Trace-based Just-in-time Compilation for Haskell

Thomas Schilling (ts319@kent.ac.uk)

University of Kent, Canterbury, Kent, UK

1 Introduction

Every programmer knows that many programming language abstractions come at a performance cost. For instance, abstracting away from how data is laid out in memory can make it harder to take advantage of data locality. Nevertheless, for many applications the benefits in speed of development or ease of reasoning about a program may outweigh the costs.

Haskell, a statically typed functional programming language, emphasizes program correctness and concise program descriptions over best absolute performance. State-of-the-art Haskell implementations can still produce remarkably efficient programs through the use of sophisticated static program analyses and transformations. Unfortunately, static compilation has its drawbacks. My work investigates how optimizations performed at runtime can remove or mitigate these drawbacks.

2 Static Optimization and Haskell

For the purposes of this work, Haskell's most interesting characteristics are:

- *Lazy evaluation.* Function arguments in Haskell are only evaluated when required by the computation (and no sooner). In the function call`(f x (g y))`[1] the expression `g y` is not evaluated before passing control to `f`. Instead a heap object called a *thunk* is allocated and a reference to that heap object is passed as the second argument to `f`. If `f` requires the value of its second argument it *evaluates* the thunk.

  Evaluation of the thunk causes it to be reduced to *weak head normal form.* This means that an object is reduced until its outermost shape (its constructor) is visible, for example, whether a list object is nil or a cons; object fields may still be thunks.

  Lazy evaluation is different from call-by-name in that a thunk is *updated* with its value after it is evaluated for the first time. If the same thunk is evaluated a second time this value is returned immediately.

[1]In Haskell function application is written using juxtaposition, hence $f(x, g(y))$ is written as `(f x (g y))`.

- *Curried function definitions.* Functions are first class values and can be passed as arguments and return values. Functions may also be applied to fewer or more arguments than the function expects, which is called *partial application* and *overapplication*, respectively. The runtime system must handle either case if they cannot be ruled out statically [8].

- *Type classes.* Haskell uses type classes [12] to support overloaded functions. They are conceptually quite similar to Java interfaces and are typically implemented using indirect function calls. A source expression `(plus x y)` where `plus` is an overloaded function of type class `C` is internally rewritten to `(plus dict x y)`. The `dict` argument is a record of functions definitions for all overloaded functions of the class `C`. The `plus` function merely looks up its implementation in this record and calls it with the two arguments `x` and `y`. Haskell implements many common operations such as comparision and addition are implemented using type classes.

Haskell has traditionally been a stronghold of static optimization. The Glasgow Haskell Compiler (GHC)[2] has a sophisticated inliner [10] to eliminate function call overhead and expose further optimizations. Library authors can specify rewrite rules to be applied during compilation [11]. This is used in particular to implement *deforestation*, the removal of intermediate data structures [4, 7].

There are certain downsides to static optimization. Optimizations can be fragile and sensitive to small changes in the input program (in particular if rewrite rules are involved). Inliner behavior must often be fine-tuned to avoid too much increase in code size while not missing optimizations in hot spots. Rewrite rules must usually be re-implemented from scratch for each data structure. Finally, programs (and all dependencies) must be recompiled to enable certain forms of profiling or debugging.

Trace-based Just-in-time (TJIT) compilation [1] has been used successfully to optimize statically typed languages such as Java [5] and C#/CIL [2] as well as dynamically typed languages such as JavaScript [6], Lua [9] and Python [3].

For my work, I implemented a prototype TJIT compiler which can successfully optimize programs where the static compiler has missed opportunities. This work focuses on performance, but the presence of a JIT compiler introduces other potential benefits such as the possibility of a platform-independent bytecode format, or to avoid the need to recompile programs for different execution modes. My prototype uses GHC as a front-end, so it is indeed possible to take advantage of *both* static and dynamic optimizations.

3 Trace-based Just-in-time Compilation

Traditional JIT compilers use basic blocks or single functions as a unit of compilation. Using a form of runtime profiling, hot functions are identified and then compiled and optimized using standard compilation techniques on the control

[2]`http://haskell.org/ghc/`

flow graph of the full function (and possibly inlined functions). The compiler typically has to traverse the program multimiple times which slows down compilation speed. If a function is inlined, its whole body is inlined increasing the problem size further. In practice, however, only a few parts of a function are actually hot and the compiler may spend a lot of time optimizing code that is executed rarely or never.

Trace-based compilation, on the other hand, compiles only straight-line code—a trace—with no control flow join points except that the last instruction might loop back to the beginning. Iterative data-flow analyses are no longer necessary and compilation can be implemented as a single forward pass and a single backwards pass (combined with, or followed by, machine code generation). A trace follows the execution of the program and is not limited to a single function. Traces usually correspond to loops in the source program which tends to result in greater scope for optimizations and removal of redundancies. E. g., if an object is allocated in one iteration of the loop only to be destructed and discarded in the next iteration, then the allocation can often be removed altogether.

A trace only corresponds to a single path through the loop. If the original code contains a conditional branch, the recorded trace will contain a *guard*. A guard verifies that execution is allowed to continue on the trace; if the guard fails execution leaves the current trace and continues in the interpreter or another trace. If a guard fails frequently it usually becomes the start of another trace.

Figure 1 gives an overview of the main phases of a trace-based JIT.

1. Execution starts in the interpreter. Only special instructions can cause the start of a new trace (e.g., function calls). Whenever the interpreter executes any of these instructions a hot counter is incremented. If the hot counter reaches a predefined threshold, the instruction is considered hot and the next execution phase begins.

2. Execution continues in trace recording mode. The interpreter still executes bytecode instructions but additionally emits intermediate code to the trace recorder. Recording continues until either of the following conditions occurs: (1) loop is found, (2) the start of an existing trace is found, or (3) an abort condition is triggered, e.g., an exception is thrown or the recorded trace has become too long.

3. If recording finished successfully, the recorded intermediate code is now optimized and translated into machine code. The machine code is then placed into the *fragment cache* and execution continues in machine code.

4. Eventually, a guard fails and execution leaves the fragment cache and falls back to the interpreter. Execution continues in the interpreter until an existing trace is encountered or another instruction becomes hot.

Since traces follow the actual execution paths of the program, the generated code is specialized to the program's behavior at runtime. In Haskell, a lazy

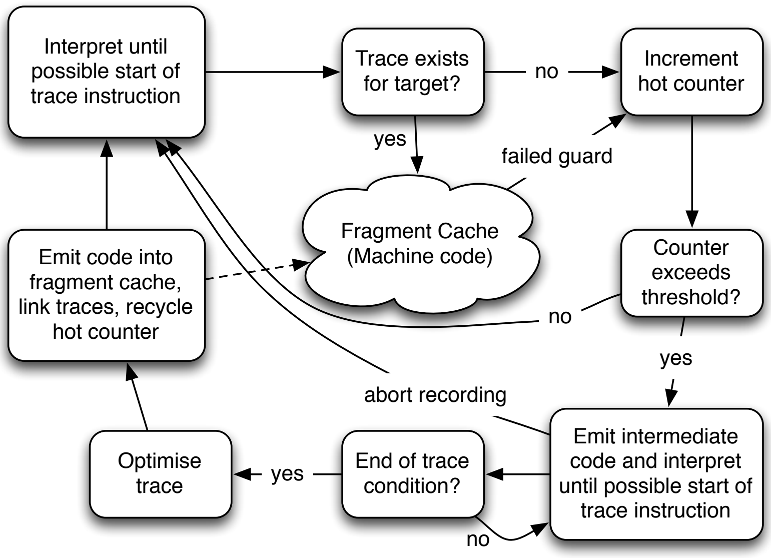
Interpret until possible start of trace instruction
Trace exists for target?
no
Increment hot counter
yes
failed guard
Fragment Cache (Machine code)
Emit code into fragment cache, link traces, recycle hot counter
Counter exceeds threshold?
no
yes
abort recording
Optimise trace
yes
End of trace condition?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

Figure 1: Overview of trace-based just-in-time compilation.

list producer will end up on the same trace as the list consumer and both are optimised together. This specialization also helps remove the overheads of type classes and function invokation.

4 Implementation Results

Our prototype implementation, called Lambdachine[3], is a custom virtual machine that derives many implementation decisions and some code from the open-source LuaJIT project[4]. This helped obtain a high-performance TJIT compiler with acceptable implementation effort.

Figure 2 shows performance for six micro-benchmarks (left) and four medium-sized benchmarks (right four groups of bars). Each bar represents the runtime relative to the fully statically-optimized program. The other bars are combinations of programs with static optimisations turned off (O0) or on (O2) and with and without JIT compilation.

For most micro-benchmarks our TJIT compiler does indeed improve performance. For SumStream it is missing an important optimisation which should result in equivalent performance to the statically optimised version.

For the medium-sized benchmarks our TJIT compiler cannot improve over the static optimizer and can be up to twice as slow. Further investigation

[3]`https://github.com/nominolo/lambdachine`
[4]`http://luajit.org/`

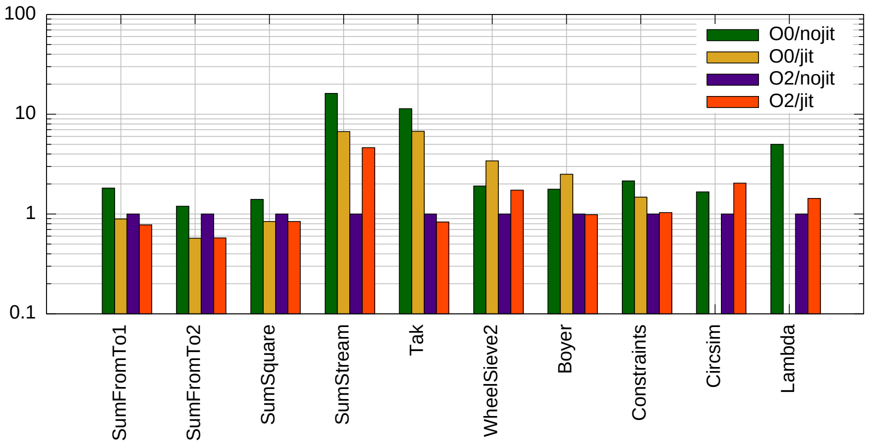

Figure 2: Benchmark runtime normalized to full static optimization (O2/nojit). Numbers include JIT-compilation time, but exclude garbage collection time.

shows that this is most likely due to poor trace selection. It turns out that the execution semantics of Haskell are sufficiently different from other languages for which TJIT compilers have been developed that techniques developed for these environments do not work well for Haskell. We are currently exploring trace selection strategies that may produce better traces for Haskell.

References

[1] 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.

[2] 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, 2010. ACM.

[3] 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.

[4] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIG-*

PLAN International Conference on Functional Programming, ICFP '07, pages 315–326. ACM, 2007.

[5] Andreas Gal and Michael Franz. Incremental Dynamic Code Generation with Trace Trees. Technical Report ICS-TR-06-16, University of California, Irvine, 2006.

[6] 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 *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478. ACM, 2009.

[7] Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture - FPCA '93*, FPCA '93, pages 223–232. ACM, 1993.

[8] Simon Marlow and Simon Peyton Jones. Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, ICFP '04, pages 4–15. ACM, 2004.

[9] Mike Pall. LuaJIT 2.

[10] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*, 12(5):393–434, July 2002.

[11] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell'01*, pages 203–233, 2001.

[12] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76. ACM, 1989.