots, x_n\ \mathbf{in}\ e]\!], C, \Gamma \rightsquigarrow I_1 \oplus I_2, \Gamma''}$$

$$\textsc{Rename}\ \frac{\mathcal{E}[\![e]\!], C, \Gamma \rightsquigarrow I, \Gamma'}{\mathcal{E}[\![\mathbf{let}\ x = y\ \mathbf{in}\ e]\!], C, \Gamma \rightsquigarrow \langle \mathtt{MOVE}\ y, x\rangle \oplus I, \Gamma'}$$

$$\textsc{Fun}\ \frac{\begin{array}{c} \mathcal{E}[\![e_1]\!], \mathsf{Ret}, \Gamma \rightsquigarrow I_1, \Gamma' \\ I_2 = \langle \mathtt{MOVE}\ x_1, \mathtt{r0}; \cdots; \mathtt{MOVE}\ x_m, \mathtt{r}(m-1)\rangle \\ I_3 = \langle \mathtt{LOADFV}\ y_1, 1; \cdots; \mathtt{LOADFV}\ y_n, n\rangle \qquad \{y_1, \cdots, y_n\} = fv(e_1) \\ t, t_1 \text{ fresh} \qquad \Gamma'' = \Gamma' \cup \{t \mapsto \mathsf{Function}_m(I_2 \oplus I_3 \oplus I_1)\} \\ I_4 = \langle \mathtt{LOADK}\ t_1, t; \mathtt{ALLOC}\ x, t_1, y_1, \cdots, y_n\rangle \\ \mathcal{E}[\![e_2]\!], C, \Gamma'' \rightsquigarrow I_5, \Gamma''' \end{array}}{\mathcal{E}[\![\mathbf{let}\ x = \lambda x_1 \cdots x_m . e_1\ \mathbf{in}\ e_2]\!], C, \Gamma \rightsquigarrow I_4 \oplus I_5, \Gamma''}$$

$$\textsc{Thunk}\ \frac{\begin{array}{c} \mathcal{E}[\![e_1]\!], \mathsf{Ret}, \Gamma \rightsquigarrow I_1, \Gamma' \\ I_2 = \langle \mathtt{LOADFV}\ y_1, 1; \cdots; \mathtt{LOADFV}\ y_n, n\rangle \qquad y_i \in fv(e_1) \\ t, t_1 \text{ fresh} \qquad \Gamma'' = \Gamma' \cup \{t \mapsto \mathsf{Thunk}(I_2 \oplus I_1)\} \\ I_3 = \langle \mathtt{LOADK}\ t_1, t; \mathtt{ALLOC}\ x, t_1, y_1, \cdots, y_n\rangle \qquad \mathcal{E}[\![e_2]\!], C, \Gamma'' \rightsquigarrow I_4, \Gamma''' \end{array}}{\mathcal{E}[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!], C, \Gamma \rightsquigarrow I_3 \oplus I_4, \Gamma'''}$$

Figure 5.8: Translation of CoreHaskell expressions to bytecode.

$$\mathcal{E}[\![e]\!], C, \Gamma \rightsquigarrow I, \Gamma$$

is a function of three inputs and produces two outputs. The three input parameters are:

- The expression e to be translated.
- The binding context C in which the expression was found. A context Ret means that the result of the expression should be returned; a context Bind y means that the result of the expression should be assigned to variable y.
- The global environment Γ, which collects all info tables.

The result of the translation is a sequence of bytecode instructions, I, and an updated global environment Γ.

Rule Var simply reduces the value pointed to by x to WHNF and then stores or returns the result, depending on the context. The translation scheme potentially generates a lot of unnecessary `MOVE`, `LOADF`, and `LOADFV` instructions. The register allocator will try to allocate both source and target of a `MOVE` instruction to the same register and then delete the move instruction. A simple dead code elimination analysis can be used to remove `LOADF` and `LOADFV` instructions as well.

Rule Tailcall translates a function call in a return context into a proper tail call. To implement tail calls efficiently, we require that all n tail call arguments are in registers `r0` through `r`$(n-1)$. The actual tail call can then be implemented simply by a jump to the start of the function's bytecode.

Any other function call must create a new stack frame and is translated to a `CALL` instruction by Rule Call. Our full implementation also supports functions which return multiple arguments. These arise from uses of *unboxed tuples*, a GHC extension (GHC Team [2011]). Multiple arguments are returned by placing the n results in registers `r0` through `r`$(n-1)$ and then returning via `RETN`. The caller can then retrieve the ith result via `MOV_RES` x_i, i. This is implemented by leaving the result inside the now-dead stack frame, that is, right above the top of the stack. This means that the caller must retrieve all results before it can issue a new `CALL` as that will overwrite the results.

Rules DATA and DATABIND handle the allocation of a heap object with constructor K. Both simply translate to the multi-argument `ALLOC` instruction. We assume that an appropriate info table has been added to the global environment Γ, which describes the tag and garbage collector meta data for the object. For each constructor K there also exists a wrapper function of the same name that simply passes on all its arguments to `ALLOC` and returns the resulting pointer. When a constructor is passed as an argument to another function, a reference to the wrapper function is used.

Rules FUN and THUNK handle the allocation of functions and thunks, respectively. In each case we use `LOADFV` to load the value of a free variable from the heap into a local register. A function receives its n arguments in registers `r0` through `r`$(n-1)$, so we use `MOVE` instructions to load them into temporary registers. Recall that the register allocator will later attempt to remove these `MOVE` instructions. The instructions for setting up the arguments and reloading the free variables is then combined with the body of the bound expression and packaged up in a new (global) info table. The code from the allocation site then constructs a new heap object with the appropriate info table and the values of the free variables at the allocation site. If an expression has no free variables, we can replace the instruction `ALLOC` x, f by `LOADK` $x, \mathsf{closure}(f)$, that is, we can move the function or thunk (actually a CAF) to the top-level. Normally, GHC will have performed this transformation before our translation to bytecode is invoked.

Figure 5.9 shows Rule CASE which handles the translation of **case** expressions. It delegates to two helpers: $\mathcal{C}[\![\overline{\rho \to e}]\!], x, C, \Gamma \rightsquigarrow I, \Gamma$ generates the code for all the case alternatives and the `CASE` bytecode instruction which dispatches to the right case alternative via a branch. The $\mathcal{P}[\![\rho]\!], x \rightsquigarrow \mathit{tag}, \mathit{label}, I$ scheme generates the code for extracting the bound pattern variables from the scrutinized heap object.

5.4 Detecting hot traces

We use a simple counter-based hotness detection scheme. The target of any `CALL`, `CALLT`, `EVAL`, or `RET` instruction can cause a hot counter to be updated. Hot counters are in fact initialised to the hotness threshold and then *decremented*

$$\textsc{Case}\ \frac{\mathcal{E}[\![e_1]\!], \mathsf{Bind}\ x, \Gamma \rightsquigarrow I_1, \Gamma' \qquad \mathcal{C}[\![\overline{\rho \to e}]\!], x, C, \Gamma' \rightsquigarrow I_2, \Gamma''}{\mathcal{E}[\![\mathtt{case}\ e_1\ \mathtt{of}\ (x)\ \overline{\rho \to e}]\!], C, \Gamma \rightsquigarrow I_1 \oplus I_2, \Gamma''}$$

$$\textsc{Alts}\ \frac{\begin{array}{c}\mathcal{P}[\![K_i\ \overline{x_i}]\!], x \rightsquigarrow t_i, L_i, I_i \qquad (i \in \{1, \cdots, n\}) \\ \mathcal{E}[\![e_i]\!], C, \Gamma_i \rightsquigarrow I'_i, \Gamma_{i+1} \qquad (i \in \{1, \cdots, n\}) \\ L \text{ fresh label} \qquad I''_i = I_i \oplus I'_i \oplus \langle \mathtt{JMP}\ L \rangle \\ I = \langle \mathtt{CASE}\ x\ \{t_i : L_i \mid i \in \{1, \cdots, n\}\} \rangle \oplus I''_1 \oplus \cdots \oplus I''_n \oplus \langle L : \rangle\end{array}}{\mathcal{C}[\![K_1\ \overline{x_1} \to e_1; \cdots K_n\ \overline{x_n} \to e_n]\!], x, C, \Gamma_1 \rightsquigarrow I, \Gamma_{n+1}}$$

$$\textsc{Unpack}\ \frac{L \text{ fresh label}}{\mathcal{P}[\![K\ x_1 \cdots x_n]\!], x \rightsquigarrow \mathsf{tag}(K), L, \langle L :; \mathtt{LOADF}\ x_1, x, 1; \cdots; \mathtt{LOADF}\ x_n, x, n \rangle}$$

Figure 5.9: Translation of CoreHaskell patterns to bytecode.

on each event. The hot counter's target is considered hot if the hot counter reached zero. The interpreter is then switched to recording mode and the hot counter is reset to its initial value.

We only consider backwards branches, that is branches to a bytecode instruction with a lower in-memory address. Each bytecode sequence starts with the special `FUNC` instruction. We can overwrite this instruction with `JFUNC` to directly enter a trace. We can also overwrite it with `IFUNC` which prohibits the instruction from ever being considered as a potential trace head. This is useful, for example, for bytecode that is part of the runtime system such as application continuations (cf. Section 5.2.6). Similarly, we can replace a `RET` instruction by `IRET` to prevent the return target from ever being considered a potential trace head.

If execution leaves a trace, a generic routine is invoked that saves the contents of all registers and restores the virtual machine state before transferring control back to the interpreter (see Section 5.13.1). This generic routine also decrements a hot counter for the particular side exit. If this hot counter reached zero, then execution in the interpreter continues in recording mode.

We perform false loop filtering (cf. Section 3.3.2) as part of the trace recorder. The trace recorder maintains a list of all branches performed while in trace recording mode as well as their stack level. If a true loop is detected, the trace recorder completes. In the case of a false loop, recording simply continues.

5.5 Trace Intermediate Representation

The trace recorder is responsible for translating the behaviour of each executed interpreter instruction into an equivalent sequence of instructions in the trace compiler's intermediate representation (IR). Instructions emitted into the trace IR buffer are optimised on the fly before being appended to the trace IR buffer.

The trace IR uses a fixed-width instruction format (same as LuaJIT's). This helps with memory management and compactness. Each IR instruction stores the opcode, the result type, and up to two operands. Instructions that require more than two operands (e.g., allocation) use one of the operands to store a reference to an auxiliary structure that stores the remaining operands.

The advantage of this format is that every trace IR instruction is only 64 bits wide, and references to the result of other instructions are simply 16 bit indexes into the trace IR buffer. A more traditional format of separately heap-allocated IR instructions would be more flexible, but would also require higher memory overhead for compiler data structures. For example, on a 64 bit architecture, a single reference to another instruction would require 8 bytes of memory, which can hold a single IR instruction in our (i.e., LuaJIT's) instruction encoding.

Table 5.2 lists all trace IR instructions currently supported by Lambdachine. The first group of instructions like `LT`, `GT`, etc. are guards and do not produce a result. All guards have an associated snapshot (see Section 5.7) which describes how to adjust the virtual machine state for the interpreter if the guard fails. Arithmetic comparisons like `LT` have type information which describes, for example, whether signed or unsigned comparison should be used.

The next group in Table 5.2 are simple arithmetic and logic instructions of one or two arguments. Again, some instructions are overloaded and store additional type information.

The third group of instructions access or modify memory. Storing a value in a field[1] requires three parameters (pointer to object, offset, and value to write),

[1]Writing to a field is only used for mutable data types and for initialising mutually recursive data.

Opcode	Oper. 1	Oper. 2	Description
`LT`	ref	ref	Guard: less than
`GE`	ref	ref	Guard: greater than or equal
`LE`	ref	ref	Guard: less than or equal
`GT`	ref	ref	Guard: greater than
`EQ`	ref	ref	Guard: equal
`NE`	ref	ref	Guard: not equal
`EQRET`	ref	ref	Guard: expected return address
`EQINFO`	ref	ref	Guard: expected info table
`NEINFO`	ref	ref	Guard: any other info table
`HEAPCHK`	lit		Guard: heap overflow check
`ADD`, `SUB`, etc.	ref	ref	Addition, Subtraction, etc.
`NEG`	ref		Negation
`BAND`, `BOR`, etc.	ref	ref	Bitwise and, bitwise or, etc.
`BNOT`	ref		Bitwise not
`FREF`	ref	lit	Reference to an object field
`FLOAD`	ref		Load a field
`SLOAD`	lit	lit	Load a stack slot
`NEW`	ref	lit	Allocate new object
`FSTORE`	ref	ref	Initialise field of recursive structure
`UPDATE`	ref	ref	Update a thunk with an indirection
`SAVE`	lit		Write current snapshot to stack
`KINT`	cst		A constant encodable in 32 bits
`KWORD`	cst		A constant encodable in 64 bits
`KBASEO`	cst		A constant offset relative to the base pointer
`NOP`			No operation
`BASE`	lit	lit	Denotes a reference to the `base` pointer
`FRAME`	lit	lit	Marker for a pushed stack frame
`RET`	lit	lit	Marker for a return from a stack frame
`LOOP`			Marks the beginning of an unrolled loop
`PHI`	ref	ref	Marks a loop-variant value (SSA Φ node)

Table 5.2: IR instructions and their formats.

but our IR format only allows two operands. We therefore split a store operation into two instructions:

$$\begin{array}{rcl} r & \leftarrow & \texttt{FREF}\ \mathit{addr}\ \mathit{offset} \\ & & \texttt{FSTORE}\ r\ \mathit{value} \end{array}$$

For consistency, the first argument to `FLOAD` ("load field") also takes a reference to an `FREF` ("field reference") instruction.

The first argument to `NEW` is a reference to the info table of the object, the second is an index into a separate structure that describes the contents of the fields of the object.

Normally, we avoid all writes to the stack (i.e., the contents of the virtual registers) and keep track of the contents the stack via an abstract stack (Section 5.6). Most of the time, we only write the values back to the stack when execution leaves the trace. Sometimes, however, we have to actually write the values back to the stack, for example because the original function call was not tail-recursive. `SAVE` is then used to force values to be written to the stack.

The fourth group describes IR instructions that represent the values of constants. The code generator later tries to encode these into the instruction that uses them. `KBASEO` represents pointers that are computed by adding a constant offset to the current base pointer (which may be different for each invocation of a trace or even for each loop iteration). Such "constants" are used to represent the "previous base" pointer of a stack frame (cf. Figure 5.4).

The final group are meta instructions that do not correspond to machine code instructions. If a backwards optimisation wants to remove an instruction from the buffer it simply overwrites them with `NOP`. `BASE` is used to refer to the current value of the `base` pointer. `LOOP`, `FRAME`, and `RET` are used by the loop optimiser. `PHI` instructions are used to denote loop-variant variables. They correspond to Φ instructions from static single assignment form (SSA) (Cytron et al. [1991]).

5.6 The Abstract Stack

The interpreter reads all its inputs from the stack and also writes its results back to the stack. If we translated this behaviour naïvely into the recorded IR code, we

`r2 = r1 + r0`	$t_1 \leftarrow$ `SLOAD` 1
	$t_0 \leftarrow$ `SLOAD` 0
	$t_2 \leftarrow$ `ADD` t_1 t_0
	`SSTORE` 2 t_2
`r3 = r2 * r3`	$t_3 \leftarrow$ `SLOAD` 2
	$t_4 \leftarrow$ `SLOAD` 3
	$t_5 \leftarrow$ `MUL` t_3 t_4
	`SSTORE` 3 t_5
`r1 = r3 - r0`	$t_6 \leftarrow$ `SLOAD` 3
	$t_7 \leftarrow$ `SLOAD` 0
	$t_8 \leftarrow$ `SUB` t_6 t_7
	`SSTORE` 1 t_8

Figure 5.10: Naïve translation from interpreter (left) to IR code (right).

would get many redundant memory reads and writes. For example, Figure 5.10 shows an example translation for a sequence of assembly instructions.

Instead of emitting code that reads values from and write values to the real stack, we instead simulate the effect of these instructions using an *abstract stack*. The abstract stack simply consists of a finite map from each stack slot to a reference to an IR instruction that produced that value. More precisely, let ReadSlot and WriteSlot be the primitives that read values from the stack, and let $S[i]$ denote the value of the abstract stack at location i, then accessing the stack works as follows:

- ReadSlot(i): If $S[i]$ is defined, then return the reference at $S[i]$. Otherwise, emit the IR instruction $r \leftarrow$ `SLOAD` i and update $S[i] := r$.
- WriteSlot(i, r): $S[i] := r$, and mark $S[i]$ as written.

Figure 5.11 shows how the abstract stack avoids the need for redundant stack reads and writes. Figure 5.11 uses the notation $[a, b, c, d]$ as a short form for $S = \{1 \mapsto a, 2 \mapsto b, \cdots\}$ and uses $-$ to stand for an undefined value. Underlined values indicate that the value has been written.

We also store an abstract base pointer that indicates which entry in the abstract stack currently corresponds to virtual register `r0`. Call instructions create

Interpreter	Generated IR	Abstract Stack
`r2 = r1 + r0`	$t_1 \leftarrow$ `SLOAD` 1	$[-, t_1, -, -]$
	$t_0 \leftarrow$ `SLOAD` 0	$[t_0, t_1, -, -]$
	$t_2 \leftarrow$ `ADD` t_1 t_0	$[t_0, t_1, \underline{t_2}, -]$
`r3 = r2 * r3`	$t_4 \leftarrow$ `SLOAD` 3	$[t_0, t_1, \underline{t_2}, t_3]$
	$t_5 \leftarrow$ `MUL` t_2 t_4	$[t_0, t_1, \underline{t_2}, \underline{t_5}]$
`r1 = r3 - r0`	$t_8 \leftarrow$ `SUB` t_5 t_0	$[t_0, \underline{t_8}, \underline{t_2}, \underline{t_5}]$

Figure 5.11: Translation using abstract stack from interpreter (left) to IR code (right).

stack frames as they would for the real stack. Return instructions also generate guards for the expected return address, and if the stack frame was created on the trace then such a guard will be redundant and be optimised away. Lambdachine currently allows the stack to grow 250 entries (machine words) below or above its position relative to size on entry to the trace. Accordingly, during trace recording the abstract stack may underflow or overflow. We simply abort recording in these cases. Some of our benchmarks triggered a trace recording to be aborted due to this condition, but the trace recorder will quickly find another trace, so this does not appear to be a problem in practise.

5.7 Snapshots

If execution leaves the trace at a guard, we have to make sure that the necessary values are written back into the real stack. For this purpose before emitting a guard, we store a copy, i.e., a *snapshot* of the abstract stack before each guard.

This snapshot only needs to store the values that were written to the abstract stack. A snapshot is implemented simply as a sequence of slot number/IR reference pairs. Additionally, it stores the current position of the `base` pointer (relative to its value at the trace entry point) and the program counter. If execution leaves the trace then we re-enter the interpreter at that program point.

5.8 Forward Optimisations

IR instructions are not emitted directly to the IR buffer, but are instead sent through an optimisation pipeline. This pipeline performs forward optimisations, that is, optimisations which require only forward data flow analyses. This includes constant folding, common subexpression elimination (CSE), store-to-load forwarding, and many algebraic simplifications.

Each time the trace recorder wants to emit an instruction it puts it into a special one-instruction buffer and attempts to apply a number of rewrite rules to the instruction. Rules may rewrite this instruction in place and then either request it to be emitted to the buffer, or request the rule engine to apply more rules on the result. A rule may also return with a reference to an existing IR instruction, in which case the to-be-emitted instruction will be dropped. Otherwise, the instruction is appended to the IR buffer.

The trace recorder records one instruction at a time, thus forward optimisations are interleaved with execution. If the trace recorder finishes successfully, e.g., by finding a loop, the remaining optimisations are performed on the IR buffer. If trace recording is aborted, the contents of the IR buffer are simply discarded.

5.8.1 Constant Folding and Algebraic Simplifications

Most of the rules used by the rule rewriting engine apply algebraic simplifications including constant folding. Other rules implement optimisations like load-to-load forwarding or reading from an object that was allocated on the heap. Table 5.3 lists a large subset of the rules employed by Lambdachine.

Suppose that the trace recorder wants to emit the instruction `ADD 3 5`. This matches the pattern `ADD` k_1 k_2 where k_1, k_2 are literals, so we apply constant-folding and replace the to-be-emitted instruction with the literal 8. More precisely, we add an instruction `KINT 8` to the buffer and return a reference to that.

For a more complicated example, consider the instruction `SUB x 5`. The second rule for `SUB` transforms this into `ADD x -5`. This may have exposed new optimisation potential, so we try to match the transformed instruction against the optimisation rules again. Now consider the case that the buffer already con-

`ADD` k_1 k_2	$\Rightarrow$	$k_1 + k_2$	
`ADD` r 0	$\Rightarrow$	r	
`ADD` (`ADD` r k_1) k_2	$\Rightarrow$	`ADD` r $(k_1 + k_2)$	
`ADD` (`ADD` r k_1) 0	$\Rightarrow$	`ADD` r k_1	
`ADD` r_1 r_2	$\Rightarrow$	`ADD` r_2 r_1	if $r_1 < r_2$ [1]
`SUB` k_1 k_2	$\Rightarrow$	$k_1 - k_2$	
`SUB` r k	$\Rightarrow$	`ADD` r $(-k)$	
`SUB` 0 r	$\Rightarrow$	`NEG` r	
`SUB` r r	$\Rightarrow$	0	
`SUB` (`ADD` r_1 r_2) r_1	$\Rightarrow$	r_2	
`SUB` (`ADD` r_1 r_2) r_2	$\Rightarrow$	r_1	
`SUB` (`SUB` r_1 r_2) r_1	$\Rightarrow$	r_2	
`SUB` r_1 (`ADD` r_1 r_2)	$\Rightarrow$	`SUB` 0 r_2	
`SUB` r_1 (`ADD` r_2 r_1)	$\Rightarrow$	`SUB` 0 r_2	
`SUB` (`ADD` r_1 r_2) (`ADD` r_1 r_3)	$\Rightarrow$	`SUB` r_2 r_3	
`SUB` (`ADD` r_1 r_2) (`ADD` r_3 r_1)	$\Rightarrow$	`SUB` r_2 r_3	
`SUB` (`ADD` r_2 r_1) (`ADD` r_1 r_3)	$\Rightarrow$	`SUB` r_2 r_3	
`SUB` (`ADD` r_2 r_1) (`ADD` r_3 r_1)	$\Rightarrow$	`SUB` r_2 r_3	
`EQ` k_1 k_2	$\Rightarrow$	Drop if $k_1 = k_2$	
`NE` k_1 k_2	$\Rightarrow$	Drop if $k_1 \neq k_2$	
`EQINFO` c k	$\Rightarrow$	Drop if info$(c) = k$	c is static closure.
`FLOAD` (`FREF` (`NEW` k $[k_1 \cdots k_n]$) i)	$\Rightarrow$	k_i	
`EQINFO` (`NEW` k_1 $[\cdots]$) k_2	$\Rightarrow$	Drop if $k_1 = k_2$	

Table 5.3: IR Transformation Rules. Variables k only match literals, r matches any reference. Arithmetic simplifications assume integer arithmetic. Floating point arithmetic is only optimised in certain rare cases to preserve correctness.

tains an instruction `x = ADD y 7`. The candidate instruction now matches the pattern `ADD` (`ADD` r k_1) k_2, so we replace the candidate instruction by `ADD y 2`. This instruction is appended to the IR buffer and a reference to it is returned to the trace recorder. The instruction `x = ADD y 7` may have become dead code (because `x` may no longer be referenced). We do not check for this immediately and simply rely on dead code elimination (performed as part of code generation) to discard this instruction later on.

Rules for guards (if applicable) either drop the guard completely, or predict that the guard will always fail. The latter may only happen during loop optimisations, in which case it would predict that the next iteration through the trace will always exit the trace. In that case the trace recorder should either try to record a longer trace (effectively unroll the loop for another iteration) or discard the current trace.

5.8.2 Common Sub-expression Elimination

The purpose of common sub-expression elimination (CSE) in our setting is to find an existing instruction in the IR buffer with the same opcode and the same operands. While the source code rarely contains duplicate code, it can occur rather frequently as the result of other optimisations. We thus perform CSE before emitting almost any instruction. Only instructions with observable side effects bypass the CSE optimisation step.

With the following auxiliary structures (adapted from LuaJIT) CSE can be implemented very efficiently, i.e., without a linear search through the full IR buffer.

The IR buffer maintains an array of references (*opcodeChain*) that maps each possible IR opcode to the most recently emitted instruction of that opcode (or nil, if no such instruction exists). Each IR instruction in turn contains a field that contains a reference to the previous instruction of the same opcode. Together this gives us for each opcode a linked list of all instructions with that opcode.

With this in place CSE can be implemented as described in Figure 5.12. Since CSE tries to find existing instructions with a given opcode the algorithm only needs to traverse the linked list for the given opcode. If there is no existing instruction of that opcode the loop condition (line 5) will fail and CSE terminates immediately.

If there are multiple instructions of the given opcode there is still no need to look at all these instructions. Since the IR buffer is automatically in SSA form (due to the structure of traces), it is impossible to reference the result of an instruction before the instruction is defined. In other words, an instruction that includes a reference to r must be defined after r. The linear search loop (lines 5-10) thus need not search beyond *limit* which is set to the larger (i.e., later occurring) of the two input references (line 4).

In practice the linear search loop rarely runs for more than one iteration. It is

```
 1: function OptCSE($opc : Opcode; op_1, op_2 : IRRef$)
 2:     $ref, limit : IRRef$
 3:     $ref \leftarrow opcodeChain[opc]$
 4:     $limit \leftarrow max(op_1, op_2)$
 5:     while $ref > limit$ do                                ▷ False if $ref =$ nil
 6:         if $buffer[ref].op_1 = op_1 \land buffer[ref].op_2 = op_2$ then
 7:             return $ref$
 8:         end if
 9:         $ref \leftarrow buffer[ref].prev$
10:     end while
11:     return nil
12: end function
```

Figure 5.12: Common sub-expression elimination. This function takes the opcodes and operands of an instruction and tries to find an instruction with the same opcode and operands in the buffer. If such an instruction is found, it returns a reference to it. Otherwise, it returns nil.

therefore efficient to run CSE before emitting each instruction into the IR buffer.

5.9 Heap Allocation

Allocation of objects on the heap is done using *sequential allocation* (also known as *bump pointer allocation.* The generated code for allocating an object with two fields (for example) looks as follows:

```
Hp += 3;                        // 1. Reserve space (3 words).
if (Hp > HpLim)                 // 2. Check for overflow.
  goto heap_overflow;
Hp[-3] = Header;                // 3. Initialise object.
Hp[-2] = Payload1;
Hp[-1] = Payload2;
Object *p = (Object *)&Hp[-3];  // Pointer to allocated object
```

The *heap pointer* (`Hp`) points to the next free byte in the allocation area. If the heap pointer points past the *heap limit pointer* (`HpLim`), then there is no more space left in the current allocation area and we either need to select a new allocation area or invoke the garbage collector. Allocating memory for an object then consists of simply incrementing the heap pointer by the desired amount of memory and checking for a heap overflow. The object is then initialised and the allocation is complete.

5.9.1 Merging Heap Checks

In the IR we separate allocation into two instructions. `HEAPCHK` reserves the desired amount of memory and checks for a heap overflow. Since a heap overflow may cause execution to leave the trace, `HEAPCHK` is a guard. `NEW` initializes the reserved memory and cannot fail.

Multiple heap checks can be combined into one by reserving memory for several objects at once. For instance, the IR code on the left is translated into the machine code on the right:

```
    HEAPCHECK #5          Hp += 5
                          if (Hp > HpLim) goto _exit
  D = NEW A [B C]         Hp[-5] = A
                          Hp[-4] = B
                          Hp[-3] = C
                          D = &Hp[-5]
  F = NEW E [D]           Hp[-2] = E
                          Hp[-1] = D
                          F = &Hp[-2]
```

5.9.2 Handling Heap Overflows

For reasons that are described in the following section, Lambdachine's heap is divided into blocks of a fixed size, which means that heap check failures are fairly frequent. Most of the time, a heap check failure means that the current block is full and we need to grab a new free block. Only once there are no more free blocks is it necessary to invoke the garbage collector.

The default behaviour of falling back to the interpreter if a guard fails could be very expensive. For example, let us assume a block has a size of 32 KBytes and consider a loop that allocates 128 bytes per iteration. That means that every 256th iteration of the loop the heap check would fail and we would execute one iteration of the loop in the interpreter. Assume the interpreter is about $10\times$ slower than the compiled machine code, then this strategy increases the running time to $(255 + 10)/256 \approx 1.035$ or by about 3.5 percent.

Normally, frequently failing guards cause a side trace to be attached, but that is not possible in this case. In the common case, a failing heap check just means that we need to mark the current block as full, update the heap pointer and heap limit and continue execution. We generate special code that calls back into the runtime system to try and grab the next free chunk and re-enter the trace with the updated heap pointer and heap limit. No other program state is affected, it is always safe to re-enter the trace right before the heap check guard. Only if there are no more free blocks available and the garbage collector needs to be invoked does execution leave the trace.

It is possible to invoke the garbage collector directly from the trace, but that would require generating meta data which describes which registers and stack slots may contain pointers into the heap. This information is already available in the interpreter, so it is simpler to just require that the garbage collector is only invoked from the interpreter.

5.9.3 Heap Checks and Side Exits

Because we combine multiple heap checks into one heap check that requests memory for multiple allocations, there is a chance that we request more memory than needed. If execution leaves the trace at a side exit not all of that reserved memory is needed. It is easy to compute the amount of overallocation at each exit point and then store it with the snapshot data. We can use this information in the following ways.

The easiest way to avoid overallocation is to simply decrement the heap pointer if the side exit is taken. That means, we either decrement the heap pointer when switching back to the interpreter, or when entering a side trace. We only need to consider side traces because the only places where a trace may transfer control to a root trace is at the end of a trace, where there will never be overallocation.

A side trace that needs to do allocation will include its own heap check. We can combine decrementing the heap pointer with the increment of the heap check. Let o be the number of bytes overallocated by the parent trace, and let s be the number of bytes allocated in the side trace, then $a = s - o$ is the number of bytes that the adjusted heap check needs to allocate. If $a \leq 0$ then the "heap check" will always succeed and can be omitted completely (if $a = 0$) or simply decrements the heap pointer. If $a > 0$ then we might be tempted to simply increment the heap pointer by a smaller amount. That, unfortunately, will cause problems if the heap check fails.

When a heap check fails and the garbage collector has grabbed a new block, the heap pointer will most likely point to the beginning of a block. This means we now have to increment the heap pointer by the full amount (s), because there will not have been any overallocation. Incrementing the heap pointer by a smaller amount will cause heap corruption as the initialisation code will now write outside the current block and will most likely overwrite data of an adjacent block. We can fix this by pre-incrementing the heap pointer by o before retrying the heap check. This increment, however, could cause another heap overflow which would require grabbing another block. A simpler solution is to simply *not* combine the heap pointer adjustments into one operation.

Therefore, given a side trace where the parent trace over-allocated 10 words on entry we generate the following code for the side trace's based on the amount of memory allocated on the side trace:

- If the side trace requires ≤ 10 words, we simply decrement the heap pointer:

```
HEAPCHECK  #4        =>        Hp -= 6
```

- If the side trace requires > 10 words, we first decrement the heap pointer by 10 and then increment it by the desired amount. If the heap check fails, execution will re-enter the trace right before the increment instruction, and will therefore request all the memory needed by the trace:

```
HEAPCHECK  #15    =>           Hp -= 10
```

```
retry: Hp += 15
       if (Hp > HpLim) goto _exit
```

5.9.4 Garbage Collection

Lambdachine uses a simple copying garbage collector (Cheney [1970]; Jones et al. [2012], Chapter 4). A copying garbage collector copies all live objects into a new memory region. Objects that were not copied are garbage and are discarded.

A copying garbage collector has the nice property that its cost is proportional to the number of *live* objects, rather than the size of the heap. In particular, if an object is allocated and becomes unreachable by the time the garbage collector is invoked no work is required to free the object. A copying garbage collector also avoids fragmentation since surviving objects are automatically compacted.

The disadvantage of a copying collector is that (a) it requires extra space and (b) that long-lived objects may get copied many times.

Problem (a) occurs if a copying collector is implemented in a naïve way. The available heap is split into two regions of equal size, the *to-space* and the *from-space*. New objects are allocated into the to-space until it is filled up. The meaning of the two spaces is then swapped and all live objects from the from-space are copied over into the to-space. All objects remaining in the from-space are dead and are simply discarded. Allocation proceeds into the to-space.

The problem with this approach is that there is always one half of the heap that is not used by the program. A common solution is to split the heap into blocks.[1] When the allocator has filled all blocks assigned to the program, the garbage collector copies surviving objects into a reserved set of blocks. The number of reserve blocks required is determined by the survival rate r of the heap objects. If the program heap consists of N blocks, we need

$$M = \lceil rN \rceil$$

extra blocks to hold the surviving objects. For functional languages, r may be as low as 0.05 (Sewe et al. [2012]). Splitting a heap into blocks can also have other uses, for example, a unit of collection in a multi-processor system (Detlefs et al.

[1]Some objects, such as large arrays, may be allocated outside these blocks.

[2004]). Since blocks are relatively easy to implement and using blocks changes the frequency of heap overflows, we included them in Lambdachine to help with getting a fair comparison with a static compiler.

A common solution to problem (b) is the use of a *generational* garbage collector (Jones et al. [2011]). A generational collector splits the heap into multiple areas, called *generations* which are collected with decreasing frequency. New objects are allocated into the *nursery* (the youngest generation). If the nursery gets filled up only the nursery is garbage collected. Objects that survive a certain number of nursery collections are moved (*promoted*) into the next generation. If that generation is filled up then both it and the nursery are collected. In principle there could be any number of generations, but the most common configurations seem to be two generations (nursery and mature space) or three generations (nursery, aging space, and mature space).

Building a generational garbage collector, however, is not easy and time-intensive to get right (there is very little room for error). For this reason, Lambdachine does not currently use a generational collector.

Our trace compiler does not optimise the garbage collector routines, thus we do not expect JIT compilation to affect garbage collection performance directly. Still, there are certainly indirect effects:

- *Allocation rate*: The JIT compiler may eliminate some allocations, and thus decrease the frequency at which the garbage collector is invoked.
- *Object demographics*: The JIT compiler eliminates mainly short-lived objects, thus the survival ratio of the remaining objects is likely higher. This may have an effect on the efficiency of the garbage collector.

In addition, choosing a simpler garbage collector design may also affect the performance of the mutator:

- *Barriers*: A generational collector uses *write barriers* to enforce certain invariants. A write barrier is some code that is run whenever a value is written to an object. Depending on the cost of this barrier code, it may have a non-trivial impact on the mutator's execution time.

- *Locality*: Garbage collection disturbs the CPU's caches. Different garbage collection strategies may therefore affect whether the mutator's working set is in the cache or not and thus have a significant influence (in either direction) on the program's run time.

Unfortunately, we have to leave the examination of the impact of these effects to future work.

5.10 Allocation Sinking

We use snapshots to avoid emitting memory writes for scalar variables and only perform them in bulk when leaving the trace or when the trace needs to build up a stack frame. Conceptually, we are pushing these writes down into side exits thereby performing *code sinking*. We can take this idea further and do the same to heap allocations which we call *allocation sinking*.

If an object is allocated on the trace, but is unlikely to survive the full trace because a pointer to it is only mentioned in side exits, then it is probably beneficial to perform this allocation only if execution leaves the trace on a side exit. In the (hopefully) common case where execution does not leave the trace, the allocation will then be avoided.

Conceptually, we rewrite the program as follows:

```
...
Object *o = new Cons(x, y);
if (x == 0) goto exit; // o escapes
...
```

$$\Downarrow$$

```
...
if (x == 0) {
  Object *o = new Cons(x, y);
  goto exit; // o escapes
}
...
```

Allocation sinking often increases register pressure. Instead of keeping one pointer to the allocated data in a local variable (i.e., in a machine register or on the stack), we now have to keep all values stored in the object in a local variable. If we run out of registers this may cause more values to be allocated on the stack. Still, avoiding the allocation will lead to fewer invocations of the garbage collector, and thus will probably improve performance overall.

There is a more complicated interaction with the garbage collector. Since allocation sinking removes only very short-lived objects it increases the survival rate of the objects seen by the garbage collector. This will have an effect on the trade-offs chosen by the garbage collector. For example, a copying collector's efficiency increases as the survival rate decreases since its overhead is proportional to the live objects. If the live ratio increases, a mark-sweep collector or a hybrid scheme may become more appropriate.

Even if a side exit is taken frequently and a side trace is attached, allocation sinking may still provide a performance advantage. A sunken allocation is inherited by the side trace which can then sink the same allocation again, so that the allocation is only performed if execution actually leaves the side trace. Side traces are discussed further in Section 5.14.

5.10.1 The Abstract Heap

To implement allocation sinking we simply extend the idea of the abstract stack to an abstract heap. Each `NEW` instruction adds a new entry to the abstract heap. When we take a snapshot of the stack, we also take a snapshot of the heap.[1] When reconstructing the virtual machine state from the snapshot we now also have to interpret the heap snapshot and perform the necessary allocations.

5.10.2 Allocation Sinking and Loops

Allocation sinking as described above is only useful if the allocation is performed on the trace and no reference to the allocated object reaches the end of the trace.

[1]Since most data types in Haskell are immutable, we do not actually need to store anything. The snapshot will contain a reference to the `NEW` instruction and we simply mark that instruction as sunken. For mutable data types we may reference the stores that have been made, or perform an actual copy of the current contents of the object according to the abstract heap.

For many traces an object is allocated in one iteration of the trace and becomes garbage in the next iteration. To optimise these cases we need loop optimisations.

The current version of Lambdachine does not have loop optimisations enabled. An earlier version of Lambdachine had a working implementation of loop unrolling and allocation sinking. Loop peeling (see Section 6.7) first creates a loop header and a loop body. Loop-variant variables are made explicit through the use of Φ-nodes. All Φ-nodes are placed immediately at the beginning of the loop body. A Φ-node

$$x_1 \leftarrow \Phi(x_0, x_2)$$

is read as: the loop variable x_1 has the value of x_0 the first time the loop is executed and the value of x_2 in any future iteration. The variable x_2 is typically initialised after x_1.

Allocation sinking now determines which allocations cannot be sunken and sinks all other allocations. An allocation is *unsinkable* if any of the following conditions holds:

1. The result of the allocation is written to the stack or to an object allocated outside of the trace (e.g., through an update instruction).

2. A allocation depends on the result of the same allocation site from an earlier iteration. In other words, the allocation's value dependencies contain a cycle and that cycle must include at least one Φ-node. For example:

$$\begin{array}{rrcl} & x_0 & \leftarrow & \cdots \\ & t_1 & \leftarrow & \texttt{new}\ \mathsf{Int}(23) \\ & t_2 & \leftarrow & \texttt{new}\ \mathsf{Int}(42) \\ \mathit{loop}: & x_1 & \leftarrow & \Phi(x_0, x_2) \\ & y_1 & \leftarrow & \texttt{new}\ \mathsf{Cons}(t_1, x_1) \\ & x_2 & \leftarrow & \texttt{new}\ \mathsf{Cons}(t_2, y_1) \end{array}$$

   The value of x_2 depends on y_1 which depends on x_1 which in turn depends on x_2. Both x_2 and y_1 are unsinkable since they are part of the same cycle.

3. The result of the allocation is (transitively) referenced from an unsinkable

allocation. In the above example both t_1 and t_2 are unsinkable because they are referenced from the unsinkable y_1 and x_2, respectively.

It turns out that the need to perform thunk updates in Haskell makes most objects unsinkable (via the first and the last condition above). Section 6.6 explains this issue in more detail.

5.11 Trace Recording and Specialisation

The trace recorder has some freedom in how it translates the recorded interpreter instruction into an equivalent sequence of IR instructions. The choices largely relate to how much we specialise the IR instructions to the observed values.

A classic example is to specialise a computation of x^n on the n argument.

```
pow x 0 = 1
pow x n = x * pow (n - 1)
```

We could specialise a call `(pow x 3)` to the exact value of the second argument. This effectively unrolls the loop and would result in the expression: `x * x * x * 1`. As a trace this would look something like this:

```
x0 = base[0];  // load ``x'' argument
n0 = base[1];   // load second argument
if (n0 != 3) goto exit1;
x1 = x0 * x0;
x2 = x1 * x0;
... // use x2 as result of call
```

This would indeed be very efficient if the second argument is always `3`, but if there are many calls to `pow` with many different values for the second argument, then this would lead to a huge number of traces.[1]

The trace recorder therefore always has to balance optimisation potential through specialisation against an increase in the number of traces required to

[1] If the trace recorder detects that the second argument is a compile-time constant, then it is more likely that this kind of specialisation is worthwhile.

cover all hot paths of the program. Lambdachine's current trace recorder is relatively conservative and only specialises on control flow, and rarely on values.

Both `EVAL` and call instructions specialise on the info table of the evaluated or called object (`EQINFO`). If the called object is a static closure (i.e., a top-level function or thunk) then this guard will be automatically be removed by the trace IR optimiser.

The `CASE` instruction will usually specialise on the info table of its argument. In some cases, however, it is worthwhile to check that the argument's info table is *not* a certain info table. Consider the program:

```
data X = A | B | C | D | E | F | G

f :: X -> Int
f x = case x of
        A -> 5
        _ -> 42
```

If the trace recorder records the second path, it should really emit a check that the info table of `x` is not the one for the `A` constructor rather than create a trace for each constructor `B` through `G`. The trace recorder will emit `NEINFO` under these circumstances.

Return instructions simply emit a guard to check for the return address. The return address implies the size of the caller's stack frame, so no further check is required for the value of the previous `base` pointer's value.

For updates, the story is a bit more complicated. When updating top-level thunks (CAFs) some extra work needs to be done to add the updated CAF as a root to the garbage collector. We could just emit a test for whether the updated object is a CAF or a regular THUNK, but that does require two memory reads: one to read the pointer to the info table out of the object's header, and another to read the object type out of the info table. In practice, we will know the info table of the updated object already, because an update always follows the evaluation of a thunk and the trace will usually include both the code for evaluating the thunk and the update; the guard for the info table of the thunk will subsume the guard introduced by the update. The second indirection (to extract the object type from the info table) can be optimised away because info tables are never modified.

5.11.1 Specialisation and Indirections

An update overwrites the thunk with an indirection to the value to which it was evaluated. This is implemented by overwriting the info table pointer in the object's header with a pointer to the special `IND` info table; and the first payload word is overwritten by a pointer to the value.

The implementation of `EVAL` will follow the indirection and return a pointer to the value. It therefore may seem sensible that the trace recorder should emit two guards:

```
  // we evaluated object: x
if (info(x) != IND_info)
  goto exit1;
y = x[1];   //
if (info(y) != Cons_info)
  goto exit2;
...
```

This works (and is currently implemented in Lambdachine), but there is an important issue to be considered. The garbage collector removes indirections to remove the additional memory access costs. That, however, also means that the first guard above will always fail for any object that has survived at least one garbage collection pass. If the object has been allocated recently, on the other hand, then the check for the indirection is fine and useful.

If the evaluated object is more likely to be one that has already been processed by the garbage collector, then we would like the trace to contain the following code:

```
L:  if (info(x) != Cons_info)
      goto pre_exit1;
    ...
```

```
pre_exit1:
    if (info(x) != IND_info)
      goto exit1;
    x = x[1]  // follow indirection
    goto L;
```

On the other hand, if it is more likely that the object is an indirection, the following could would be preferable:

```
    if (info(x) != IND_info)
      goto L;
    x = x[1];
L:  if (info(x) != Cons_info)
      goto exit1;
```

Lambdachine currently does not generate either of these variations, yet. In fact, the interaction between specialisation and trace quality remains largely unexplored. Section 6.4 suggests that there is room for improvements, but we have to leave that to future work.

5.12 Register Allocation and Code Generation

Code generation for a trace starts with the last instruction and proceeds backwards until it reaches the first instruction. This enables registers to be allocated on the fly. Register allocation requires knowledge of the live ranges of variables, that is the range of instructions from its definition to its last use. Live ranges are used to determine which registers are available at a given program point. They are also used to guide which variables to store on the stack (spill) instead of registers if there are no more registers available.

If register allocation proceeds backwards we will encounter the last use of a variable first. We can easily find its definition site from the variable name since variables are just references to their defining IR instructions. Because instructions form a linear sequence, the length of the live range is simply the delta of the indices of the two instructions.

The register allocator maintains the following data structures:

$$\mathit{Spill} ::= \mathbb{N} \mid \textsf{none} \qquad \mathit{Reg} ::= \textsf{rax} \mid \textsf{rcx} \mid \cdots \mid \textsf{none}$$

- The current allocation $\mathit{Alloc} : \mathit{IRVar} \to \mathit{Reg} \times \mathit{Spill}$ maps each variable to its currently allocated register and spill slot. A value may be held both in a register and a spill slot at the same time. In the implementation this mapping is stored inside each instruction.
- $\mathit{Cost} : \mathit{Reg} \to \mathit{IRVar} \times \mathbb{N}$ maps each register to the variable whose value it currently holds as well as the estimated cost of not keeping this value in a register. The mapping of *Cost* must be consistent with *Alloc*, that is, whenever $\mathit{Cost}(r) = (x, n)$ then $\mathit{Alloc}(x) = (r, s)$ for each register r.

Code generation now proceeds as follows. For each instruction, starting with the last instruction and working backwards to the first instruction, the code generator performs the following steps:

1. If the instruction defines an output, ensure that the output variable is allocated to a register, then mark that register as free. If the variable has a spill slot assigned emit the necessary store.
2. Allocate a register for each input to the instruction. If the input is a constant then its value may have to be loaded into a register first if the constant cannot be encoded as an immediate argument to the instruction. If no register is available, this will also cause spill code to be emitted.
3. Finally, the instruction is encoded with the assigned registers for each of the operands. In the x86 architecture most instructions store the instruction's result in one of the input operands. The IR instructions do not enforce this restriction, so these instructions are often require an additional move instruction to be emitted.

Snapshots are treated like instructions with many inputs and no outputs: each variable mentioned in the snapshot is allocated to a register. The one difference is that if no register is available, the variable is allocated directly on the stack and no register is spilled. Variables used only in a snapshot are also marked as *weakly* allocated which makes them more likely to be spilled if there are no free registers available.

5.12.1 Spilling Registers

The goal of spilling is to decide which register to free up by evicting its current contents. Ideally, the register whose eviction would cause the least performance overhead should be chosen, but this is difficult to predict at compilation time, so heuristics must be used.

If the register currently holds the value of a constant its value can easily be reloaded on demand. On x86-64, even 64-bit constants can be loaded using a single instruction, but on most other architectures constants may have to be (re-)loaded from read-only memory.

If no constants can be spilled, weakly referenced registers should be spilled which are registers that hold values only referenced from snapshots. Such values are only needed if execution leaves the trace at the associated side exit, which is hopefully less common.

If no registers satisfy the above criteria a spill decision must be made based on the usage of the variables. A simple heuristic is to spill the register with the longest remaining live range (Poletto and Sarkar [1999]). Another simple heuristic is to count the number of uses of each variable, but that information is more expensive to compute and does not appear to be better overall (Sagonas and Stenman [2003], Section 5.3). Since code generation works backwards, the register with the longest remaining lifetime is simply the register whose IR variable is defined first.

Finally, if the trace contains an unrolled loop, then the spill heuristic will prefer to keep loop-variant variables in registers and instead spill loop-invariant values.

The *Cost* mapping encodes all these heuristics into a single integer cost value.

To spill a register we simply evict the register with the lowest cost. The spill cost is computed when a register is allocated and need not be adjusted later on, so no additional computation is required.

Unlike many register allocators described in the literature, Lambdachine's allocator does not spill the full live range of a variable, but spilling occurs whenever another instruction requires a free register. This means that we have to be careful when managing spill slots. A spill slot cannot always be reused when the defining instruction has been emitted. A spill slot can only be marked as free when all interfering variables, that is, all variables whose live ranges overlap with the spilled variable, have been defined. This information is not readily available, so we simply do not reuse spill slots.

5.13 Switching between interpreter and bytecode

The code generator for x86-64 reserves three registers for special purposes:

- `rbp` is the `base` pointer.
- `r12` is the heap pointer.
- `rsp` points to the top of the C stack.

We use `rsp`, i.e., the C stack, to access additional state from within the trace. For example, `[rsp + 0]` contains the value of the heap limit pointer (for heap overflow checks) and `[rsp + 8]` points to the end of the stack (for stack overflow checks). Spill slots are also stored on the C stack.

Entering a trace thus requires to first set up the contents of the C stack and then jumping to the first machine code instruction of the trace.

The C stack also stores the ID of the currently executing trace. This ID is set when entering a trace from the interpreter and it must also be updated each time execution switches to another trace. This ID is needed by the code that restores the VM state in case of a failing guard with no attached side trace.

5.13.1 Restoring VM State from a Snapshot

Many trace compilers generate custom code to restore the VM state, so-called *exit stubs*. We need one exit stub for each trace exit. If a trace has many potential exit points, this can amount to a significant amount of code. Furthermore, if an exit is taken frequently, then a side trace will be attached and the exit stub becomes dead code.

To avoid the overheads associated with exit stubs—code size, compilation time, and management of memory for exit stubs—we chose to use LuaJIT's strategy of employing a generic exit handler that is shared by all exits.

Each compiled trace consists of the compiled machine code, the final contents of the IR buffer, and an array of snapshots. When a guard fails it jumps to a small routine that saves the contents of all registers, pushes the exit number onto the stack and invokes the generic stack restoration routine, restoreSnapshot.

The restoreSnapshot routine reads the current trace ID from the C stack, uses the trace ID to look up the trace IR buffer and snapshots, and then uses the exit ID to look up the snapshot corresponding to the taken exit.

Each IR instruction in the IR buffer has been annotated by the code generator with the register or stack slot (or both) of where its result has been stored. Each entry in a snapshot is a reference to an IR instruction. Together with the saved register's contents, restoreSnapshot can restore the contents of each stack slot by simply iterating over the snapshot entries.

Using a generic routine is slower for each exit than simply executing pre-generated code. To balance this cost, we use (like LuaJIT) a lower hotness threshold for side exits than for root traces. Lambdachine currently uses a hotness threshold of 53 for root traces and 7 for side traces.

5.14 Linking Side Traces with the Parent Trace

Snapshots can also be used to simplify linking of side traces with the parent's guard. Consider the following excerpt of a trace:

```
...
if (info(x) != Cons_info) {
  // snapshot = [0:x, 1:y, 3:z, 4:x]
  // register allocation: x:rax, y:rcx, z:rsi
  goto exit4;
}
```

Now assume this guard fails frequently and a side trace is recorded. We would like to avoid writing the values of x, y, and z to memory, but rather read them directly from registers rax, rcx, and rsi. We therefore emit place holder IR instructions at the beginning of the side trace as follows:

```
x' = base[0], Inherited
y' = base[1], Inherited
z' = base[3], Inherited
```

We also initialise the abstract stack of the side trace to:

$$\{0 \mapsto \texttt{x'}, 1 \mapsto \texttt{y'}, 3 \mapsto \texttt{z'}, 4 \mapsto \texttt{x'}\}$$

While recording the side trace any read from, say, virtual register r0 will become a reference to x'. Once recording completed the code generator will compile all IR instructions into machine code, but will stop right before the placeholder instructions. At this point, the register allocator may have chosen different registers for x', y', z' than what was chosen for x, y, z. The code generator now emits a sequence of moves to make sure that the contents of x and x', etc. match up. For example, for the register assignment:

$$\begin{array}{lll} \{ & \texttt{x} \mapsto \texttt{rax}, & \texttt{x'} \mapsto \texttt{rcx}, \\ & \texttt{y} \mapsto \texttt{rcx}, & \texttt{y'} \mapsto \texttt{rax}, \\ & \texttt{z} \mapsto \texttt{rsi}, & \texttt{z'} \mapsto \texttt{rdi} \quad \} \end{array}$$

the code generator might emit the following code:

```
mov rdx, rax   ; tmp = rax
mov rax, rcx   ; rax = rcx
mov rcx, tmp   ; rcx = tmp
mov rdi, rsi
```

This example shows that sometimes we need a temporary register to implement a swap operation. Ideally we want the side trace to use the same register as the parent. The register allocator supports *register hints* which causes defines a preferred register for an instruction, but the allocator will only use that register if it is free. It will not spill any already allocated register to free up the preferred register.

Nevertheless, even if the code generator has to emit some move instructions, the result is generally much more efficient than writing to memory in the parent and then reading them from memory on the side trace.

Chapter 6

Evaluation

This chapter describes our performance evaluation of the trace-based JIT compiler. We start with an evaluation of the run time performance (Section 6.3) followed by an investigation of the quality of the selected traces (Section 6.4). In the second part of this chapter we explore how lazy evaluation prevents some important optimisations (Section 6.6) and suggest a method that may help such optimisations to be applicable in many cases (Section 6.8).

6.1 Implemented Features

While Lambdachine does not yet support all Haskell features, it does implement a set of features chosen to support meaningful benchmarks. In particular, Lambdachine supports the following Haskell features:

- Basic types such as `Char`, `Word`, `Int`. And all operations on them.
- Algebraic data types.
- Arbitrary precision integers via the `integer-simple` library.
- Efficient string literals stored in memory as C-style strings.
- The special `State#` data type which is used internally by the `IO` and `ST` monads.

The runtime system can record all instructions and implements a simple copying garbage collector (described in Section 5.9.4). The heap is split into blocks of 32 kilobytes. We do not simply use a conservative garbage collector (such as Boehm and Weiser [1988]) because it would make allocation more expensive and thus could noticeably change the trade-offs involved.

While the complete memory manager code is only about 950 lines of C++ code (not counting comments), it was still rather expensive in terms of implementation cost, because bugs in the garbage collector are notoriously difficult to debug. We tried to keep the implementation as close as possible to GHC's copying collector in order to reduce the risk of introducing subtle bugs.

Lambdachine's garbage collector is *not* generational, which means that long-lived data structures can introduce significant overhead due to repeatedly being copied during collection. For our performance evaluation we therefore disregard garbage collection performance, and concentrate on mutator performance, i.e., the time spent actually executing the program. Unfortunately, the two cannot be separated fully, because a garbage collector rearranges objects in memory and may therefore affect cache utilization of the program. We advise the reader to be aware of this caveat when interpreting the results in this chapter.

Lambdachine at this stage does not yet support all features of Haskell'98. Notably, Lambdachine does not yet support the following features:

- Floating point arithmetic (anything involving `Double` or `Float`).
- Arrays, either mutable or immutable.
- Mutable variables. These can be seen (and implemented) as mutable arrays of length one.
- Foreign function calls. This also implies that there are no I/O functions.
- Threads or any other kind of concurrency.
- Asynchronous exceptions.

In particular the lack of any support for I/O means that Lambdachine can not yet run regular Haskell programs which all start with a `main` function which will almost always invoke a function like `print` to produce some output. We instead rewrote all our test and benchmark programs to evaluate to a value of type `Bool`. Expected output can be tested by comparing it to a string literal, and it is usually straightforward to rewrite a program to test for an expected value.

GHC implements arbitrary precision integers (type `Integer`) either using the GNU Multiple Precision arithmetic library (GMP) or using a linked list of words called `integer-simple`. Using GMP is much more efficient (it uses arrays under the hood), but requires special support from the garbage collector. Lambdachine uses `integer-simple` and comparisons with GHC use a version of GHC compiled with `integer-simple` as well.

6.2 Benchmarks

To evaluate the performance of Lambdachine we collected a number of micro benchmarks and also modified a number of benchmarks from the *spectral* set of the `Nofib` benchmark suite (Partain [1993]) to not use I/O.

The micro benchmarks are:

- *sumupto1*: This benchmark simply calculates the sum of fixed-size integers (`Int`) from `1` to `200000000`. The core of the benchmark is simply: `sum (upto 1 n)`, however, with a custom implementation of both `sum` and `upto` that is specialised to `Int`. Because we are using custom implementations of these functions, no shortcut fusion will occur.

- *sumupto2*: Like *sumupto1*, but `sum` is generalised to work over any instance of the `Num` type class. Both `sum` and `upto` are marked as `NOINLINE` which essentially disables any optimisations by the static optimiser. The JIT compiler ignores `NOINLINE` annotations. We use this benchmark to simulate what happens if GHC for some reason decides not to inline a definition that is actually hot.

- *sumquare*: This tests nested loops by calculating `sum [ a * b | a <- [1..n], b <- [a..n] ]`, again specialised to type `Int`.

- *sumstream*: Is an implementation of *sumupto1*, but uses streams instead of lists. The full program is shown in Figure 6.13 on page 147.
- *wheel-sieve2*: This benchmark is from the *imaginary* section of `Nofib`. It computes the 40,000th prime number using the method described in Runciman [1997].
- *tak*: Is another benchmark from the *imaginary* section of `Nofib`. It consists of a single function that calls itself many times recursively.

The larger benchmarks from the *spectral* section of `Nofib` are:

- *boyer*: A term rewriter.
- *constraints*: Solves the n queens constraint solving problem using 5 different algorithms.
- *circsum*: A circuit simulator.
- *lambda*: Evaluates a lambda calculus term twice: once using a simple evaluator and once using a monadic evaluator.

6.3 Mutator Performance

Since Lambdachine has a much simpler garbage collector than GHC, we only compare the times spent executing the user program (*mutator* time) and exclude garbage collection time.

Figure 6.1 shows the relative mutator time of Lambdachine when the code has been compiled with full static optimisations (GHC's `-O2` command line option). Note, that since Lambdachine uses GHC in the bytecode compiler it gets the same optimisations, although, the compilation result is not always completely the same because both use the GHC API a bit differently.

Note that we include JIT compilation overhead in the mutator time. Most benchmarks run less than 1 second (when optimised). Benchmarks for JIT compilers often measure peak-performance by discarding the first few iterations of a

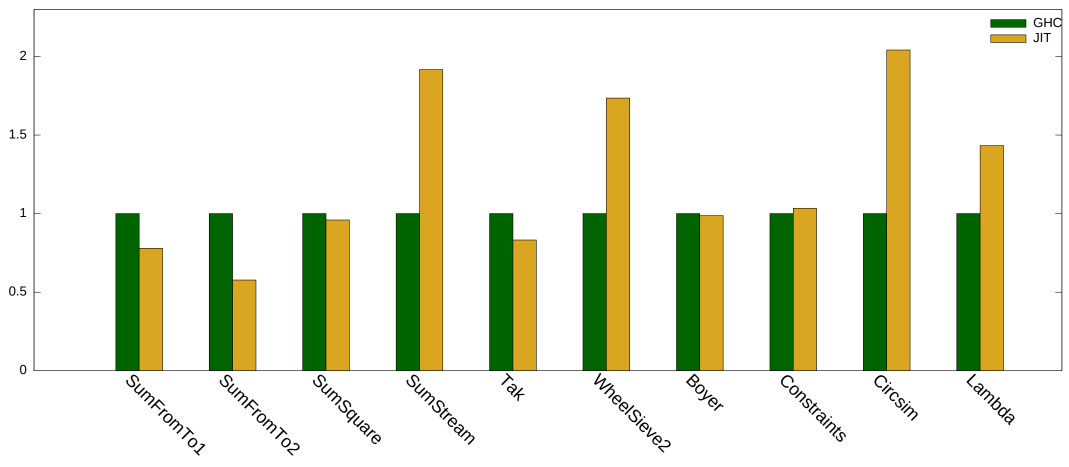


Figure 6.1: Relative mutator time of GHC and Lambdachine with full static optimisations (`-O2`); normalised to GHC mutator time.

benchmark to give the JIT compiler time to compile the benchmark kernel. We did *not* do this.

Figure 6.1 shows that Lambdachine can in some cases improve runtime performance over GHC-compiled code. This is most likely due to improvements in code locality and due to additional interprocedural optimisations in the case of SumFromTo2. For other benchmarks the JIT compiler adds overheads of up to 105%. We suspect that this is due to poor trace selection and is discussed in the following sections. The SumStream micro benchmark consists of a single loop which Lambdachine does not yet optimise very well (loop variables are currently written to and read from the stack on each iteration).

Figure 6.2 compares the speedup of Lambdachine over code that has not been optimised statically. While the JIT compiler manages to improve perfomance for most benchmarks, it causes additional slowdown for at least two benchmarks. Again, we suspect that this is caused by a failure of the trace selection heuristics.

Figure 6.3 compares the memory behaviour and garbage collector performance of GHC and Lambdachine. The second bar in each group shows the relative amount of total bytes allocated throughout the program. For some benchmarks, Lambdachine allocates a little bit less or a little bit more total memory, likely due to different object layout than GHC. This does not explain the large differ-

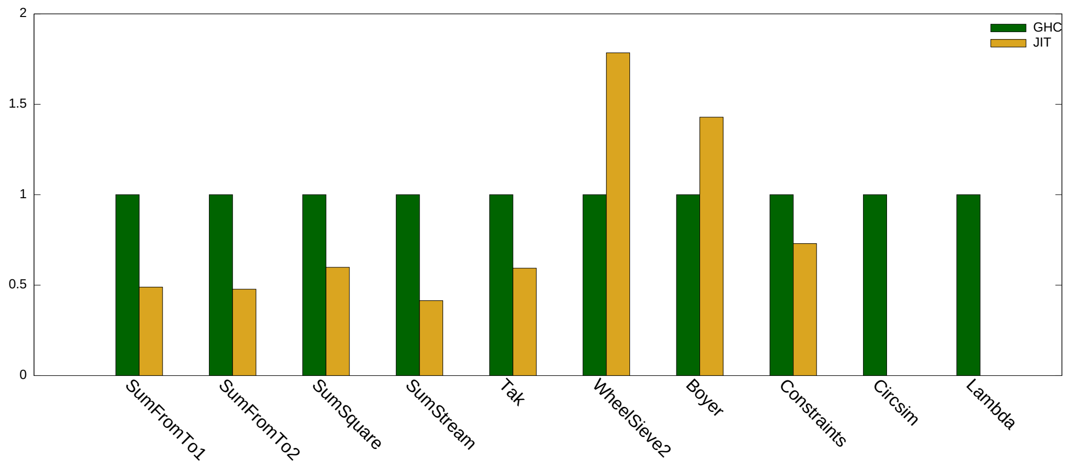


Figure 6.2: Relative mutator time of GHC and Lambdachine minimal static optimisations (-O0); normalised to GHC mutator time. No data for Circsim and Lambda.

ences for benchmarks WheelSieve2, Circsim, and Lambda. This difference may be caused by differences in static optimisations, or it may point to a bug in Lambdachine. The third bar in each group shows the total GC time relative to GHC. Lambdachine uses a non-generational copying collector. When the survival rate of allocated objects is very low, then such a collector can be very efficient. Long-lived objects, however, are copied repeatedly. GHC uses a generational garbage collector. Objects that live long are only collected during more expensive, but less frequent major collections. Accordingly, Lambdachine's garbage collector does well when the survival rate is very low (e.g., SumFromTo1, SumSquare) and worse on benchmarks where some heap objects live longer (e.g., Circsim).

The full benchmarks result can be found in the Appendix on page 182.

6.4 Trace Coverage and Trace Completion

We suspect that our larger benchmarks are negatively affected by the trace selection scheme picking traces that do not provide enough optimisation potential or that may not run for long enough.

One way evaluate the quality of the selected traces is to measure the trace

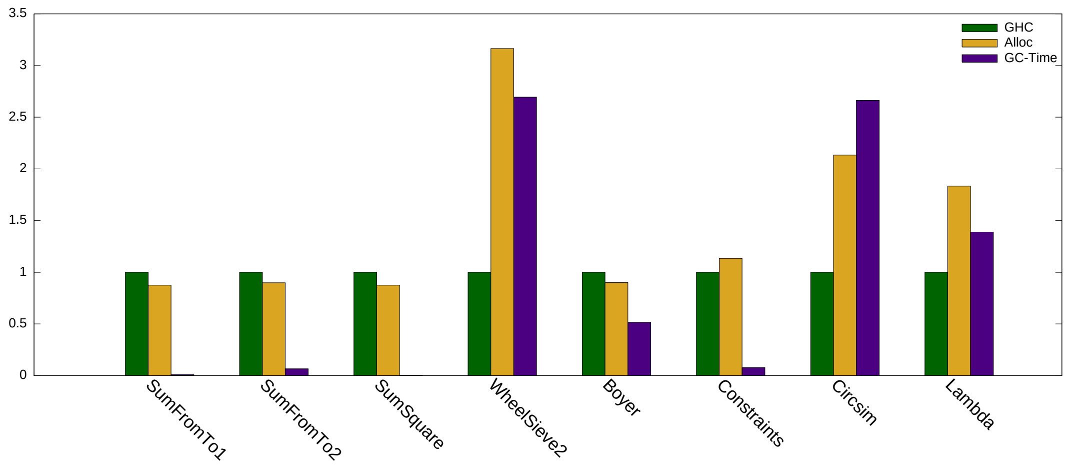


Figure 6.3: Relative total allocation and GC time, normalised to GHC

completion ratio. We call the execution of a trace *complete* if execution reaches the end of the trace and does not leave at a side exit. For a trace representing a loop this means that we executed a single iteration. We define the trace completion ratio as:

$$\mathsf{Completion.Ratio} = \frac{\mathsf{Completions}}{\mathsf{Completions} + \mathsf{Exits}} \times 100\%$$

where $\mathsf{Completions}$ is the number of times execution reached the end of the trace and Exits is the number of times execution left the trace through a side exit.

In general we want the trace completion ratio to be high, especially for traces that are executed many times. Side traces are attached very quickly, so switching back and forth between compiled code and the interpreter is not a big issue. Poor trace completion rate, however, reduces the optimisations potential, since we only optimise within a single trace.

Table 6.1 shows a summary of the trace behaviour for five of our benchmarks with a non-trivial number of traces. The "effective trace completion rate" weighs the completion rate by the number of times a trace was executed. We see that all of our benchmarks have a rather poor completion rate of less than 50%.

The remaining columns in Table 6.1 show the minimum number of traces that were needed to *cover* a certain percentage of the program's execution time. For

Table 6.1: Trace Coverage and Completion Overview

Benchmark	Effective Compl.Ratio	Traces required to cover 50%	90%	95%	99%	Total Traces
WheelSieve2	47%	3	4	5	18	69
Boyer	25%	12	78	113	208	700
Constraints	36%	15	85	128	246	1616
Circsim	27%	14	94	158	335	849
Lambda	35%	4	16	18	25	892

instance, 50% of the time spent executing benchmark *lambda* was shared between only four traces.

We actually use the number of times a trace was entered as an approximation for the amount of time spent executing it. If a trace has a low completion rate, this will attribute more time to that trace than was actually spent executing it. Nevertheless, it gives a general idea of the relative execution frequency of traces.

To gain a better understanding of where these poor completion rates are coming from we use scatter plots that plot execution frequency against completion ratio.

The figures shown in this section use scatter plots containing one point per trace. We plot the number of times the trace was entered on y-axis against the trace completion ratio (in percent). We would prefer to have the highest concentration of points in the top-right corner indicating that the most frequently executed traces have a high completion ratio. Because there quite a large amount of clustering, we show two plots for each data set: an exact scatter plot and a smoothed scatter plot. The smoothed scatter plot gives a better picture of where the highest concentration of points are at the expense of accuracy.

Figure 6.4 shows the results for the *boyer* benchmark. We see a high concentration of traces with medium execution count (around 1000), but good completion. The top left corner is, unfortunately, a bit too populated indicating a number of traces with high execution count but very poor completion ratio.

For the *constraints* benchmark, shown in Figure 6.5, the pattern looks worse. There is a clustering around 100% completion ratio, but also a large cluster

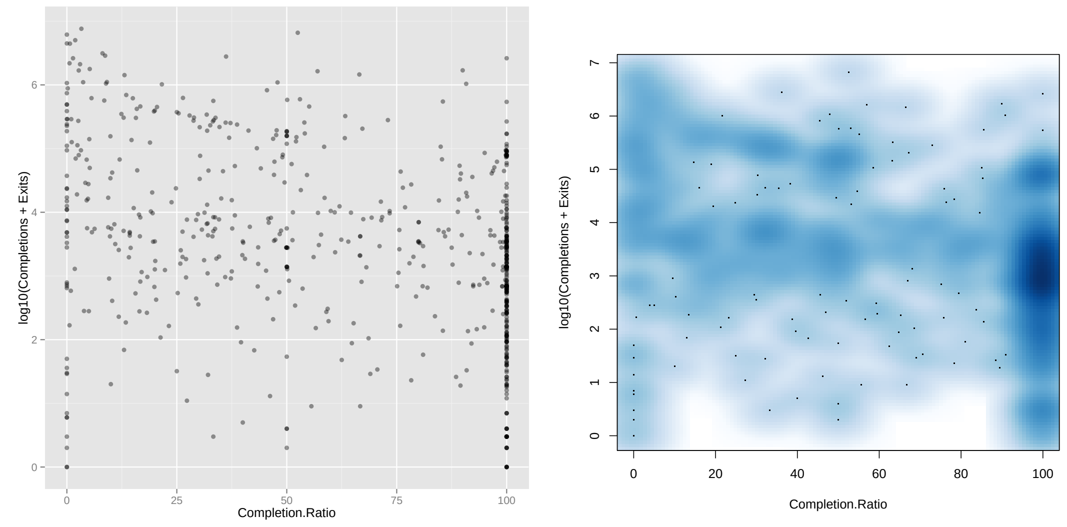


Figure 6.4: Trace completion for *boyer* benchmark (700 traces).

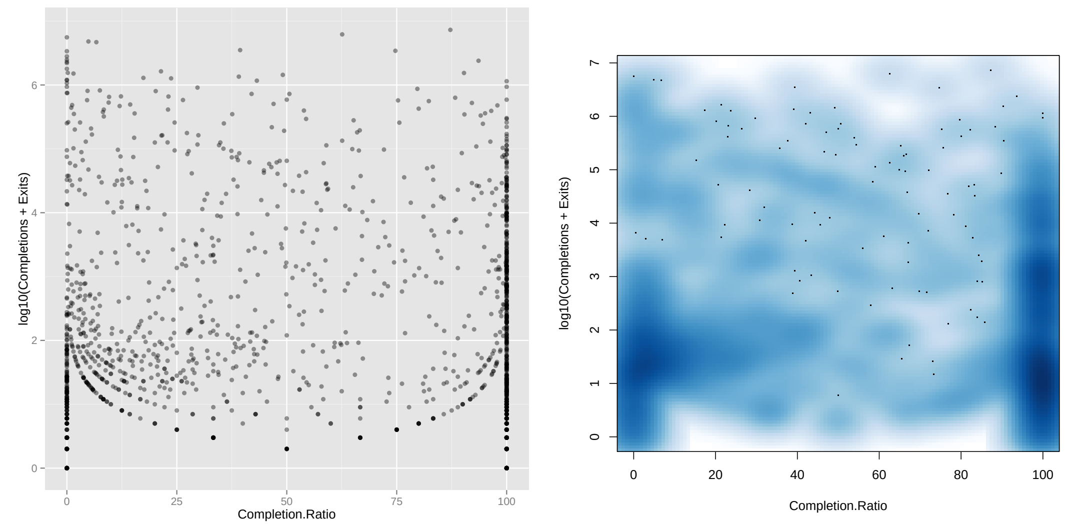


Figure 6.5: Trace completion for *constraints* benchmark (1616 traces).

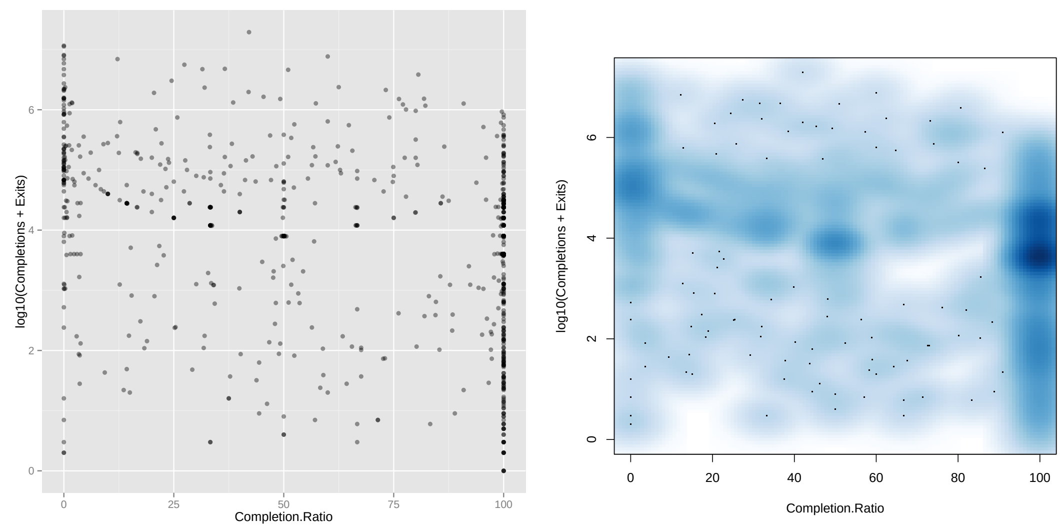


Figure 6.6: Trace completion for *circsim* benchmark (849 traces).

around 0%. Furthermore, there are many traces with execution counts < 100, for which compilation time probably does not pay off.

Figure 6.6 shows the distribution of traces for the *circsim* benchmark. This graph looks a bit better. There is again a cluster of traces with completion ration close to 100% and decent execution counts (around 10000). Unfortunately, there is also a dense area in the top left. Due to the higher execution count of those traces ($10,000 - 10,000,000$), their poor completion ratio pulls down the overall completion ratio of the program.

The data for the *lambda* benchmark, shown in Figure 6.7, exhibits behaviour that is most likely either a bug in our implementation, or pathological behaviour of the trace selection scheme. A closer look at the trace data reveals that the trace compiler generates very long sequences of traces.

Figure 6.8 shows a node for each trace and each side trace contains has an edge that points to its parent. The label on the edge denotes at which of the parent's exit points the side trace starts. There are three traces that cause a long chain of traces to follow: `17`, `322`, and `324`. Each chain is almost 300 traces long, neither of which ever runs to completion. In fact, each trace exits once through each of 7 exits and all the remaining times through exit 29. The pattern is the same for all the subsequent traces except the last one. Trace `17` is entered 3960

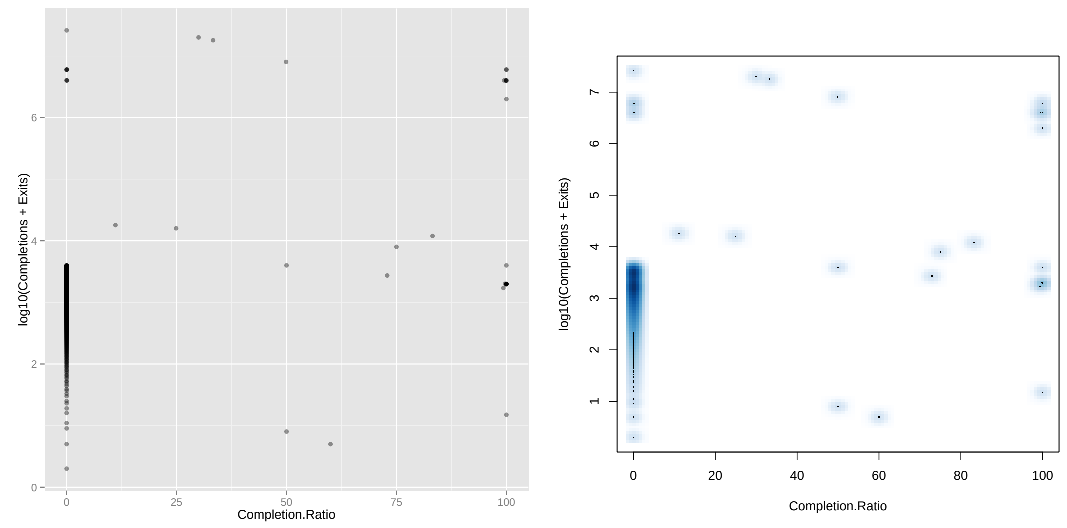


Figure 6.7: Trace completion for *lambda* benchmark (892 traces).

times, trace `18` is entered 3946 times. That, in turn, is only a tiny fraction their ancestor, trace `12`, which completes 3,995,972 out of 3,999,963 times it is entered. The time spent executing these traces is not much of an issue. The time spent recording and compiling as well as the memory they take up will be an issue.

This particular case may be a bug in the implementation. Nevertheless, an implementation should be able to protect against such pathological cases (regardless of the cause). A simple strategy is to limit the total number of side traces and their nesting depth.

Finally, Figure 6.9 shows that the *wheelsieve2* benchmark suffers from a cluster of traces with close to zero completion rate around $10,000$ trace executions. Traces with good completion ratios tend to have lower execution counts.

Clearly, trace selection is in need of improvement. All of our larger benchmarks have clusters of traces with completion ration close to zero percent. This indicates wrong specialisations (e.g., too aggressive) on the side of the trace recorder, on the one hand, and perhaps bugs in our implementation. It is likely that these deficiencies are the reason for Lambdachine comparing unfavourably to GHC for these benchmarks.

Diagnosing trace selection issues is often not easy. Programs are first optimised by GHC which already makes it harder to relate it back to the source

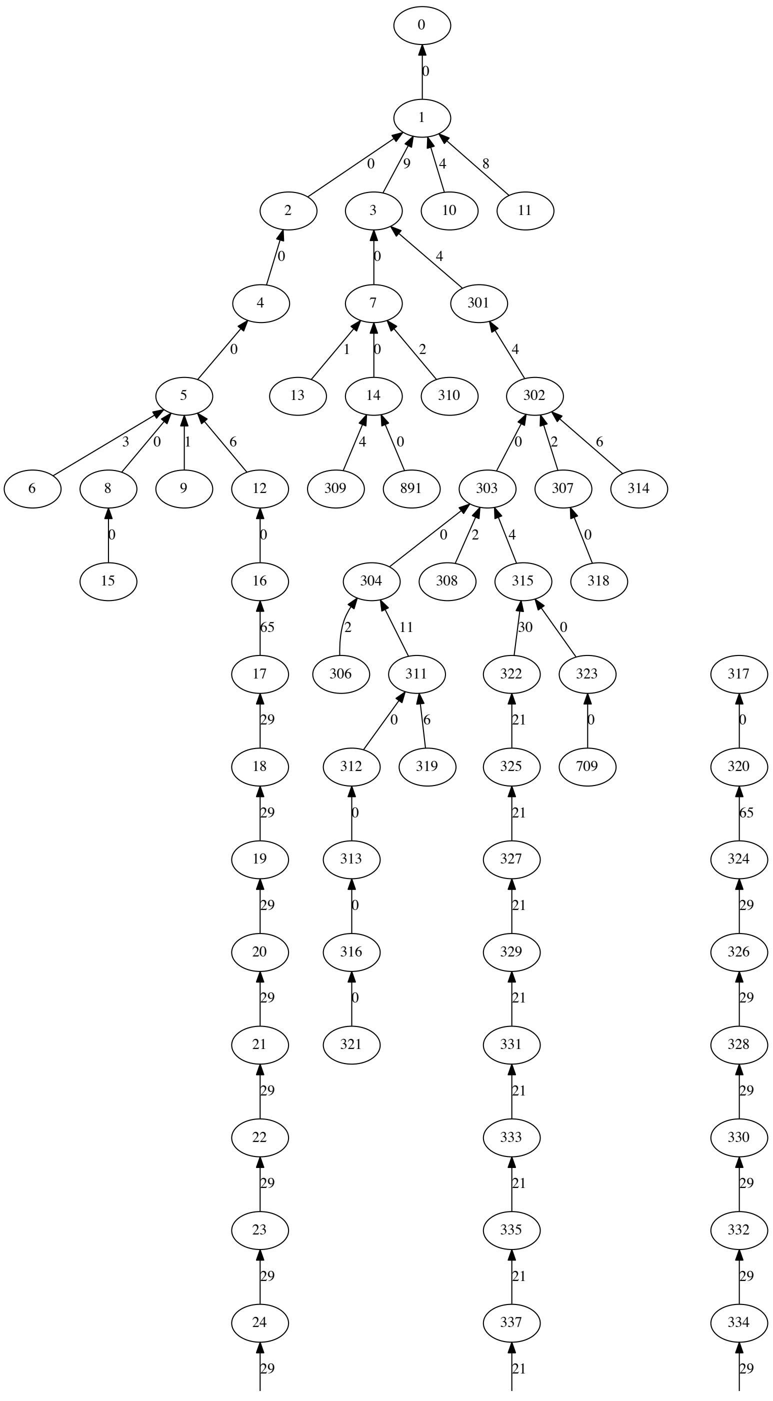


Figure 6.8: Top part of the trace graph for the *lambda* benchmark.

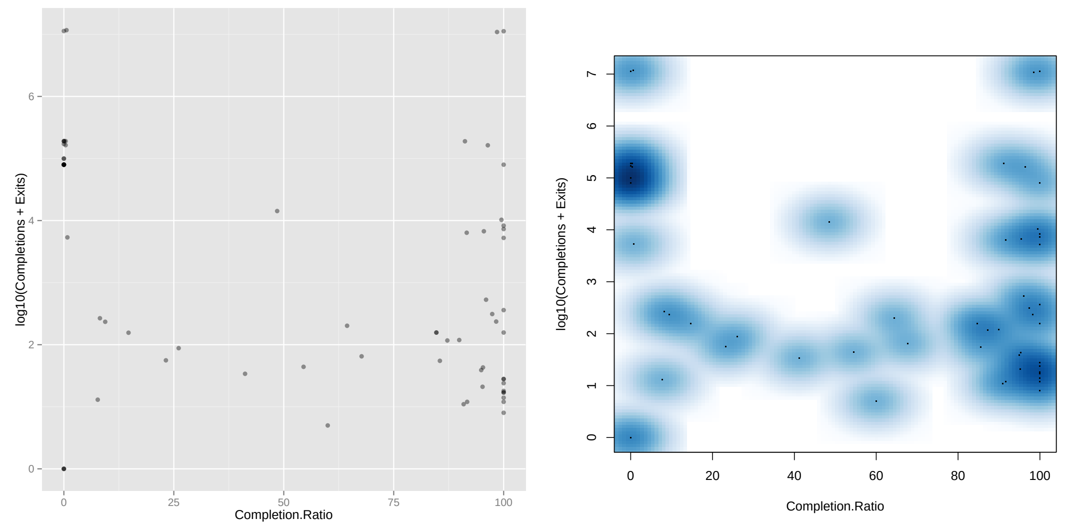


Figure 6.9: Trace completion for *wheelsieve2* benchmark (69 traces).

program. Relating traces back to the optimised program also is not straightforward. Ultimately, we would like to develop tools both to improve trace selection heuristics, and also to help users running programs on top of Lambdachine to diagnose which parts of the program are not optimised well.

6.5 Hotness Thresholds

Lambdachine uses simple counters to detect the hot parts of a program. The hotness threshold determines when a particular section of the program is considered hot. In this section we determine what effect the hotness threshold has on program performance and the number of traces that are created.

Lambdachine uses two (potentially) different hotness thresholds. One for root traces and one for side traces. The root trace threshold determines when a target of a branch executed by the interpreter is considered hot. The side trace threshold determines when a side exit is considered hot and a side trace is attached.

Figure 6.10 compares the relative mutator time for each of our benchmarks compared to our default thresholds. All programs were compiled with full static optimisations (`-O2`). We tested four root trace thresholds (103, 53, 23, 11) and

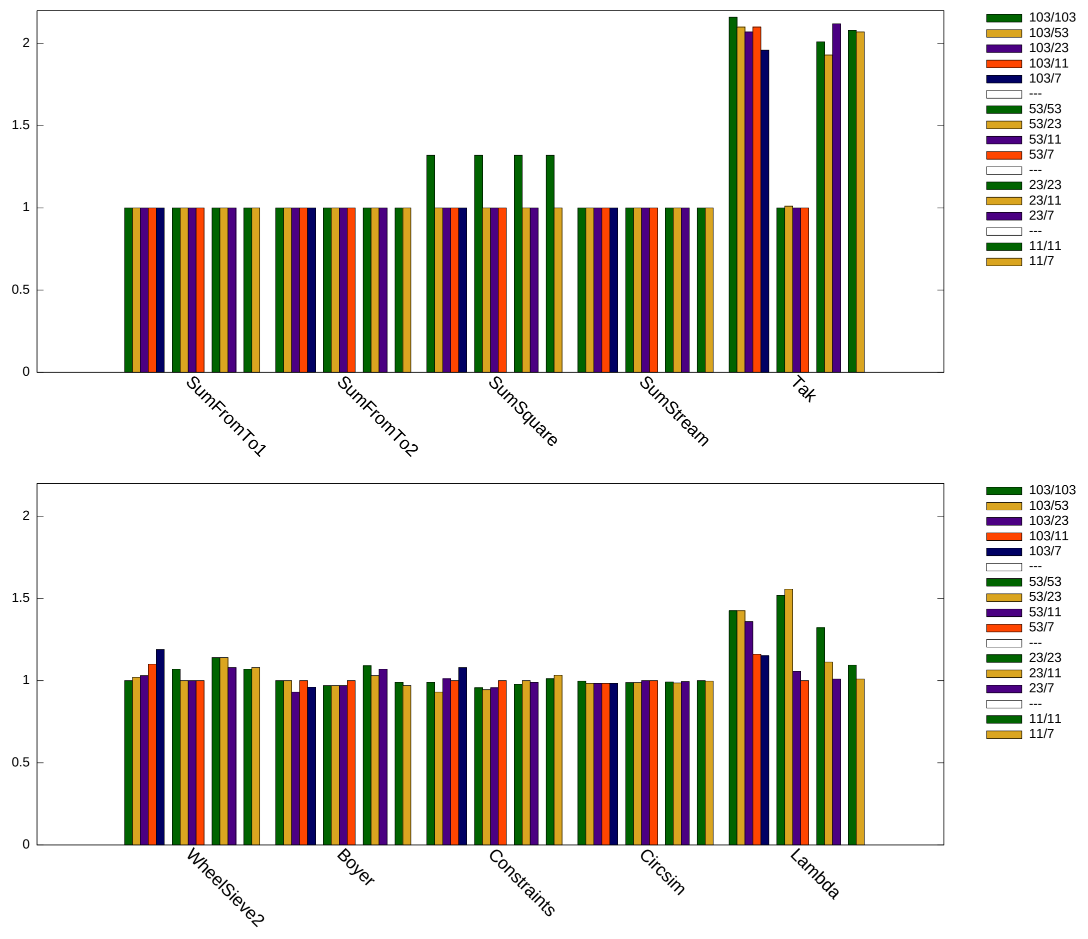


Figure 6.10: Relative mutator time for different hotness thresholds (root trace / side trace), normalised to the performance of the default thresholds (53/7) for each benchmark.

combinations with smaller or equal side trace thresholds (103, 53, 23, 11, 7).[1] We normalised the performance of each benchmark to our default settings (53/7).

Most of our micro benchmarks, with the exception of *Tak*, are not sensitive to the hotness threshold. The *SumSquare* benchmark is negatively affected if root and side trace thresholds are the same. If both thresholds are the same then we may form a root trace just moments before we would form a side trace to cover the same path. The side trace will be created soon after and link with the newly created root trace. This apparently reduces the quality of the traces overall. For the *tak* benchmark the root threshold seems to be the most important one.

The *Boyer*, *Circsim* and *Constraints* benchmarks are less sensitive to the hotness threshold. The *Constraints* works slightly better for larger side trace thresholds. Lower thresholds tend to create more traces which negatively affects compilation overhead.

Overall, there are no clearly optimal trace selection thresholds. Our default of 53/7 appears to be a reasonable choice, but additional benchmarks may suggest better defaults in the future.

Selecting good traces appears to be difficult. While some of our benchmarks are on par with GHC despite issues with selecting good traces, other benchmarks perform poorly in comparison.

There is another issue why Lambdachine stays below its performance potential. This has to do with Haskell's use of lazy evaluation and its of updates.

6.6 The problem with updates

While the Haskell specification (Peyton Jones [2003]) only specifies that Haskell programs have non-strict semantics, all widely used implementations actually implement lazy evaluation. This means that each thunk is updated (overwritten) with its value after it has been evaluated. If the same thunk is evaluated again, then `eval` will simply return the value with which the thunk was updated. This is safe, because Haskell's type system guarantees that thunk evaluation code cannot

[1]We picked prime numbers as parameters as we expected these to be less likely to occur as loop counts in benchmarks.

have side effects.[1]

Omitting the update could dramatically increase the runtime of some programs. An extreme example is the following well-known, and rather clever implementation of a function that returns the nth Fibonacci number and relies on lazy evaluation to do this efficiently.

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
fib n = fibs !! n
```

The list `fibs` initially contains only the first two numbers in the Fibonacci sequence. When the `fib` function is first invoked with an argument $n > 1$, `fibs` is evaluated until it has length $n + 1$. If `fib` is later invoked with an argument $m < n$, then the result has already been computed previously and `fib` simply returns that element from the list.

An implementation of Haskell that does not use lazy evaluation and instead uses call-by-name would turn the above function into the exponentially less efficient, but constant-space, version:

```
fibs2 _ = 1 : 1 : zipWith (+) (fibs2 ()) (tail (fibs2 ()))
fib' n = fibs2 () !! n
```

which is about as efficient as the naïve (non-memoizing) recursive implementation of calculating the nth Fibonacci number.

6.6.1 Unnecessary Updates

On the other hand, performing *unnecessary* updates can have a negative effect on performance. In the case of Lambdachine they inhibit allocation sinking and make certain traces dramatically less efficient. Figure 6.11 shows the final trace IR buffer for the example from Chapter 4 which computes the expression `(sum [1..n])`.

[1]The user may circumvent the type system by using functions such as `unsafePerformIO` to produce thunks that are not guaranteed always to produce the same result. However, if the user chooses to use such functions then it is the user's responsibility to ensure that such properties hold. Safe Haskell (Terei et al. [2012]) can be used to track or exclude Haskell modules that use features which may be used to circumvent the type system.

```
entry:
  Object *t1 = base[2]
  if (info(t1) ≠ &cl_upto_ys)
    goto exit1;
  int t2 = t1[1]
  int t3 = t2 + 1
  Object *t4 = new I#(t3)
  Object *t5 = t1[2]
  if (info(t5) ≠ I#) goto exit3
  int t7 = t5[1]
  if (t3 > t7) goto exit4
  Object *t8 = new cl_upto_ys(t3, t5)
  Object *t9 = new Cons(t4, t8)
  update (t1, t9)
  Object *t11 = base[0]
  if (info(t11) ≠ &NumDict) goto exit6
  t12 = t11[2]
  if (info(t12) ≠ &plusInt) goto exit7
  t13 = base[1]
  if (info(t13)) ≠ &I#) goto exit10
  int t15 = t13[1]
  int t16 = t3 + t15
  Object *t17 = new I#(t16)
  base[1] = t17
  base[2] = t8
  goto entry;
```

Figure 6.11: Optimised Trace IR for example from Chapter 4.

The object pointed to by `t1` has been allocated outside of the trace. This means that the object *may* be shared, i.e., that may be is reachable through some other pointer stored somewhere in the program's heap or stack. Lambdachine currently does not have an easy way of determining if there could be such a pointer to that object. Therefore, to be on the safe side, Lambdachine will always perform the update.

The following example shows an example program where this update is necessary:

```
-- may use huge amounts of memory
range = upto 1 1000000000

test = sumAux 0 range
```

```
-- possibly other references to ``range''
```

For this code, the compiled trace shown in Figure 6.11 is optimal since the thunks produced by `upto` are indeed shared and the trace must construct the list of numbers as a side effect.

The more common case, however, does not require an update:

```
-- runs fine
-# NOINLINE root #-
root limit = sumAux 0 (upto 1 limit)

test = root 1000000000
```

The new thunk `upto 1 limit` is created each time `root` is called and `sumAux` only touches each thunk once.

We would like to optimise this example to a trace that does not use any allocation at all. Using allocation sinking and loop optimisations we can do just that. If we omit the update of `t1` on Line 14 in Figure 6.11, then the allocation of `t9` becomes dead code which in turn makes `t4` dead code.

6.7 Loop Optimisation

Without loop optimisations, however, we cannot remove all allocations. For example, the thunk `t8` is live until the end of the trace. On the next iteration of the loop it becomes `t1` which is last used in Line 7. We have to remove allocations across loop iterations.

We can do that using *loop peeling*. Loop peeling consists of creating another copy of the loop which then loops back to right before the copy. In other words, we split the code into two parts: one for the first iteration, and one for all the other iterations.

This approach makes it very simple to split a loop into a loop header which contains the loop-invariant bits and a loop body which only contains loop-variant parts of the program. This approach was pioneered by LuaJIT and has also been added to PyPy (Ardö et al. [2012]).

```
entry:
  Object *t1 = base[2]
  if (info(t1) ≠ &cl_upto_ys)
    goto exit1;
  int t2 = t1[1]
  int t3 = t2 + 1
  Object *t5 = t1[2]
  if (info(t5) ≠ I#) goto exit3
  int t7 = t5[1]
  if (t6 > t7) goto exit4
  Object *t8 = new cl_upto_ys(t3, t5)
  Object *t11 = base[0]
  if (info(t11) ≠ &NumDict) goto exit6
  t12 = t11[2]
  if (info(t12) ≠ &plusInt) goto exit7
  t13 = base[1]
  if (info(t13)) ≠ &I#) goto exit10
  int t15 = t13[1]
  int t16 = t3 + t15
  Object *t17 = new I#(t16)
  // abstract stack: [t11, t17, t8]
loop:
    ; t1 ↦ t8   was: Object *t1' = base[2]
  ~~if (info(t8) ≠ &cl_upto_ys) ...~~
    ; t2 ↦ t3   was: int t2' = t8[1]
  int t3' = t3 + 1  ; t3 ↦ t3'
    ; t5 ↦ t5   was: Object *t5' = t8[2]
  ~~if (info(t5) ≠ I#) goto exit3~~
    ; t7 ↦ t7   was: int t7' = t5[1]
  if (t3' > t7) goto exit4
  Object *t8' = new cl_upto_ys(t3', t5)  ; t8 ↦ t8'
    ; t11 ↦ t11  was: Object *t11' = base[0]
  ~~if (info(t11) ≠ &NumDict) goto exit6~~
    ; t12 ↦ t12  was: 12' = t11[2]
  ~~if (info(t12) ≠ &plusInt) goto exit7~~
    ; t13 ↦ t17  was: t13' = base[1]
  ~~if (info(t13)) ≠ &I#) goto exit10~~
    ; t15 ↦ t16  was: t15' = t13[1]
  int t16' = t3' + t16   ; t16 ↦ t16'
  Object *t17' = new I#(t16')  ; t17 ↦ t17'
    ; abstract stack: [t11, t17', t8']
    ; due to allocation sinking:
  ParallelAssign([t11, (t17),  (t8),  t16,  t3,  t5],
                 [t11, (t17'), (t8'), t16', t3', t5])
  goto loop
```

Figure 6.12: Optimised Trace IR with removed `update` and peeled loop.

After loop peeling we end up with the program in Figure 6.12.

The loop peeling step works as follows. We start at the beginning of the trace and emit a new copy of the instruction through the regular optimisation pipeline. This will take into account the current contents of the abstract stack. For example, the first instruction `t1 = base[2]` gets optimised to a reference to `t8` since that is the value currently stored in abstract stack slot 2. Any future reference to `t1` in the original code will become a reference to `t8` in the copied code. We store this information in a mapping and rename inputs to copied instructions before emitting them into the IR buffer. The guard on `t1` will now become a guard on `t8`. The new guard is now trivially satisfied and can be removed. This continues until we have fully processed the original loop.

At the end of loop unrolling we obtain a new abstract stack. In Figure 6.12 this abstract stack is:

```
[t11, t17', t8']
```

At the end of the copied loop, we want to jump back to the beginning of the copied loop (not the beginning of the trace), so we have to make sure that the next execution around the loop uses the new values. We implement this by emitting an assignment:

```
t17 = t17'
t8 = t8'
```

Because `t17`, `t17'`, `t8`, and `t8'` are heap allocated structures and because we may have code that references their fields directly, we also have to ensure that those optimised field references use the right values. We therefore have to emit additional assignments:

```
t16 = t16'  ; due to t17 = t17'
t3 = t3'    ; due to t8 = t8'
t5 = t5     ; due to t8 = t8', unnecessary
```

Readers familiar with SSA will recognize these assignments as serving the same purpose as Φ nodes. Indeed, our compiler does emit `PHI` instructions to represent these moves and attempts to make most of these move instructions unnecessary by allocating the same register to both sides.

Both `t17` and `t8` are heap-allocated objects and we would like to avoid actually performing the allocation. In this example, all allocations can be sunken, so the register allocator does not actually assign a register to any of `t17`, `t17'`, `t8`, and `t8'`. Assuming register allocation can assign the same register to both `t3` and `t3'`, and to both `t16` and `t16'` we end up with the following, very efficient loop:

```
   ... initial part of trace unchanged
 loop:
   int t3 = t3 + 1;
   if (t3 > t7) goto exit4;
   int t16 = t3 + t16;
   goto loop;
```

This loop corresponds to a fully deforested and unboxed version of the original program and will likely be much more efficient than the version shown in Figure 6.11.

Unfortunately, this whole optimisation can be foiled by a single update instruction. In general we cannot safely remove the update and thus cannot take advantage of such powerful optimisations. There are two workarounds:

- One option is to make it explicit that we do not wish to cache the result of `upto`. This can be done by using streams instead of lists. A version of our example program is shown in Figure 6.13. A stream is a (co-inductive) structure consisting of an initial state and a function to compute a new state and a possible output from a given state. For the `uptoS` function the state is simply the next value in the sequence. The `sumAuxS` function simply keeps computing and summing the next value in the sequence until it reaches the end of the stream. The loop in `sumAuxS` is tail-recursive and hence runs with constant stack size.

  Coutts et al. [2007] showed that the same technique can also be used to improve static optimisations. Their framework, however, designed to be used transparentely in place of conventional list-based APIs.

Our current implementation does not implement this loop optimisation and hence we lose out to GHC on this benchmark (`SumStream` in Table 2).[1]

- Another option is to obtain the necessary information to prove that an update is unnecessary. The optimisation potential then depends on how accurate this information is.

  It is worth mentioning, however, that such automatic analyses require special tools for the user. If an expression has type `Steam Int`, then the user knows what optimisations to expect. An expression of type `[Int]`, in turn, may or may not get optimised away and subtle changes to the program could have dramatic effects on the efficiency of the resulting program. More concretely, duplicating a stream duplicates the stream's state, whereas duplicating a list that is used as a stream will now cause all the output of the stream to be stored. Since duplication is not represented in the type system the user has to consult more lower-level parts of the system (e.g., the compiler's internal structures) to detect when inadvertent sharing is affecting performance.

The following section discusses how existing Haskell implementations have performed sharing analysis statically and how we might adapt them to work well in Lambdachine.

6.8 Sharing analysis

The safety requirement for avoiding updates is that the thunk to be updated is not shared. This means that there is no possibility that `eval` will be called again on the same thunk at any point in the program's future.

One way to prove that a thunk is non-shared is to perform a static sharing analysis. The key trade-off for such an analysis is precision versus implementation complexity and performance. For Haskell, one such analysis was described in Wansbrough and Peyton Jones [2000]. It used subtyping as a restricted form of

[1]An earlier version of our implementation did support this optimisation, but due to the problem with updates has "bit-rotted" because other parts of the implementation were prioritised.

```
data Stream a = forall s. Stream !s (s -> Step s a)

data Step s a
  = Done
  | Skip s
  | Yield a s

sumAuxS :: Num a => a -> Stream a -> a
sumAuxS acc0 (Stream s0 next) = go acc0 s0
 where
  go !acc s =
    case next s of
      Done -> acc
      Skip s' -> go acc s'
      Yield x s' ->
        let !acc' = x + acc in go acc' s'

uptoS :: Int -> Int -> Stream Int
uptoS lo hi = Stream lo next
 where
   next :: Int -> Step Int Int
   next i = if i > hi then Done
                      else let !s' = i + 1 in
                           Yield i s'

test n = sumAuxS 0 (uptoS 1 n)
```

Figure 6.13: Stream version of our running example.

polymorphism to reduce the technical complexity of the analysis. Unfortunately, this system is difficult to implement and does not identify a significant fraction of unshared thunks for most programs.

More recently Hage et al. [2007] described an analysis that uses a mechanism similar to type classes to track sharing information, which may be simpler to implement than a system based on more complicated forms of subtyping. Their operational semantics also requires that heap objects are annotated with whether they are shared or not. It would require further investigation how this would be implemented in a high-performance implementation.

Boquist's GRIN compiler integrated a sharing analysis into the global points-to analysis which is required for optimising `eval`. GRIN's whole-program analysis uses an abstract interpretation with an abstract heap. Of course, in our setting we would like to avoid having to perform a whole-program analysis.

6.8.1 A dynamic sharing analysis

Given that a fully static analysis is likely to be either very difficult to implement, or not very precise, it is worth considering how a dynamic sharing analysis might work.

One option is to integrate it with the garbage collector by using a reference counting mechanism. Unfortunately, reference counting tends to be very expensive to implement because of the large number of object writes it generates. Each time a new reference to an object is created, the reference count stored inside the object must be updated. Some of these reference updates can be optimised away by performing a local analysis, but this is unlikely to remove very many reference updates. Furthermore, the reference must be stored somewhere in the object. Haskell programs tend to use many small objects (e.g., cons cells), thus adding an extra field to every object could prove expensive. In a concurrent setting we additionally may have to use atomic operations to update the reference count of objects which introduces additional costs.

So, if full reference counts are too expensive then perhaps we can use simpler reference counts? One option is to use single-bit "sticky" reference counts. Such a reference count only denotes two states: single reference (1) or possibly more than one reference (ω). Because this is imprecise, if the reference count ever reaches ω it never goes back down to 1 (hence the term "sticky"). Since this is only one bit it can potentially be stored in the object header without adding any further storage requirements for heap objects. A garbage collector can then use this information to immediately recycle the memory for objects if the reference count is 1. All other objects must be collected later using some other garbage collection mechanism.

A sticky reference count would use a minimum number of bits to maintain the desired sharing information. We lose some accuracy since a shared object can never be unshared (except, perhaps, after the garbage collector has run). Still, there are potential drawbacks with this scheme. When we want to duplicate a pointer to an object, we have to either:

1. Unconditionally update the object's reference count. In a parallel implementation this write may have to be atomic, too.

2. We can avoid the unconditional write by reading the current value first, and only update it if the object has not been shared, yet. Note, that since the object has not yet been shared, we do not need to use an atomic write.

In either case, we require a memory access even if we do not want to access the object.

An alternative strategy is to attach the sharing information to the pointer, rather than the object. In Lambdachine, objects on the heap are always allocated at a multiple of the underlying hardware's word size. On the x86-64 architecture this means an object is always aligned at a multiple of 8 bytes, thus the 3 least significant bits of a pointer to a heap object must all be 0. We can use any of these bits to store whether the object at the target address may be shared. We write $\mathsf{shared}(p) \in \{1, \omega\}$ to refer to the value of this bit:

- If $\mathsf{shared}(p) = 1$ then the pointer p is the only pointer to that object. A pointer returned from any of the `ALLOC` instructions will have this property.

- If $\mathsf{shared}(p) = \omega$ then there *may* be other pointers, in registers or from some object in the heap, to the same object.

A newly-allocated object is guaranteed to be unshared. We use a local static analysis to track when a pointer is duplicated. If it is, then the program is rewritten (at compile time) to use the special `dup` primitive which takes a pointer that may or may not be shared and returns a pointer that is shared and can be used any number of times. Operationally, if $q = \texttt{dup}\ p$ then $\mathsf{shared}(q) = \omega$ and q points to the same heap object that p did. The variable p must not be used anywhere else in the program (but q may).

Another way of viewing this is as transforming the program into a version that can be typed via linear logic (e.g., Turner et al. [1995]) and requiring that each variable which is annotated with ω to goes through a use of `dup` first.

This alone is not enough, though. Consider the following simple program:

```
alloc x = Cons x Nil

share y = let z = alloc y in
          case dup z of
            z2 -> Cons z2 (Cons z2 Nil)

test = sum (share (4 + 5))
```

The argument `y` to the `share` function is used only once, so the pointer stored inside the `Cons` cell in the function `alloc` will be tagged as unique ($\mathsf{shared}(\texttt{x}) = \mathsf{shared}(\texttt{y}) = 1$). The reference returned from `alloc` is also unique, since the `Cons` cell was just allocated. That same reference, however, is now duplicated in the body of `share`. The problem is that sharing is a transitive property: sharing a pointer to a formerly unshared object makes all objects reachable from that object shared, too. In other words, `y` is now shared, but its value has already been stored in another heap object.

A simple, but potentially very inefficient, approach would be to recursively mark all reachable from a shared objects as also shared. This would require memory writes. There would be no race conditions, though, because we only need to touch objects that were previously unshared. In fact, the traversal would stop at any reference already marked as shared. Such an approach could perhaps

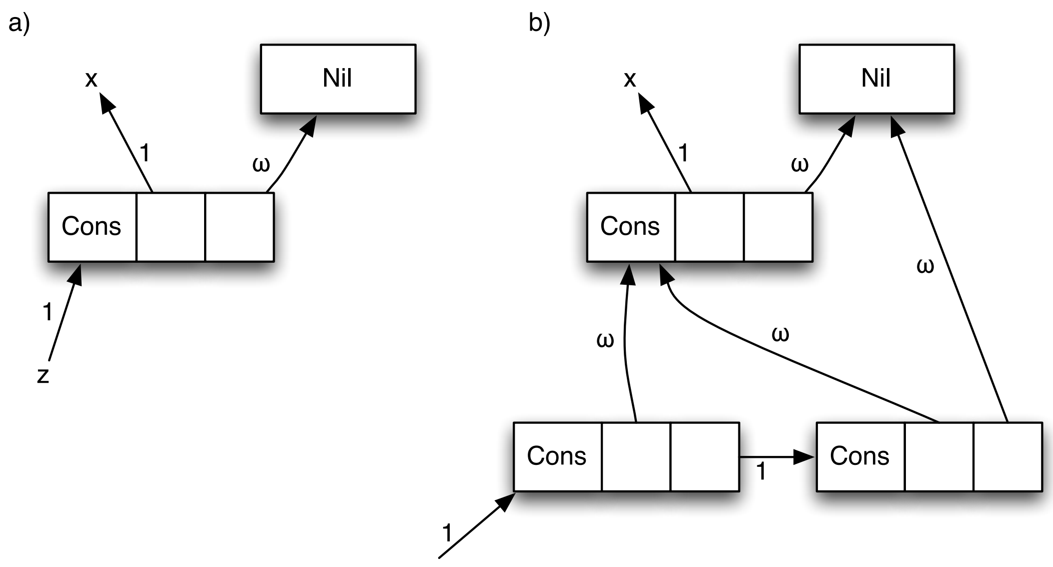


Figure 6.14: Sharing behaviour of `alloc` and `share` functions.

be workable if combined with garbage collection work, for example, where shared objects are moved into a different area of the heap.

Instead of tagging the transitive closure eagerly, we apply tagging lazily. Note that any path to a shared object (e.g., the thunk `(4 + 5)` in our example) from the roots (i.e., from local variables) has to follow a pointer tagged as shared. It is thus enough to propagate the tag information whenever we follow a pointer. More precisely, for any `LOADF` q, p, n instruction that loads a pointer field q from the object p, we ensure:

$$\mathsf{shared}(q) = \mathsf{shared}(p) \sqcup \mathsf{shared}(p[n])$$

where

$$x \sqcup y = \begin{cases} \omega, & \text{if } x = \omega \vee y = \omega \\ 1, & \text{otherwise} \end{cases}$$

This load instruction can be implemented in four instructions of x86-64 assembly by exploiting the fact that $\mathsf{bitor}(x, 0) = x$:

`mov` q, p	
`and` $q, 1$	Extract sharing bit from pointer
`and` $p, -2$	Clear sharing bit in pointer($-2 = \mathsf{bitnot}(1)$)
`or` $q, [p + 8 * n]$	Read field and combine sharing bit

On RISC-style architectures it can also be implemented in four instructions but requires one additional temporary register:[1]

`and` $tmp, p, 1$
`and` $p, p, -2$
`load` $q, p, 8 * n$
`or` q, q, tmp

If the sharing bit of p is known to be 1 (for example, because the pointer variable has been shared in the current function), this then simplifies to:

`mov` $q, [p + 8 * n - 1]$
`or` $q, 1$

The JIT compiler may choose to selectively specialise on the value of the sharing bit and emit suitable guards. For example, the compiler will want to generate different thunk evaluation code for shared thunks versus unshared thunks, but if no optimisations would be enabled later on (other than avoiding the tagging/untagging code), then the trace could remain polymorphic.

Since we have not implemented this analysis we cannot give any estimate to how well it might perform. The static part of introducing `dup` calls is fairly simple as we only have to check for variables that are shared syntactically. The changes to the execution mechanism are also very small, so overall it is indeed fairly simple to implement. A full evaluation of this scheme would also have assess both the accuracy and the performance impact of this scheme.

First, since objects can never be unshared, the sharing information would still be an overapproximation. It would be interesting how many dynamically unshared objects would be tagged as shared.

[1] The upcoming 64-bit ARM architecture seems to allow 8 tag bits in pointers than need not be masked before dereferencing a pointer. This would avoid one instruction.

Second, extracting a pointer out of an object becomes more expensive. This is likely to be a common operation, so it could have a significant impact on the overall performance. It also further complicates traces selection because we now may have to come up with another heuristic for when to specialising on the sharing state of an object. On the other hand, the optimisations it enables could lead to an overall performance improvement. Sharing can also be useful for other components of the runtime system like the garbage collector and could lead to performance improvements there.

6.9 Summary

While Lambdachine does not yet support full Haskell, we were able to evaluate its performance using a number of small benchmarks. The results are mixed (Section 6.3). While Lambdachine performs well on some microbenchmarks even improving performance slightly over GHC-generated machine code, it does perform worse on larger benchmarks with a slowdown of up to about 2× (Figure 6.1).

One particularly important issue is the poor behaviour of the trace selection heuristics (Section 6.4. Some benchmarks produce a very large amount of traces which are rarely executed fully (Table 6.1). This works against the assumptions made by the trace optimiser and is likely the main reason why Lambdachine performs poorly on some benchmarks.

We evaluated the impact of various hotness thresholds on mutator performance (Section 6.5). Most benchmarks' performance was largely unaffected by our tested range of hotness thresholds, but for two benchmarks the performance did vary noticably. The hotness threshold affects which traces are selected, so this again points to a weakness in the trace selection heuristics.

Apart from issues with trace selection, we also discussed an important optimisation issue that limits the optimisation potential of the JIT compiler. While it is useful that the JIT compiler can be used in addition to a static optimiser like GHC, we would like to be able to more effectively optimise code that GHC was not able to (e.g., due to conservative inlining). The JIT compiler's lack of knowledge about which updates can be omitted prevents such important optimisations (Section 6.6). We propose a possible analysis that could provide this information in Section 6.8, but it is very hard to estimate whether the runtime overheads of this analysis will pay off overall.

Chapter 7

Related Work

In this chapter we mention related work, mainly concerning the efficient implementation of functional programming languages. Just-in-time compilation is an active research area and JIT compilers for many widely used programming languages are under active development. Aycock [2003] provides a general overview of work in this area; in this chapter we focus on notable recent research.

7.1 Implementation of Lazy Functional Languages

The evaluation model of lazy functional languages is notably different from the Von-Neumann model implemented by most widely available hardware platforms. Early work focused on efficient methods of executing lazy functional programs on standard hardware. This was done by devising abstract machines that bridge the semantic gap between the two execution models. An abstract machine must be close enough to the high-level language to make compiling to it straightforward. At the same time, it must be possible to implement the operations of the abstract machine efficiently.

7.1.1 G-Machine and STG-Machine

The G-Machine (Augustsson [1987]; Johnsson [1987]) is a stack-based abstract machine that first constructs a representation of the program's expression graph in on the heap which is then evaluated by modifying the graph data structure. The

G-machine was later refined to the spineless G-machine (Burn et al. [1988]) which eliminated some redundant memory allocations. Peyton Jones [1992] introduced the Spineless Tagless G-machine (STG) which still serves as the foundation of today's most widely used Haskell implementation, GHC (GHC Team [2011]). The term "tagless" refers to STG's method of efficiently looking up the evaluation code for a graph node (see Section 5.2.4).

STG's implementation details have been refined a number of times. Marlow and Peyton Jones [2004] modified the calling convention, and Marlow et al. [2007] re-introduced a limited form of tagging. Efforts to support parallel execution (Marlow et al. [2009]) further affected various aspects of the implementation.

7.1.2 GRIN

Boquist's GRIN language (Boquist [1999]; Boquist and Johnsson [1996]) is another abstract machine. More precisely, GRIN can be seen as a family of abstract languages of various abstraction levels. The GRIN compiler gradually rewrites the use of higher-level constructs into lower-level constructs until the program can straightforwardly be translated into machine code.

The GRIN optimisation pipeline relies very heavily on whole-program optimisation to eliminate the need for expensive abstract machine constructs. GRIN also uses interprocedural (a.k.a., global) register allocation essentially giving each function a custom calling convention.

7.1.3 ABC Machine

The Clean programming language (Brus et al. [1987]) is based on the ABC machine (Plasmeijer and van Eekelen [1993]). The ABC machine is named after its three stacks: the argument stack, the basic value stack (or operand stack), and the control stack. There also is a parallel version of the ABC machine, called pABC (Plasmeijer and van Eekelen [1993]).

7.1.4 Other

The Reduceron (Naylor and Runciman [2008, 2010]) is a processor designed to efficiently implement graph reduction. It is designed to take advantage of parallelism offered by the use of custom hardware. The processor uses six independent memory units to reduce the single memory bus as a bottle neck. It also performs some simple dynamic analysis in parallel to the evaluator. The Reduceron implementation described by Naylor and Runciman [2010] runs on an FPGA at a clock frequency of 96 MHz whereas state of the art Intel hardware at the time ran at about 2.5 to 3 GHz. Programs running on the Reduceron are about 4× slower than the same programming compiled by GHC and running on 2010 Intel hardware (Core 2 Duo).

7.2 Lambdachine as dynamic GRIN

The key idea of GRIN is that using whole-program analysis and optimisation higher-order constructs such as `EVAL` and `apply` can be lowered into first-order constructs such as C-style `switch` statements. Conceptually, `EVAL` is implemented as a large C `switch` statement that considers any possible tag that its argument could have. Since this case analysis would be very inefficient Boquist performs a whole-program analysis to determine which tags could potentially occur at each call site of `EVAL`. The code of `EVAL` would then be *inlined* at each call site, but all impossible cases of the `switch` statement can be omitted.

Lambdachine does essentially the same transformation, but instead of relying on whole-program analysis to detect possible cases, the possible cases are explored lazily. Code is only generated for cases that occur frequently enough. Instead of a `switch` statement, the guards used by Lambdachine are simply `if` statements. The polymorphic `apply` primitive is treated the same way. Any use of a polymorphic `apply` is specialised to the most common monomorphic instantiations.

GRIN's static approach has one key advantage, though. GRIN's global program analysis is a heap points-to analysis combined with a sharing analysis. The points-to analysis determines the possible shapes of the heap node pointed to by each variable in the program. The points-to analysis is combined with a sharing analysis. This sharing analysis is crucial: it improves the accuracy of the analysis and it also allows the omission of updates. As explained in Section 6.6, omitting updates can be crucial to expose further optimisations. The sharing analysis proposed in Section 6.8 essentially embeds the information collected by GRIN's static sharing analysis into runtime data structures. While a runtime analysis is potentially more precise than a static analysis, it does, of course, also introduce a runtime cost.

Both GRIN and Lambdachine take advantage of the first-order nature of the resulting program. Knowledge about the concrete shape of objects (i.e., constructor, or which thunk) can be used to optimise later parts of the program.

7.3 Method-based JIT Compilation

The implementers of the object-oriented Self programming language developed many techniques that are now widely used, many of them are described in Hölzle [1994]. This includes polymorphic inlining caching (Hölzle et al. [1991]) and deoptimisation (Hölzle et al. [1992]).

The optimisation techniques pioneered by Self are now used by Java compilers such as the HotSpot VM (Kotzmann et al. [2008]). The use of JIT compilation required efficient optimisation techniques (e.g., Click [1995]) and the JVM has been a major driver for developing further dynamic optimisations. For example, Shankar et al. [2008] observe a program's memory allocation behaviour and tune the inlining heuristic to remove many short-lived object allocations.

7.4 Trace-based JIT Compilation

The Dynamo system (Bala et al. [1999, 2000]) was the first system to use traces to optimise programs at runtime. Dynamo worked at the machine code level and managed to improve runtime performance for programs that were compiled with less aggressive static optimisation levels. Dynamo only ran on the PA-RISC architecture. The DynamoRIO project (Bruening [2004]; Bruening et al. [2003]) adapted Dynamo's ideas and built a transparent optimisation framework for Intel hardware running Linux or Windows as the operating system. The goal of having a transparent, multi-threaded system lead to design constraints which ultimately caused performance to be worse than native execution.

Gal and Franz [2006] introduced trace-based optimisation of Java programs using trace trees. Their implementation (Gal et al. [2006]) was aimed at devices where low memory usage was more important than best performance. Trace trees later found their way into the TraceMonkey compiler for JavaScript (Gal et al. [2009]), but it has since been replaced by a method-based compiler.

The PyPy system (Rigo and Pedroni [2006]) allows authors to derive a virtual machine implementation from a language interpreter written in RPython, a statically typed language with a syntax similar to Python. Using annotations, PyPy can also derive a trace-based JIT compiler for the virtual machine (Bolz et al. [2009]). PyPy is also the name of an implementation of Python using the PyPy framework. Other programming languages have been implemented using the PyPy framework, for example PHP (Homescu and Şuhan [2011]).

LuaJIT is perhaps the most successful trace-based JIT compiler. We describe it in more detail in Section 7.9.

Inoue et al. [2011] used an existing method-based compiler for Java and added a trace-based JIT. Programs compiled by the trace-based JIT compiler were on average slightly slower (5%) than those compiled by the method JIT compiler. One problem was the quality of selected traces which they later improved via false loop filtering (Hayashizaki et al. [2011]) and refined trace selection strategies (Wu et al. [2011]).

7.4.1 Trace-based JIT Compilation for Haskell

Peixotto [2012] investigated how to take advantage of low-level optimisations to speed up Haskell programs. He found that Haskell's execution model makes it difficult to expose enough optimisation potential to a static compiler and proposed to use trace-compilation. He evaluated utilizing DynamoRIO to this end, but that did not lead to performance improvements, mostly because DynamoRIO could not optimise indirect function calls (which arise from the way GHC implements evaluation and function return). Peixotto also implemented a static trace-based optimisation scheme where the program is run once and the program is rewritten statically to use traces selected based on the profiling data collected during the first run. The rewritten program then gets compiled again (using LLVM). The rewritten program did indeed have better optimisation potential and lead to an average performance improvement of 5% over the unmodified program. Peixotto notably did not use side traces and each basic block occured in at most one trace. This suggests that there is room for further performance improvements.

7.5 Static Trace Compilation

Before traces found they way into dynamic compilers, they were used to optimise programs statically. Fisher [1979, 1981] introduced trace scheduling to translate linear microcode into horizontal microcode, that is, code where parallelism has been made explicit. Fisher optimised traces statically and relied on a profile run to pick the best trace.

The Multiflow compiler (Lowney et al. [1993]) was a continuation of this work and with the goal to build an industrial-strength trace compiler for hardware with very large instruction words (VLIW). The Multiflow compiler targeted architectures that could issue up to 28 instructions simultaneously. One motivation for pursuing this approach was the promise that the hardware could be very simple if instruction-level parallelism (ILP) is exploited by the software. Modern high-performance architectures use out-of-order execution to exploit ILP. An out-of-order CPU can also work around pipeline stalls more easily. Such stalls occur frequently due to memory accesses. A VLIW processor would simply stall the complete pipeline.

Static trace compilers select a sequence of basic blocks as the compilation unit and can use any instruction scheduling algorithm to optimise such a trace. The instruction scheduler may move instructions without restrictions (except for instruction dependencies, of course). In particular, an instruction may be moved to a different basic block. This requires compensation code to be added if execution leaves at a point after the original point, but before the new point of the instruction.

The Transmeta Crusoe processor uses a VLIW architecture internally but can run x86 code by dynamically translating it to the internal code format. The dynamic translator, called *Code Morphing Software* (CMS), translate the x86 source program into a collection of traces of VLIW traces which are kept in a trace cache. By choosing a different dynamic translator, the same could also "natively" execute other binary programs such as Java bytecode or ARM machine code. Similarly, if future versions of the hardware support new features (e.g., wider instruction words) then only the translator needs to be updated to take advantage of the new hardware features.

Modern CPU implementations dynamically translate the source instruction set (e.g., x86) into an internal instruction set. Trace caches are sometimes used internally to remember the last sequence of decoded instructions. This trace cache then only caches a single trace. The trace cache can increase performance as instruction issue rate is no longer limited by the rate at which instructions are decoded. It may also decrease power consumption since the instruction decoder is now idle.

7.6 Register-based Bytecode

Traditionally, many bytecode architectures were mostly a stack-based architecture. They are easy to implement and lead to a compact representation of the code. Some architectures use multiple stacks, but it usually involves an operand stack onto which arguments are pushed. Operations pop their arguments off the stack and push results back onto the stack.

Platforms that use a stack-based architecture are too numerous to mention. Notable examples include the Java Virtual Machine (Lindholm and Yellin [1999]) and the Common Language Infrastructure (ECMA International [2012], used by the .NET platform).

More recently, some new architectures are using register-based bytecodes. In a register-based architecture operations can read and write their arguments from arbitrary locations of the current stack frame. This means that the instruction encoding becomes larger because each instruction now needs to store the stack frame offset of each input and output operand. While each instruction becomes larger the total instruction count gets smaller, because fewer administrative instructions such as pushes, pops, or instructions that modify the ordering of the stack. It also allows common sub-expression elimination, constant propagation and a few other optimisations that are not always possible (or beneficial) in a stack-based architecture.

Davis et al. [2003] found that by using a stack-based format the total number of instructions executed was reduced by about 35% at about 45% increase of memory traffic due to the larger bytecode format. For an interpreter this increased memory traffic does not necessarily translate into large runtime overheads, however, as an implementation would simply load, say, four bytes at a time instead of one byte at a time. Thus the bytecode size increase will most likely only affect the cache efficiency.

Register-based bytecode formats are attractive for implementations where good interpreter performance is desirable. Interpreter performance is dominated by the overhead to decode and dispatch instructions, thus reducing the number of instructions dispatched by the interpreter can have a positive impact on the interpreter's performance. Shi et al. [2008] found that using a register-based bytecode format lead to a speedup over a stack-based format of 1.48× if the interpreter is implemented using a C `switch` statement and still 1.15× if the more efficient inline-threaded dispatch technique is used. Architectures based on a register-based bytecode include LuaJIT 2 (Pall [2013]), Lua 5.1 (Man [2006]), Dalvik (Section 7.8).

Implementation details often significantly affect performance. For example, LuaJIT 2 uses a simpler bytecode format compared to Lua 5.1 and uses an interpreter which is hand-written in assembly, while the standard Lua 5.1 interpreter is written in C and uses a simple `switch` statement. The LuaJIT 2 interpreter is on average over 3× faster on 32-bit x86 and around 2× faster on 64-bit x86 and (32-bit) ARM.[1]

[1]On an Intel Core2 E8400 3.0 GHz x86 processor, and a Texas Instruments OMAP4460 1.2 GHz Cortex-A9 ARM processor. Details at `http://luajit.org/performance.html`.

7.7 Implementation of Fast Interpreters

Apart from the design of the bytecode format, the implementation technique can have a large impact on the complexity and performance of the interpreter. The simplest interpreters use a simple C `switch` statement, but (depending also on the C compiler used) this is often not particularly efficient. More efficient implementations use either compiler-specific C language extensions, hand-written assembler code, various forms of code generation, or a combination of these.

The main overhead of an interpreter is the instruction dispatch and decode work. Instruction dispatch requires either an indirect branch or a sequence of binary branches. Either of these is difficult to predict by the hardware, so techniques have been developed to reduce these overheads (Casey et al. [2007]).

A techniques without code generation is *threaded code* (Bell [1973]) where each bytecode implementation directly transfers control to the next instruction instead of jumping back to a central `switch` statement. This improves performance on modern hardware because it improves branch prediction. Even better performance can be achieved using various amounts of code generation. Inline-threading (Gagnon [2002]; Gagnon and Hendren [2003]) copies the implementation of bytecode instructions into a buffer and executes the resulting code directly. Context-threading (Berndl et al. [2005]) generates a call instruction for each non-branch bytecode and custom code for branch instructions. This exposes the program's control flow to the hardware branch predictor which greatly reduces the number of mispredicted branches. The increase in performance, however, comes at the cost of portability due to the use of code generation.

Other techniques for optimising interpreters include modifications to the instruction set, or to the running program (for example to specialise on observed behaviour). A good overview is given in Casey et al. [2007].

7.8 Dalvik Virtual Machine

Google's Dalvik Virtual Machine (DVM) is the main platform for running Java-based applications on the Android operating system (Bornstein [2008]). It is thus an alternative to the Java Virtual Machine (JVM) (Lindholm and Yellin

[1999]). Dalvik uses a different bytecode format than the JVM, but application authors generate Java bytecode files which are then converted into Dalvik's own `dex` format via a tool (named `dx`).

Dalvik's bytecode is register-based in order to improve interpreter performance relative to the JVM's more compact stack-based format which, however, does less work per instruction. Dalvik additionally uses a trace-based JIT compiler. The stated reason for using a trace-based compiler over a method-based compiler as, for example, used by the HotSpot JVM (Kotzmann et al. [2008]) is to reduce the amount of generated machine code. Unfortunately, according to a recent performance study by Oh et al. [2012] this did not work out too well.

Oh *et al.* compared Dalvik against a HotSpot-derived VM called *phoneMe advanced*, an open source implementation of the JavaME (Micro Edition) platform. Both implementations were run on a tablet using an ARM Cortex-A8 CPU and 1 GB of RAM. Their study compared both implementations using EEMBC GrinderBench, a benchmark aimed at evaluating workloads for mobile devices. Their findings include:

- The Dalvik C interpreter is about 6% faster than the JVM's C interpreter. Dalvik also includes an interpreter written in assembly which is 60% faster than the JVM's C-based interpreter. It is not clear how an assembly based implementation of the JVM bytecode would fare in this comparison. Dalvik's bytecode contains instructions of many different sizes which suggests that decoding overhead is still rather high on average, which potentially negates the gains of using a register-based bytecode.

- The JVM's JIT compiler achieves an average speedup of about 11.7× over the the interpreter while the Dalvik's JIT compiler only achieves an average speedup of 4.0×. Since the interpreter speedup is very small this means that the code produced by the Dalvik JIT compiler is about three times slower than that produced by the JVM JIT compiler. The Dalvik VM generates ARM Thumb2 instruction whereas the reference JVM generates only regular ARM instructions. Regular ARM instructions are 4 bytes each; Thumb2 allows a mix of 2- and 4-byte instructions with the aim of reducing code size. This may degrade performance somewhat (approx. 6% according to ARM), but it does not explain the factor-of-three difference.

- Despite code size being an explicit design goal, the Dalvik VM generates about 10-20% more code than the JVM. This is despite using Thumb2 whose goal is to reduce code size. The JVM compiler also seems to be 4 times faster than the Dalvik compiler.

- They identify a number of reasons for the poor performance of the Dalvik JIT compiler. The main reason appears to be that traces are very short. Trace recording stops at any method call or branch. Dalvik's traces are therefore simply dynamic basic blocks. The reason for this choice may be to avoid excessive code duplication. It does, however, have several negative knock-on effects.

  - There is very little potential for optimisations. Each trace also needs to read and write most of its operands from memory instead of keeping them in a register. This causes extra memory traffic and extra generated code.
  - Since traces are so short every branch target is a potential trace head. This would require too many counters, so instead Dalvik uses a fixed-size table of counters where multiple branch targets may share the same counter (due to a hash collision). This may degrade the quality of selected traces.
  - Extra code necessary for linking traces together (called *chaining cells*) cause non-trivial overhead both in terms of code size as well as execution time.

Oh et al. [2012] try to address these shortcomings by allowing traces to span across branches (but not method calls). If combined with additional optimisations this gives a small performance improvement of about 5%. Most likely, however, the lack of interprocedural optimisations is the limiting factor. They also point out that in practice the poor performance of the Dalvik JIT compiler does not matter that much as most performance-sensitive applications (e.g., games, video) spend most of their time executing native code rather than Java code.

7.9 LuaJIT

LuaJIT is an implementation of the Lua programming language. Lua is a dynamically typed programming language designed specifically be easily embeddable into C or C++ programs. Version 1 of LuaJIT was a method-based compiler (written mostly in Lua itself), but version 2 uses a trace-based compiler and a faster interpreter written in assembly.

A lot of the implementation techniques of Lambdachine are derived from the LuaJIT codebase (Pall [2013]). There is no comprehensive documentation of the LuaJIT internals, but a short summary of techniques used is given in Pall [2009]. Important characteristics of the LuaJIT implementation are.

- The bytecode format is designed for easy decoding. For instance, opcodes and operands are aligned at byte boundaries and there are only two instruction formats. This avoids extra bit-shifting or masking operations for most instructions. The interpreter is written in assembly and uses direct threading. Parts of the following instruction are pre-decoded which helps offset the cost of the indirect branch on processors with out-of-order execution.

- The interpreter uses NaN-tagging. All primitive values use 8 bytes. Floating point numbers use all of the available 64-bits. There are about 2^{52} bit patterns that represent an invalid floating point number, a NaN (not a number). Modern hardware, however, only ever generates one possible bit pattern, so LuaJIT encodes other values in these 52 bits.

- Trace detection takes advantage of special bytecodes for loops. Default hotness threshold is 56 loop iterations.

- The compiler intermediate representation (IR) uses 16-bit references instead of pointers, which helps reduce the memory overhead of IR instruction. All IR instructions are 64 bits wide and are stored in a linear buffer. References to the output of one instructions are simply indexes into this buffer. IR references are statically partitioned into two ranges, one for constants and one for actual instructions. Instructions in the IR buffer are automatically in SSA form.

- The optimiser is based on a rule-matching engine. Before an IR instruction is emitted to the IR buffer it is sent through the rule-matching engine which might replace it with another instruction or eliminate it altogether.
- The machine code generator works backwards. Because the IR is always in SSA form this means that liveness analysis, register allocation and even dead code elimination can be integrated into the code generation pass.
- Snapshots capture the state of the abstract stack at guards. LuaJIT does not generate machine code to restore the concrete stack but instead uses a generic handler that uses the snapshot to restore the concrete stack. This reduces the amount of code generated at increased cost for trace exits. To offset this cost LuaJIT has a very low hotness threshold for compiling side traces (by default 10).
- Because Lua is a dynamically typed language there are many guards generated due to type check. To avoid generating snapshots for each of these guards, LuaJIT uses sparse snapshots. If a later guard fails it is safe to exit at an earlier snapshot provided that no side-effecting operation has occured in the meantime. Execution will re-enter the interpreter at an earlier point and redo some work that has already been done on the trace, but that is safe to do.

Lambdachine uses the same bytecode and IR design, but both have very different semantics. The register allocator and the parts of the code generator that emit machine code are basically the same as LuaJIT's. In 2012 allocation sinking was added to LuaJIT, but Lambdachine's allocation sinking pass was developed independently.

7.10 Compiler-only Virtual Machines

Some systems forego an interpreter and generate native code to execute the parts of the program for which no traces have been generated yet. DynamoRIO (Bruening [2004]; Bruening et al. [2003]) virtualizes x86 instructions on x86 hardware, so the baseline compiler just needs to decode the instructions and copy the code.

Both SPUR (Bebenita et al. [2010a]), a trace-based JIT compiler for Microsoft's Common Intermediate Language (CIL), and Maxpath (Bebenita et al. [2010b]), a trace-based JIT compiler for Java, use compilation for every execution mode. The stated reason for choosing a non-optimising compiler instead of an interpreter for baseline execution is performance. Neither reports any results comparing start-up times or memory overhead introduced by this decision.

The advantage of having a fast baseline execution mechanism is the possibility to delay optimisation decisions and collect more profiling data first. Maxpath uses this to collect multiple related traces into trace regions and only compiles the whole trace region once it has "matured". This reduces the chance of compiling code that is not actually hot. Maxpath also allows trace regions to contain inner control flow join points which allows it to optimise traces without biased branches as a whole.

Security concerns may further affect the performance overhead of compile-only systems. Many modern systems mark memory systems as either executable or writeable, but never both. This reduces the attack surface by making it difficult to directly write executable code into the target's address space. No single security mechanism can protect against all attacks, but having several counter measures in place is generally a good idea. A JIT compiler should mark memory as writable, then generate or update existing code, and finally mark the memory as execute-only. Each change of permissions is a system call and may take several hundreds of CPU cycles, not to mention that it may flush important caches (e.g., TLB) which will cause additional overhead later on.

Other projects using compilation only are V8 and Mozilla's Jägermonkey/Ionmonkey, all are JavaScript implementations.

7.11 Supercompilation

Supercompilation is a generalisation of partial evaluation. A supercompiler applies optimisation rules and inlines very aggressively. To bound the optimisation effort, a supercompiler must decide when to stop inlining and replace any recursive calls by calls to an appropriately specialised version of the recursive call.

Mitchell and Runciman [2008] introduced supercompilation for Haskell with Supero, which was later refined in Mitchell [2010]. Jonsson and Nordlander [2009] described a supercompiler for a strict functional language. Bolingbroke and Peyton Jones [2010] introduced a framework for expressing supercompilation in terms of four components: an evaluator, a splitter, a memoizer, and a termination checker. This helped sort out issues with earlier formulations which did not correctly handle all aspects of lazy evaluation. Optimisations that do not preserve sharing can lead to drastically reduced performance.

All of these systems still have issues with compilation times. The details of the termination heuristic are difficult to get right. If the termination checker stops compilation too soon, there may be too little room for optimisation, but if compilation is stopped too late, compilation time increases as well as code size.

The benchmarks chosen to evaluate supercompilers are often very small, so it remains to be seen how supercompilation can be made to work well with large input program.

Chapter 8

Conclusions and Future Work

This thesis investigated the suitability of trace-based just-in-time compilation for the lazy functional programming language Haskell. We described a prototype of such a compiler which supports a pure subset of Haskell and evaluated it using a collection of small benchmarks.

While the JIT compiler did well for micro benchmarks, the larger benchmarks exposed some issues in our prototype's trace selection strategy. Our trace selection heuristics are derived from standard heuristics described in the literature which have been designed for dynamic binary translators or object-oriented languages. These appear to work well enough if the hot code sections are comprised of simple `for` or `while` loops. In Haskell, however, loops are represented as tail-recursive functions. Some iteration constructs also are not tail-recursive and instead consume stack space proportional to the size of its inputs. Such functions cannot become part of a single trace, but need to be represented using at least two traces, one for the stack building phase and one for the stack deconstruction phase. This limits the potential for optimisations.

Call-by-need evaluation and the corresponding need to update a thunk with its value causes powerful loop optimisations to no longer become applicable. If the update can be omitted, the JIT compiler could perform loop optimisations that can produce similarly good results as stream fusion. The JIT compiler, however, has a more restricted view of the program and cannot normally omit the update. We proposed a dynamic sharing analysis that could provide this information and may also be helpful for the memory manager, but without an implementation it is difficult to judge whether this would be a net win. An improved static analysis could also be worthwhile.

While Haskell is a statically typed language, there is a fair amount of runtime "dynamism" due to laziness, higher-order functions and type classes. The JIT compiler removes this by specializing on concrete occurrences of the runtime instances. Unfortunately, without larger benchmarks it is difficult to evaluate how well Lambdachine's specialization strategy performs. In fact, it is quite difficult to map traces back to the Haskell source because the GHC front-end compiler already transforms the program through inlining and other static optimisations.

These observations suggest a number of areas of future work which promise to improve the insights and results of this thesis.

8.1 Feature Completeness

Our implementation does not yet support all features of Haskell 2010 never mind all of GHC's features. This limits the available set of benchmark programs. We suspect that the benchmarks used in this thesis do not represent the programming style that many modern high-performance Haskell programs use as many of them were written over 20 years ago (Partain [1993]). Modern benchmark suites such as Fibon[1] (Peixotto [2012]), however, rely on many Haskell standard libraries. To run these benchmarks, an implementatin must at least include support for arrays, the C foreign function interface, and floating point numbers.

Being able to run at least a larger selection of such benchmarks could help guide which aspects of trace selection and which optimisations are worth focusing on.

Supporting lightweight threads or parallel execution is an even bigger challenge. One possible way to support these is to integrate with the GHC runtime system which already supports a wide range of concurrency features. The downside is that the GHC runtime system is quite tightly integrated with the rest of GHC and would likely require substantial changes to become compatible with Lambdachine's low-level design. An alternative approach of building a new run-

[1]`https://github.com/dmpots/fibon`

time system would likely require even more effort, but might provide the opportunity to re-evaluate some design decisions made in GHC and could perhaps take advantage of the presence of a just-in-time compiler.

8.2 Trace Selection

There are a number of approaches that may improve trace selection. Static analysis may help to detect the equivalent of a `for` or `while` loop of imperative languages. The trace selector then may prefer those loops over other kinds of loops. A simple first step would be to use a lower hotness threshold for tail-recursive calls so that such loop-like control flow gets turned into traces first.

In order to better diagnose issues with trace selection it is very useful to have tools that simplify mapping traces back to the source program (or at least to the GHC Core program). Logging the execution trace of a full program can also be useful to perform offline analyses and produce a “perfect” set of traces (e.g., by optimising coverage and trace completion rate) for particular programs and compare them to the actual traces select by the trace selection heuristics. Such tools could later also be useful to users when debugging performance issues in user programs.

Currently, Lambdachine only allows at most one trace per bytecode address. If the trace is called from many different places (e.g., a commonly used function like `map`) then the trace will first examine what should be done by inspecting the shape of its arguments (via guards). Using a naïve scheme this degenerates to a linear sequence of comparisons, jumping from trace to side trace to side trace, etc. It may be necessary to move these checks outside of the trace. Instead of emitting these guards as code, each trace would be annotated with meta data describing the expected shape of its arguments (e.g., which info table). The interpreter would then perform these checks before entering a trace. Similarly, if trace recording reaches the beginning of another trace, the corresponding guards would become part of the calling trace.

This would make entering traces more expensive and multiple traces may start at the same trace anchor. This increases complexity, so careful consideration is required. Perhaps, with the right trace selection strategy such polymorphism is mostly eliminated by starting root traces from mostly-monomorphic consumers.

8.3 Avoiding Unnecessary Updates

Even with improved trace selection, Lambdachine's performance may still lag behind GHC's static optimisations because the need to preserve updates reduces the effectiveness of loop optimisations. There are two main questions worth persuing.

Firstly, is there an analysis that provides a net benefit? In Section 6.8.1 we suggested one such analysis which is mostly dynamic and annotates each object with a tag indicating whether it might be shared. Mainting this information could prove to introduce noticeable runtime overhead. The hope is that this overhead is outweighed by the additional optimisations that it enables. Sharing information could potentially also be exploited by the garbage collector. It may also be possible to reduce the amount of runtime data needed by employing a more sophisticated static analysis.

Secondly, relying on compiler optimisations can be a risky business. The co-recursive `Stream` data type (Coutts et al. [2007]) changes the program representation to make it easier for the compiler to optimise. Streams also decouple demand-driven evaluation from memoisation of the result which removes exactly the issue with updates. An alternative approach therefore would evaluate which program patterns are causing the issue and adapt libraries to be explicit about the kind of behaviour they want to rely on.

8.4 Portable and Checked Bytecode

Our current implementation only works on 64-bit x86 hardware. The compiler produces byte code which assumes a 64-bit architecture and would have to be adapted slightly to support 32-bit architectures. An even better approach would involve the design of a portable byte code that is translated into platform-specific byte code at load time (or at installation time). Such portable bytecode could also be annotated with type information which is verified before the program is executed.

Figure 8.1: In-memory representation of a list of integers.

8.5 Execution Profiling

One of the promises of a virtual machine with a JIT compiler is the potential for better runtime introspection by selectively recompiling parts of the program to include more instrumentation. Our front-end compiler cannot yet generate the necessary meta data (such as source location information) to make this possible. We imagine that such meta data could take eventually be used to rewrite the bytecode at runtime to turn extra profiling on or off.

8.6 Representation Optimisation

GHC requires a uniform representation of polymorphic data types. This has knock-on effects for the efficiency of data representation. In particular, polymorphic values must always be allocated on the heap. Because the elements of the type are polymorphic, the in-memory representation of a list of integers a list of integers will be stored in a rather inefficient manner, as shown in Figure 8.1.

If a Haskell program uses many objects of type `List Int` where the integer is known to be in normal form we may want to store this in a more efficient data format:

```
data List_SInt = Nil_SInt | Cons_SInt Int# List_SInt
```

If some of the integers may be unevaluated, we can still improve efficiency by using different constructors depending on whether the argument is in normal form or not.

```
data List_Int = Nil_Int
              | Cons_NFInt Int# List_Int
              | Cons_Int Int List_Int
```

The problem of introducing a new data represenation is that we can no longer use functions that worked on the old representation. This can lead to a huge amount of code duplication and hence is only worthwhile in a rare number of cases.

In many cases the more efficient representation can be generated automatically by the compiler. It would therefore be nice if the compiler could perform this transformation automatically if instructed to do so via, say, a compiler pragma. Unfortunately, it is very difficult for a static compiler to avoid huge code size increases. Every function that operates on lists and is called directly or indirectly by the program would have to be duplicated. Using whole-program analysis the compiler may be able to exclude some functions, but such an analysis is usually expensive.

An alternative approach would be to abstract the storage type into an interface. Figure 8.2 shows how a type class could abstract over constructing a list (`nil`, `cons`) and pattern matching on a list (`unpack1`). Note that this makes using lists *less* efficient. We have to rely on the compiler to generate specialised versions of the functions working on `Listlike` data. In particular, list construction requires an additional function call and, even worse, list destruction might actually allocate data(!) as demonstrated by the two instances in Figure 8.2.

Figure 8.3 shows an alternative definition of `unpack1` which would often be more efficient by utilising continuation-passing style. It avoids the allocation of a `Just` node and the nested pair by passing the extracted argument directly to the supplied continuation. The downside of this approach is that consumers of the data structure can no longer use `case` expressions, but must pass continuations.

GHC 7.6 manages to optimise both implementations to the same code where all class methods have been inlined and the code specialised. However, good performance is harder to guarantee for larger programs.

This technique is used for some high performance Haskell libraries like the `vector` package, but it comes with a fair amount of both conceptional and implementation overhead. Library functions must be (re-)written to work with the generalised types.

The Haskell package `adaptive-containers`[1] implements the same idea but uses a less safe API. Instead of `unpack1`, this package exposes three functions: the total function `null` to check if the list is empty and the two partial functions `head`

[1]`http://hackage.haskell.org/package/adaptive-containers`

```haskell
{-# LANGUAGE TypeFamilies, MagicHash, FlexibleContexts, BangPatterns #-}
import GHC.Prim
import GHC.Types

class Listlike l where
  type Elem l
  nil :: l
  cons :: Elem l -> l -> l
  unpack1 :: l -> Maybe (Elem l, l)

instance Listlike [a] where
  type Elem [a] = a
  nil = []; cons = (:)
  unpack1 []     = Nothing
  unpack1 (x:xs) = Just (x, xs)

data List_SInt = Nil_SInt | Cons_SInt Int# List_SInt

instance Listlike List_SInt where
  type Elem List_SInt = Int
  nil = Nil_SInt;  cons (I# n) l = Cons_SInt n l
  unpack1 Nil_SInt = Nothing
  unpack1 (Cons_SInt n l) = Just (I# n, l)

uptoLL :: (Listlike l, Num (Elem l), Ord (Elem l)) =>
          Elem l -> Elem l -> l
uptoLL x y | x < y     = cons x (uptoLL (x + 1) y)
           | otherwise = nil

sumLL :: (Listlike l, Num (Elem l)) => l -> Elem l
sumLL list = go 0 list
 where go !acc l = case unpack1 l of
                     Nothing -> acc
                     Just (x, xs) -> go (acc + x) xs

test = sumLL (uptoLL 1 10000 :: List_SInt)
```

Figure 8.2: Abstracting over the way lists are represented in memory.

```
class Listlike l where
  unpack1 :: l -> r -> (Elem l -> l -> r) -> r
  ...
instance Listlike [a] where
  unpack1 []     n c = n
  unpack1 (x:xs) n c = c x xs
  ...
instance Listlike List_SInt where
  unpack1 Nil_SInt        n c = n
  unpack1 (Cons_SInt x l) n c = c (I# x) l
  ...

sumLL :: (Listlike l, Num (Elem l)) => l -> Elem l
sumLL list = go 0 list
 where go !acc l = unpack1 l
                   acc  -- nil case
                   (λ x xs -> go (acc + x) xs) -- cons

toList :: Listlike l => l -> [Elem l]
toList l = unpack1 l [] (λ x xs -> x : toList xs)
```

Figure 8.3: Alternative implementation of `unpack1`.

and `tail` to extract the components of a non-empty list. This causes repeated pattern matching (as part of the implementation of each of these functions) if several of these functions need to be called, but this overhead is likely to be removed through inlining. The package also re-implements all the list functions based on this new interface.

For the (common) case where the optimised representation can be derived from the polymorphic representation we can do better by working at a lower level. Rather than requiring the user to turn a concrete data type into a generic interface, we can change the *implementation* of data construction and destruction. We change the byte code primitives for allocation to look up the correct constructor through an indirection, just like the `cons` and `nil` class methods would do. Similarly, we change the byte code primitives for matching on constructors and extracting fields to go through an indirection as well. Such indirection would naturally introduce performance overheads, but the JIT compiler will quickly generate specialised code for all commonly-encountered cases.

There are a number of advantages to moving the necessary indirections to the level of the bytecode semantics:

- No changes to the source code are necessary. This is especially useful if we would like to reuse existing libraries.
- Represenation changes can be done at runtime and will be transparent to the program. For example, we may choose to change the representation only for data with a certain amount of heap usage.

The downside is that we would pay the performance overhead even if we choose not to optimise the data representation.

8.7 Conclusions

This thesis has investigated the applicability of trace-based just-in-time compilation to Haskell programs. To this end we implemented Lambdamachine, a virtual machine for Haskell and a trace-based just-in-time compiler.

In our performance evaluation shows that this approach is indeed promising, but our current trace selection scheme works rather poorly. Our compiler is fast enough so that compilation overhead is not problematic, but even relatively small benchmarks cause large number of traces to be generated many of which with poor completion rate. Laziness further prevents some crucial allocation optimisations from becoming applicable. We suggested a dynamic analysis that could determine places where such heap allocations are valid, but it is unclear whether the overheads of such an analysis will outweigh its advantages.

Source Code Statistics

Table 1 lists the files and their (non-comment) number of source lines. The last column shows an estimate of the how much of the code has a one-to-one mapping to code from LuaJIT.

Source File	Description	SLOC	similarity
`loader.cc/.hh`	Bytecode loader	590 + 145	0%
`ir.hh/ir-inl.hh`	Trace IR definition	703 + 16	90%
`ir.cc`	Trace IR buffer operations	789	50%
`ir_fold.cc`	Trace IR folding engine	356	90%
`bytecode.hh`	Bytecode definition	167	70%
`bytecode.cc`	Bytecode printing	229	0%
`assembler.hh`	Code generator types and opcode definitions	474	90%
`assembler-debug.hh`	Tools for debugging the assembler	90	100%
`assembler.cc`	Code generator and register allocator implementation	1385	90%
`capability.hh`	Interpreter type definitions	90	0%
`capability.cc`	Interpreter implementation	839	0%
`objects.cc/.hh`	Heap object definitions and printing	196 + 165	0%
`jit.hh`	JIT compiler types and utilities	291	60%
`jit.cc`	Trace recorder implementation	1182	10%
`machinecode.cc`	Machine code allocation	135	95%
`memorymanager.cc/.hh`	Memory allocator and garbage collector	710 + 251	0%
`miscclosures.cc/.cc`	Closures used by the runtime system	338 + 53	0%
`thread.cc/.hh`	Lightweight thread definition	74 + 43	0%
`options.cc/.hh`	Program option parsing	163 + 49	0%
`main.cc`	The VM executable main function	184	0%
other utils		68	0%
other headers		459	5%
VM Total		10432	35%
`unittest.cc`	Various unit tests	2038	-
`lcc`	Bytecode compiler (written in Haskell)	5932	-
Total		18402	20%

Table 1: Source Code Statistics

Full Benchmark Data

Table 2 shows the detailed performance results for all of our benchmarks. Each benchmark was run in six configurations. The first three configurations of each benchmark is the execution time when compiled with GHC using three different optimisation levels: `O0` means no optimisations, `O2` is high optimisation level.[1] The other three configurations list the execution time when the benchmark is executed with Lambdachine. Since our front-end compiler (which translate Haskell programs into bytecode) uses GHC, we can control the static optimisation level via the same optimisation levels that GHC uses: `O0` thus means no static optimisations and only dynamic optimisations, `O2` means both static and dynamic optimisations.

The four columns are defined as follows:

- "MUT" is the time spent in the mutator (in seconds), i.e., excluding garbage collection. In the JIT execution mode this number includes the overhead of the trace recorder and those trace optimisations that are performed during trace recording.
- "GC" is the time spent in the garbage collector (in seconds).
- "JIT" is time spent in the JIT compiler (in seconds), but excluding the trace recorder. This number includes optimisations performed after trace recording completed and code generation.
- "Mem" is the total amount of memory allocated (in Gigabytes) by the program (most of which will have been garbage collected).

[1]GHC also supports optimisation level `O3`, its use is generally discouraged as it mainly increases compilation time and rarely better performance than `O2`.

The last column lists the number of traces that were compiled by the JIT compiler.

Table 2: Mutator performance for all benchmarks.

Benchmark	GHC/ JIT	Opt. Lvl.	MUT (s)	GC (s)	JIT (s)	Mem (GB)	Trace (No.)
SumFromTo1	GHC	`-O2`	2.31	0.12	-	14.63	-
		`-O1`	2.31	0.12	-	14.63	-
		`-O0`	4.21	0.15	-	17.81	-
	JIT	`-O2`	1.80	0.00	0.00	12.80	1
		`-O1`	1.80	0.00	0.00	12.80	1
		`-O0`	2.06	0.01	0.00	16.00	1
SumFromTo2	GHC	`-O2`	4.02	0.15	-	17.81	-
		`-O1`	4.02	0.15	-	17.81	-
		`-O0`	4.81	0.15	-	17.81	-
	JIT	`-O2`	2.32	0.01	0.00	16.00	2
		`-O1`	2.32	0.01	0.00	16.00	2
		`-O0`	2.30	0.01	0.00	16.00	2
SumSquare	GHC	`-O2`	2.45	3.57	-	16.46	-
		`-O1`	2.48	3.57	-	16.46	-
		`-O0`	3.44	0.16	-	18.07	-
	JIT	`-O2`	2.35	0.01	0.00	14.41	5
		`-O1`	2.13	0.01	0.00	14.41	4
		`-O0`	2.06	0.01	0.00	16.21	3
SumStream	GHC	`-O2`	0.24	0.00	-	< 0.01	-
		`-O1`	0.24	0.00	-	< 0.01	-
		`-O0`	3.88	0.10	-	11.20	-
	JIT	`-O2`	0.46	0.00	0.00	< 0.01	1
		`-O1`	0.46	0.00	0.00	< 0.01	1
		`-O0`	1.61	0.00	0.00	11.20	2
Tak	GHC	`-O2`	1.01	0.00	-	< 0.01	-
		`-O1`	1.01	0.00	-	< 0.01	-

Continued on next page

Table 2: Mutator performance for all benchmarks.

Benchmark	GHC/ JIT	Opt. Lvl.	MUT (s)	GC (s)	JIT (s)	Mem (GB)	Trace (No.)
		-O0	11.48	0.37	-	22.12	-
	JIT	-O2	0.84	0.00	0.00	< 0.01	6
		-O1	0.84	0.00	0.00	< 0.01	6
		-O0	6.82	0.14	0.00	22.06	83
WheelSieve2	GHC	-O2	0.34	2.06	-	0.67	-
		-O1	0.36	2.13	-	0.67	-
		-O0	0.65	1.64	-	1.24	-
	JIT	-O2	0.59	5.55	0.00	2.12	69
		-O1	0.59	5.58	0.00	2.12	53
		-O0	1.11	8.89	0.05[1]	3.60	95
Boyer	GHC	-O2	0.75	0.66	-	2.20	-
		-O1	0.75	0.66	-	2.20	-
		-O0	1.33	1.25	-	3.23	-
	JIT	-O2	0.72	0.34	0.02	1.98	700
		-O1	0.70	0.35	0.04	1.99	710
		-O0	1.88[2]	1.12	0.02	3.39	679
Constraints	GHC	-O2	0.88	2.31	-	2.16	-
		-O1	0.89	2.31	-	2.16	-
		-O0	1.89	1.49	-	4.25	-
	JIT	-O2	0.85	0.18	0.06	2.45	1616
		-O1	0.85	0.18	0.06	2.45	1617
		-O0	1.30	0.31	0.08	4.20	1830
Circsim	GHC	-O2	1.71	3.74	-	5.31	-
		-O1	1.72	3.73	-	5.31	-
		-O0	2.85	6.10	-	8.14	-
	JIT	-O2	3.43	9.96	0.06	11.33	849

Continued on next page

[1]Compilation time varies between 50 ms and 200 ms, presumably due to memory protection calls.

[2]Trace recorder aborts frequently due to an unimplemented feature.

Table 2: Mutator performance for all benchmarks.

Benchmark	GHC/ JIT	Opt. Lvl.	MUT (s)	GC (s)	JIT (s)	Mem (GB)	Trace (No.)
		`-O1`	3.43	9.92	0.06	11.33	850
		`-O0`	-[1]	-	-	-	-
Lambda	GHC	`-O2`	0.74	0.90	-	0.90	-
		`-O1`	0.74	0.91	-	0.90	-
		`-O0`	3.69	3.40	-	7.15	-
	JIT	`-O2`	0.97	1.25	0.09	1.65	892
		`-O1`	0.98	1.25	0.09	1.65	889
		`-O0`	-[2]	-	-	-	-

End of table

Table 3 lists the non-GC time (mutator time + JIT compiler time) for our benchmarks for different combinations of hotness thresholds. All programs were compiled with full static optimisations (`-O2`). The first number is the threshold for root traces, the second is the threshold for side traces.

[1]Failed due to an unimplemented feature in the trace recorder

[2]Failed due to a bug in the bytecode compiler.

Root/Side hotness threshold	SumFromTo1	SumFromTo2	SumSquare	SumStream	Tak	WheelSieve2	Boyer	Constraints	Circsim	Lambda
103/103	1.76	2.30	2.94	1.12	1.79	0.58	0.74	0.90	3.49	1.51
	1	2	8	2	6	35	511	812	725	44
103/53	1.76	2.30	2.22	1.12	1.75	0.59	0.74	0.85	3.44	1.51
	1	2	7	2	5	37	533	821	752	37
103/23	1.76	2.30	2.22	1.12	1.72	0.60	0.69	0.92	3.44	1.44
	1	2	5	2	4	44	635	1304	795	148
103/11	1.76	2.30	2.22	1.12	1.74	0.64	0.74	0.91	3.44	1.23
	1	2	5	2	4	52	628	1382	858	579
103/7	1.76	2.30	2.22	1.12	1.63	0.69	0.71	0.98	3.44	1.22
	1	2	5	2	4	56	699	2249	908	891
53/53	1.76	2.30	2.94	1.12	0.83	0.63	0.72	0.87	3.46	1.61
	1	2	8	2	7	43	519	801	770	45
53/23	1.76	2.30	2.22	1.12	0.84	0.58	0.72	0.86	3.46	1.65
	1	2	7	2	6	55	544	1009	776	51
53/11	1.76	2.30	2.22	1.12	0.83	0.58	0.72	0.87	3.50	1.12
	1	2	5	2	6	58	618	1352	849	583
53/7	1.76	2.30	2.22	1.12	0.83	0.58	0.74	0.91	3.50	1.06
	1	2	5	2	6	69	700	1616	776	892
23/23	1.76	2.30	2.94	1.12	1.67	0.67	0.81	0.89	3.47	1.40
	1	2	8	2	5	63	575	1043	801	215
23/11	1.76	2.30	2.22	1.12	1.60	0.67	0.76	0.91	3.45	1.18
	1	2	7	2	4	65	577	1355	862	582
23/7	1.76	2.30	2.22	1.12	1.76	0.64	0.79	0.90	3.48	1.07
	1	2	7	2	4	75	603	1592	889	892
11/11	1.76	2.30	2.94	1.12	1.73	0.63	0.73	0.92	3.50	1.16
	1	2	8	2	5	77	587	1308	860	590
11/7	1.76	2.30	2.22	1.12	1.72	0.64	0.72	0.94	3.49	1.07
	1	2	5	2	4	96	582	1622	929	897

Table 3: Hotness thresholds and their effect on performance. This table shows for each benchmark the mutator runtime (in seconds) and number of traces for different hotness thresholds. We distinguish hotness thresholds for root traces (R) and for side traces (S).

References

The v8 javascript engine. URL `https://developers.google.com/v8/`. 33

Håkan Ardö, Carl Friedrich Bolz, and Maciej FijaBkowski. Loop-aware optimizations in pypy's tracing jit. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 63–72, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384586. URL `http://doi.acm.org/10.1145/2384577.2384586`. 38, 142

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeño jvm. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 47–65, New York, NY, USA, 2000. ACM. ISBN 1-58113-200-X. doi: 10.1145/353171.353175. URL `http://doi.acm.org/10.1145/353171.353175`. 36

Lennart Augustsson. *Compiling lazy functional languages, part II.* PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987. 14, 155

John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35:97–113, June 2003. ISSN 03600300. doi: 10.1145/857076.857077. URL `http://portal.acm.org/citation.cfm?doid=857076.857077`. 155

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. Technical Report HPL-1999-78, HP Laboratories Cambridge, 1999. 46, 52, 54, 57, 158

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *PLDI'00: Proceedings of the ACM*

SIGPLAN 2000 conference on Programming Language Design and Implementation, pages 1–12. ACM, 2000. doi: 10.1145/349299.349303. URL `http://portal.acm.org/citation.cfm?id=349303`. 46, 47, 52, 54, 59, 158

Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium&#174;-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 191–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL `http://dl.acm.org/citation.cfm?id=956417.956550`. 33

Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: A Trace-based JIT Compiler for CIL. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 708–725, New York, NY, USA, 2010a. ACM. doi: 10.1145/1869459.1869517. 57, 168

Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 59–68, New York, NY, USA, 2010b. ACM. ISBN 978-1-4503-0269-2. doi: 10.1145/1852761.1852771. 47, 51, 56, 168

James R. Bell. Threaded Code. *Communications of the ACM*, 16(6):370–372, June 1973. ISSN 0001-0782. doi: 10.1145/362248.362270. URL `http://doi.acm.org/10.1145/362248.362270`. 163

Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. doi: 10.1109/CGO.2005.14. URL `http://dx.doi.org/10.1109/CGO.2005.14`. 163

Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18, 1988. 126

Maximilian Bolingbroke and Simon Peyton Jones. Supercompilation by Evaluation. In *Haskell '10: Proceedings of the third ACM Haskell symposium on Haskell*. ACM, 2010. 3, 169

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the Meta-Level: PyPy's Tracing JIT compiler. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009. doi: 10.1145/1565824.1565827. 159

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation Removal by Partial Evaluation in a Tracing JIT. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 43–52. ACM, 2011. doi: 10.1145/1929501.1929508. URL `http://codespeak.net/svn/pypy/extradoc/talk/pepm2011/bolz-allocation-removal.pdf`. 38

Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1999. 14, 79, 156

Urban Boquist and Thomas Johnsson. The GRIN Project: A Highly Optimising Back End for Lazy Functional Languages. In *Implementation of Functional Languages*, pages 58–84, 1996. 14, 79, 156

Dan Bornstein. Dalvik vm internals, 2008. URL `http://sites.google.com/site/io/dalvik-vm-internals`. 163

Derek L. Bruening. *Effient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, September 2004. 54, 55, 58, 59, 158, 167

Derek L. Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international* *symposium on Code generation and optimization: feedback-directed and run-time optimization*, CGO '03, pages 265–275. IEEE Computer Society, 2003. ISBN 0-7695-1913-X. 47, 158, 167

T. H. Brus, Marko C. J. D. van Eekelen, M. O. Van Leer, and Marinus J. Plasmeijer. Cleana language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, pages 364–384. Springer, 1987. 156

G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless g-machine. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, LFP '88, pages 244–258, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: 10.1145/62678.62717. URL `http://doi.acm.org/10.1145/62678.62717`. 14, 156

Kevin Casey, M. Anton Ertl, and David Gregg. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6), October 2007. ISSN 0164-0925. doi: 10.1145/1286821.1286828. URL `http://doi.acm.org/10.1145/1286821.1286828`. 163

C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11): 677–678, November 1970. ISSN 0001-0782. doi: 10.1145/362790.362798. URL `http://doi.acm.org/10.1145/362790.362798`. 110

Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 1–19, New York, NY, USA, 1999. ACM. ISBN 1-58113-238-7. doi: 10.1145/320384.320386. URL `http://doi.acm.org/10.1145/320384.320386`. 38

Cliff Click. *Combining Analyses, Combining Optimizations.* PhD thesis, Rice University, 1995. 158

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN* *international conference on Functional programming*, ICFP'07, pages 315–326. ACM, 2007. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291199. 4, 145, 173

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/115372.115320. URL `http://doi.acm.org/10.1145/115372.115320`. 100

Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The Case for Virtual Register Machines. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators*, pages 41–49, New York, New York, USA, 2003. ACM Press. ISBN 1581136552. doi: 10.1145/858570.858575. URL `http://portal.acm.org/citation.cfm?doid=858570.858575`. 81, 162

David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. *Proceedings of the 4th international symposium on Memory management - ISMM '04*, page 37, 2004. doi: 10.1145/1029873.1029879. URL `http://portal.acm.org/citation.cfm?doid=1029873.1029879`. 110

Robert T. Dimpsey, Rajiv Arora, and Kean Kuiper. Java server performance: A case study of building efficient, scalable jvms. *IBM Systems Journal*, 39(1): 151–174, 2000. doi: 10.1147/sj.391.0151. 52

Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, November 2000. ISSN 0362-1340. doi: 10.1145/356989.357008. URL `http://doi.acm.org/10.1145/356989.357008`. 46

ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* Geneva, Switzerland, 5.1 edition, June 2011. URL `http://www.ecma-international.org/publications/standards/Ecma-262.htm`. 33

ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. Geneva, Switzerland, 6th edition, June 2012. URL `http://www.ecma-international.org/publications/standards/Ecma-335.htm`. 161

Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0860-1. doi: 10.1145/2034675.2034690. URL `http://doi.acm.org/10.1145/2034675.2034690`. 12

M. Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5, 2003. URL `http://www.jilp.org/vol5/v5paper12.pdf`. 45

Joseph A. Fisher. *The optimization of horizontal microcode within and beyond basic blocks: an application of processor scheduling with resources*. PhD thesis, New York University, 1979. 160

Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *Computers, IEEE Transactions on*, C-30(7):478 –490, july 1981. ISSN 0018-9340. doi: 10.1109/TC.1981.1675827. 160

Etienne Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, December 2002. 163

Etienne Gagnon and Laurie Hendren. Effective Inline-Threaded Interpretation of Java Bytecode Using Preparation Sequences. In Grel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 170–184. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-00904-7. doi: 10.1007/3-540-36579-6_13. URL `http://dx.doi.org/10.1007/3-540-36579-6_13`. 163

Andreas Gal and Michael Franz. Incremental Dynamic Code Generation with Trace Trees. Technical Report ICS-TR-06-16, University of California, Irvine, 2006. URL `www.ics.uci.edu/~franz/Site/pubs-pdf/ICS-TR-06-16.pdf`. 39, 56, 159

Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In *VEE'06*, 2006. 159

Andreas Gal, Jason Orendorff, Jesse Ruderman, Edwin Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, Michael Franz, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, and Boris Zbarsky. Trace-based Just-in-time type Specialization for Dynamic Languages. In *PLDI'09*, May 2009. doi: 10.1145/1543135.1542528. URL `http://portal.acm.org/citation.cfm?doid=1543135.1542528`. 33, 52, 53, 56, 159

The GHC Team. The glorious glasgow haskell compilation system user's guide, version 7.0.4, 2011. URL `http://www.haskell.org/ghc/docs/7.0.4/users_guide.pdf`. 95, 156

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. *Proceedings of the conference on Functional programming languages and computer architecture - FPCA '93*, pages 223–232, 1993. doi: 10.1145/165180.165214. URL `http://portal.acm.org/citation.cfm?doid=165180.165214`. 4

Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. A generic usage analysis with subeffect qualifiers. In *ICFP'07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 235–246. ACM, 2007. 148

Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, and Toshio Nakatani. Improving the Performance of Trace-based Systems by False Loop Filtering. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 405–418. ACM, 2011. ISBN 978-1-4503-0266-1. doi: http://doi.acm.org/10.1145/1950365.1950412. URL `https://researcher.ibm.com/researcher/files/us-pengwu/asplos160-hayashizaki.pdf`. 49, 159

David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving Region Selection in Dynamic Optimization Systems. In *38th Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO'05), pages 141–154. IEEE, 2005. ISBN 0-7695-2440-0. doi: 10.1109/MICRO.2005.22. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1540955`. 46, 47, 51

Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* PhD thesis, Stanford University, 1994. 158

Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, 1991. Springer. doi: 10.1007/BFb0057013. 158

Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143114. URL `http://doi.acm.org/10.1145/143095.143114`. 158

Andrei Homescu and Alex Şuhan. Happyjit: a tracing jit compiler for php. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 25–36. ACM, 2011. doi: 10.1145/2047849.2047854. 159

John Hughes. Why functional programming matters. *The Computer Journal*, 32 (2):98–107, 1989. doi: 10.1093/comjnl/32.2.98. 2

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.1 Reference Manual.* Lua.Org, 2006. ISBN 8590379833. 77

Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler, 2011. URL `https://researcher.ibm.com/researcher/files/us-pengwu/CGO2011_TraceJIT.pdf`. 159

Thomas Johnsson. *Compiling lazy functional languages.* PhD thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987. 14, 155

Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 1st edition, 2011. ISBN 978-1420082791. 111

Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* CRC Press, 2012. ISBN 978-1-4200-8279-1. 110

Peter A. Jonsson and Johan Nordlander. Positive supercompilation for a higher order call-by-value language. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 277–288, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480916. URL `http://doi.acm.org/10.1145/1480881.1480916`. 169

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot^{TM}Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):7:1–7:32, May 2008. ISSN 1544-3566. doi: 10.1145/1369396.1370017. URL `http://doi.acm.org/10.1145/1369396.1370017`. 52, 60, 158, 164

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86, Palo Alto, California, Mar 2004. doi: 10.1109/CGO.2004.1281665. 40

Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201432943. 33, 161, 163

P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7:51–142, 1993. ISSN 0920-8542. URL `http://dx.doi.org/10.1007/BF01205182`. 10.1007/BF01205182. 160

Kein-Hong Man. *A No-Frills Introduction to Lua 5.1 VM Instructions.* 2006. URL `http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf`. 162

Simon Marlow and Simon Peyton Jones. Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages. In *ICFP '04: The 2004 International Conference on Functional Programming*, pages 4–15, 2004. doi: 10.1145/1016848.1016856. 26, 28, 90, 156

Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster Laziness Using Dynamic Pointer Tagging. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 277–288, 2007. 23, 24, 25, 156

Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore Haskell. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 65–78, 2009. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596563. URL `http://portal.acm.org/citation.cfm?doid=1596550.1596563`. 156

Neil Mitchell. Rethinking Supercompilation. In *ICFP'10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 309–320. ACM, 2010. 3, 169

Neil Mitchell and Colin Runciman. A supercompiler for core haskell. In Olaf Chitil, Zoltn Horvth, and Viktria Zsk, editors, *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 147–164. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-85372-5. doi: 10.1007/978-3-540-85373-2_9. URL `http://dx.doi.org/10.1007/978-3-540-85373-2_9`. 168

Matthew Naylor and Colin Runciman. The reduceron: Widening the von neumann bottleneck for graph reduction using an fpga. In Olaf Chitil, Zoltn Horvth, and Viktria Zsk, editors, *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 129–146. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-85372-5. doi: 10.1007/978-3-540-85373-2_8. URL `http://dx.doi.org/10.1007/978-3-540-85373-2_8`. 14, 157

Matthew Naylor and Colin Runciman. The Reduceron Reconfigured. In *ICFP'10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 75–86. ACM, 2010. 14, 157

Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon. Evaluation of Android Dalvik Virtual Machine. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 115–124, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1688-0. doi: 10.1145/2388936.2388956. URL `http://doi.acm.org/10.1145/2388936.2388956`. 164, 165

Mike Pall. LuaJIT 2.0 intellectual property disclosure and research opportunities. Lua-l mailing list message, 2009. URL `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`. 47, 55, 77, 166

Mike Pall. LuaJIT 2, 2013. URL `http://luajit.org/`. 9, 41, 56, 77, 162, 166

Will Partain. The nofib benchmark suite of haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993. 127, 171

David M. Peixotto. *Low-Level Haskell Code: Measurements and Optimization Techniques*. PhD thesis, Rice University, 2012. 159, 171

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. doi: 10.2277/0521826144. URL `http://haskell.org/onlinereport`. 139

Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2:127–202, 1992. 14, 80, 156

Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993. ISBN 0-201-41663-

8. URL http://wiki.clean.cs.ru.nl/Functional_Programming_and_Parallel_Graph_Rewriting. 14, 156

Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, September 1999. ISSN 0164-0925. doi: 10.1145/330249.330250. URL http://doi.acm.org/10.1145/330249.330250. 120

Armin Rigo and Samuele Pedroni. PyPy's Approach to Virtual Machine Construction. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006. doi: 10.1145/1176617.1176753. 159

Colin Runciman. Lazy wheel sieves and spirals of primes. *Journal of Functional Programming*, 7(2):219–225, 1997. 128

Konstantinos Sagonas and Erik Stenman. Experimental evaluation and improvements to linear scan register allocation. *Softw. Pract. Exper.*, 33, 2003. doi: 10.1002/spe.533. 120

Thomas Schilling. Challenges for a trace-based just-in-time compiler for haskell. In *Proceedings of the 23rd International Conference on Implementation and Application of Functional Languages*, IFL'11, pages 51–68. Springer-Verlag, 2012. ISBN 978-3-642-34406-0. doi: 10.1007/978-3-642-34407-7_4. URL http://dx.doi.org/10.1007/978-3-642-34407-7_4. 9

Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan Ricci, and Samuel Z. Guyer. new scala() instance of java: a comparison of the memory behaviour of java and scala programs. In *Proceedings of the 2012 international symposium on Memory Management*, ISMM '12, pages 97–108, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1350-6. doi: 10.1145/2258996.2259010. URL http://doi.acm.org/10.1145/2258996.2259010. 110

Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *Proceedings of the 23rd*

ACM SIGPLAN conference on Object-oriented programming systems languages and applications, OOPSLA '08, pages 127–142, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: 10.1145/1449764.1449775. URL `http://doi.acm.org/10.1145/1449764.1449775`. 158

Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual Machine Showdown: Stack Versus Registers. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 153–163. ACM, 2005. ISBN 1-59593-047-7. doi: 10.1145/1064979.1065001. URL `https://www.usenix.org/events/vee05/full_papers/p153-yunhe.pdf`. 81

Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):2:1–2:36, January 2008. ISSN 1544-3566. doi: 10.1145/1328195.1328197. URL `http://doi.acm.org/10.1145/1328195.1328197`. 162

James Edward Smith and Ravi Nair. *Virtual Machines: Versatile Platforms For Systems And Processes*. Elsevier, 2005. ISBN 1-55860-910-5. 34

Sunil Soman and Chandra Krintz. Efficient and general on-stack replacement for aggressive program specialization. In *International Conference on Programming Languages and Compilers (PLC), Las Vegas, NV*, 2006. 40

David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. In *Proceedings of the 2012 symposium on Haskell symposium*, Haskell '12, pages 137–148, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364524. URL `http://doi.acm.org/10.1145/2364506.2364524`. 140

David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 1–11, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. doi: 10.1145/224164.224168. URL `http://doi.acm.org/10.1145/224164.224168`. 150

Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990. 3

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16'th Symposium on Principles of Programming Languages*. ACM Press, 1989. 28

Keith Wansbrough and Simon Peyton Jones. Simple Usage Polymorphism. In *The Third ACM SIGPLAN Workshop on Types in Compilation*, 2000. 146

Peng Wu, Hiroshige Hayashizaki, Hiroshi Inoue, and Toshio Nakatani. Reducing trace selection footprint for large-scale java applications without performance loss. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 789–804. ACM, 2011. doi: 10.1145/2048066.2048127. 54, 159