Meta-tracing makes a fast Racket

Carl Friedrich Bolz
Software Development Team
King's College London
cfbolz@gmx.de

Tobias Pape
Hasso-Plattner-Institut Potsdam
Tobias.Pape@hpi.uni-potsdam.de

Jeremy Siek
Indiana University
jsiek@indiana.edu

Sam Tobin-Hochstadt
Indiana University
samth@cs.indiana.edu

Abstract

Tracing just-in-time (JIT) compilers record and optimize the instruction sequences they observe at runtime. With some modifications, a tracing JIT can perform well even when the executed program is itself an interpreter, an approach called meta-tracing. The advantage of meta-tracing is that it separates the concern of JIT compilation from language implementation, enabling the same JIT compiler to be used with many different languages. The RPython meta-tracing JIT compiler has enabled the efficient interpretation of several dynamic languages including Python (PyPy), Prolog, and Smalltalk. In this paper we present initial findings in applying the RPython JIT to Racket. Racket comes from the Scheme family of programming languages for which there are mature static optimizing compilers. We present the result of spending just a couple person-months implementing and tuning an implementation of Racket written in RPython. The results are promising, with a geometric mean equal to Racket's performance and within a factor of 2 slower than Gambit and Larceny on a collection of standard Scheme benchmarks. The results on individual benchmarks vary widely. On the positive side, our interpreter is sometimes up to two to six times faster than Gambit, an order of magnitude faster than Larceny, and two orders of magnitude faster than the Racket JIT compiler when making heavy use of continuations. On the negative side, our interpreter is sometimes three times slower than Racket, nine times slower than Gambit, and five times slower than Larceny.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers; D.3.2 [*Software Engineering*]: Applicative (functional) languages

General Terms Languages, Design, Performance

Keywords Meta-tracing, JIT, Scheme, Racket, CEK

1. Introduction

Just-in-time (JIT) compilation has been applied to a wide variety of languages, with early examples including Lisp, APL, Basic, Fortran, Smalltalk, and Self (Aycock 2003). These days, JIT compilation is mainstream; it is responsible for running both server-side Java applications (Paleczny et al. 2001) and client-side JavaScript applications in web browsers (Hölttä 2013).

Mitchell (1970) observed that one could obtain an instruction sequence from an interpreter by recording its actions. The interpreter can then jump to this instruction sequence, this *trace*, when it returns to interpret the same part of the program. For if-then-else statements, there is a trace for only one branch. For the other branch, Mitchell suggests reinvoking the interpreter. Bala et al. (2000) applied tracing JIT compilation to optimize native assembly code and Gal et al. (2006) showed that tracing JIT compilers can efficiently execute object-oriented programs, which feature control flow that is highly dynamic and data-dependent.

Developing a just-in-time compiler is traditionally a complex undertaking. However, Bolz et al. (2009) developed *meta-tracing*, an approach that can significantly reduce the development cost for tracing JIT compilers. With meta-tracing, the language implementer does not have to implement a JIT compiler for their language. Instead they simply write an interpreter for their language in a special implementation language called RPython. Then some annotations (hints) can be added to their interpreter, and RPython's tracing JIT compiler will be automatically inserted into the language implementation. Rigo and Pedroni (2006) and Bolz and Tratt (2013) describe an efficient meta-tracing implementation of Python, called PyPy.

Meta-tracing has not previously been applied to a programming language in the Scheme family. Scheme is interesting for two reasons: on the one hand, Scheme presents a number of challenges, such as continuations, guaranteed tail call optimization, heavy use of higher-order programming and multiple return values. On the other hand, Scheme has traditionally had very good ahead-of-time (AOT) compilers that do static analysis of Scheme code and optimize it heavily, such as Gambit (Feeley 2014). It is therefore an open question whether a tracing JIT compiler, and more so a non-language specific meta-tracing compiler, can compete.

In this paper we present *Pycket*, an implementation of the Racket language (Flatt and PLT 2010) in RPython. Pycket was written in a relatively straightforward way, closely following the control, environment, and continuation (CEK) abstract machine (Felleisen and Friedman 1987). We chose to base our interpreter on the CEK-machine for two reasons: first, the CEK-machine embodies perhaps the most straightforward way to implement a language with first-class continuations. Second, we wanted to see if meta-tracing could produce good results on an interpreter written at a higher-level than the usual bytecode interpreters.

In this paper, we make the following contributions.

- We show that meta-tracing makes a CEK-based interpreter for Racket competitive with the existing AOT and JIT compilers.
- We explain the transformations we applied to Pycket to make it amenable to optimization by meta-tracing.
- We report the performance of Pycket on a standard set of Scheme benchmarks, comparing it to Larceny (Clinger and

Hansen 1994), Gambit (Feeley 2014), and the Racket JIT compiler.

The paper is structured as follows: in section 2 we review some of the salient features of Scheme and Racket and give some background on RPython and its tracing JIT compiler. In section 3 we describe the Pycket interpreter and the transformations that were needed to make it amenable to optimization by the RPython tracing JIT compiler. We evaluate the performance of Pycket in section 5 and discuss related work in section 6. Section 7 concludes with some future directions.

2. Background

Pycket is based on RPython and Racket; we give brief introductions to both in this section.

2.1 RPython and Meta-Tracing

RPython is a statically typed subset of Python that can be compiled into efficient C code. The types do not have to be given in the source code but are inferred. It was designed to be an implementation language for interpreters for (dynamic) languages.

The feature that makes RPython interesting is its support for meta-tracing. When compiling an interpreter written in RPython to C, optionally a tracing JIT compiler can be inserted into the generated C code. This requires some source code annotations by the interpreter author to make it work (Bolz et al. 2009, 2011b). The inserted tracing JIT compiler produces linear sequences of machine code by recording the operations the interpreter executed while running a specific piece of the user's program. The recorded sequence of operations (called a *trace*) is optimized and then machine code is emitted. Traces are only produced for parts of the program that have been executed more than a specified threshold.

Because the produced machine code is linear, it needs to encode potential control flow branches in a special way. This is done with guards, an instruction that encodes the conditions that have to be true to stay on the trace. If a guard fails, execution falls back to the interpreter. If it fails often enough, a trace for that side path is produced as well. Quite often the produced traces are actually loops, meaning that they end with jumps to their own beginning.

2.2 Scheme and Racket

Scheme (Sussman and Steele Jr. 1975; Sperber et al. 2010) is a small functional language based around the imperative call-by-value λ-calculus with a core set of additional data structures. Starting with Steele (1978), Scheme also has a long history of optimizing compilers, often focusing on difficult-to-compile features such as first-class functions (in the 1970s when Scheme was invented) and first-class continuations (subsequently).

Racket (Flatt and PLT 2010) is a mature functional language and implementation, derived from Scheme with significant extensions. For our purposes, we focus mainly on areas where it overlaps with Scheme, and on its performance characteristics.

Notably for our purposes, Racket extensively uses macros and other forms of syntax extension; prior to compilation all syntax extensions are expanded into a small set of core language forms. Pycket makes use of this pass by invoking Racket itself to perform this expansion, and then implements only the dozen core forms.

The current Racket implementation features both an AOT compiler and a JIT compiler. The AOT compiler takes the core forms mentioned above and produces a stack-based bytecode. At runtime, upon first invocation of a function, the bytecode of the function is compiled to machine instructions by the JIT compiler.

The implementation strategy of Racket places most optimization burden on the AOT compiler to bytecode. This compiler performs a number of transformations important for functional languages such as lambda-lifting, as well as standard transformations such as constant propagation and dead code elimination. In particular, significant inlining is performed in this pass, including inlining of recursive functions, resulting in a kind of loop unrolling.

Because the Racket JIT compiler is invoked on the first call to a function, it can only take advantage of dynamic information present at that time. Therefore, while the JIT compiler does make use of information such as types of constant references, it does not perform the kind of dynamic optimizations often associated with JIT compilers for dynamic languages. In particular, it does not feed back type information collected at runtime about variables into the compiler (Hölzle and Ungar 1994).

3. The Implementation of Pycket

Pycket is implemented directly as an interpreter for the CEK abstract machine (Felleisen and Friedman 1987), with only minor modifications. The CEK-machine is a well-known abstract machine that makes environments and continuations explicit in the representation and it represents control using abstract syntax trees (ASTs). States within the CEK-machine are triples of an expression to be evaluated, the environment of that expression which stores bindings of names to values, and a continuation object that describes what to do after the expression has been evaluated. The machine is a set of rules describing transitions between these states. We implement mutation using the Python heap, instead of an explicit heap as in the CESK-machine.

Following the CEK-machine has a number of advantages. First, it makes supporting difficult-to-implement Racket features such as `call/cc`, proper tail calls, and multiple return values straightforward. Second, using the CEK-machine as a starting point is also interesting in that, implemented naively, it is not a particularly efficient way to execute Scheme-like languages. It is therefore an excellent way to test the hypothesis that RPython's meta-tracing can generate an efficient implementation from such a high-level interpreter.[1]

Listing 1 shows the main loop of the Pycket interpreter. The local variables represent the current expression to be evaluated (`ast`), the current environment (`env`), and the continuation (`cont`). In every iteration of the loop, one transition rule of the CEK-machine is applied, by calling the method `interpret` on the current `ast` (line 4), which returns a new triple `ast, env, cont`. For transitions that produce values, the `interpret` method decides what to do next by invoking the `plug_reduce` method on the current continuation `cont`. If the computation is finished, a special `Done` exception is thrown, which encapsulates the return value.

The main loop has two hints to RPython's JIT compiler generation machinery, on lines 3 and 5–6. These hints are discussed further in section 4, but they do not affect the semantics of the program.

```
1 try:
2   while True:
3     jitdriver.jit_merge_point()
4     ast, env, cont = ast.interpret(env, cont)
5     if ast.should_enter:
6       jitdriver.can_enter_jit()
7 except Done, e:
8   return e.values
```

Listing 1. Interpreter main loop

[1] While a direct interpreter on the AST would be even more high-level, it would make it harder to implement continuations, and would require proper tail calls from the underlying language, which RPython does not provide.

3.1 Expressions

The AST is a representation of the program being interpreted. The AST has to represent only the core forms, with the remainder handled by Racket's macro expander; these forms are lambda abstractions, function application, quotes, variables, let bindings, **`if`**, **`letrec`**, **`begin`**, and **`set!`**. As an example, the AST class representing if-expressions is seen in listing 2.

An if-expression references a test, a then-expression and else-expression, all of which are themselves ASTs. To interpret an if-expression, first the test is evaluated and the value resulting from that evaluation is compared to `#f`. Depending on whether that is true or false, the then- or the else-expression is returned as the new AST to be interpreted. The environment and continuation remain the same.

```
class If(AST):
 # constructor omitted
 def interpret(self, env, cont):
  val = self.tst.interpret(env)
  if val is values.false:
   return self.els, env, cont
  else:
   return self.thn, env, cont
```

Listing 2. If AST node

3.2 Continuations

Continuations encapsulate what the interpreter must do after the current AST is fully evaluated to a value. We represent continuations as a stack implemented as a linked list, The chain of continuations combines the roles of an operand stack and the procedure call stack of a more traditional bytecode interpreter.

As a simple example, listing 3 shows the **`begin`** continuation, implementing the **`begin`** form. This form evaluates a sequence of expressions, returning the value of the last. The begin continuation is therefore used to mark that the results of all but the last expression are ignored. The `plug_reduce` method of a continuation is called with the result of evaluating the previous expression in the **`begin`** form as the `vals` argument. That value is then ignored and the next expression is evaluated. For the last expression of the **`begin`** no new continuation is needed, and the value of that expression is eventually passed on to the previous continuation.

```
class BeginCont(Cont):
 # constructor omitted
 def plug_reduce(self, vals):
  jit.promote(self.ast)
  env = self.env
  i = self.i
  if i == len(self.ast.body) - 1:
   return self.ast.body[i], env, self.prev
  else:
   cont = BeginCont(self.ast, i+1, env, self.prev)
   return self.ast.body[i], env, cont
```

Listing 3. Begin continuation

As an optimization, we fuse the expression re-construction (*plug*) and new-expression evaluation (*reduce*) portions of the CEK-machine, a standard practice in optimizing abstract machines. Thus the `plug_reduce` method directly finds the next expression to evaluate.

Prior to interpretation, we translate all expressions to A-normal form (Danvy 1991; Flanagan et al. 1993). This introduces additional let-bindings for all non-trivial expressions (e.g. function application), so that function operands and the test of an **`if`** are always either constants or variables. This transformation is not required for our implementation, but significantly simplifies the continuations we generate, enabling the tracing JIT compiler to produce better code.

3.3 Environments

Environments are implemented as linked lists of arrays of values. Listing 4 shows the implementation of the environment class. It stores a list of values and a reference to the outer environment. AST nodes track the lexical nesting structure, meaning that environments need not store variable names (see section 4.2).

Almost all variables in typical Racket programs are immutable, but **`set!`** must also be supported. To simplify the representation of environments, Pycket performs *assignment conversion*: every mutable variable is transformed into an immutable one whose value is a mutable heap cell. This makes mutable variables somewhat slower, but benefits all others by enabling the JIT compiler to look up variables only once.

```
class ConsEnv(Env):
 # constructor omitted
 @jit.unroll_safe
 def lookup(self, sym, env_structure):
  jit.promote(env_structure)
  for i, s in enumerate(env_structure.symbols):
   if s is sym:
    v = self.value_list[i]
    assert v is not None
    return v
  return self.prev.lookup(sym, env_structure.prev)
```

Listing 4. Environment and lookup

Pycket also simplifies environment and closures to handle simple recursive functions, as is common in functional language implementations. For example Pycket implements a simplified form of the letrec rewriting for procedures (Waddell et al. 2005).

3.4 Values

Values in Pycket are represented straightforwardly as instances of several classes, one class for each kind such as fixnum, flonum, bool, pair, vector, etc. We specialize the representation of vectors to enable unboxed storage for homogeneous vectors of fixnums or flonums using the *storage strategies* technique (Bolz et al. 2013).

Racket's pairs are immutable (mutable pairs are constructed by `mcons` and accessed with `mcar` and `mcdr`). Therefore it is possible to choose one of several specialized representations of pairs at allocation time. Currently Pycket only specializes pairs when the `car` is a fixnum, which it represents in unboxed form.

In all other cases, unlike Racket and almost all other functional language implementations, fixnums are heap allocated. Pycket does not use pointer tagging.

4. Interaction with Meta-Tracing

One immediate hurdle when applying RPython's meta-tracing JIT compiler to Pycket is that one of the hints that the interpreter author needs to give is where loops at the language level can occur. Because Racket transforms loops into recursive function calls, we mark the start of a function as a place where the JIT compiler should start tracing. This is done by setting a flag `should_enter` on the body AST of every lambda form. The bytecode dispatch loop (listing 1) reads the flag (line 5) and calls the corresponding method on the JIT driver (line 6). Another hint is given just inside the loop body, indicating that this is the main loop of the interpreter (line 3).

A further set of hints indicate that data structure are immutable. RPython assumes that all instances are mutable; since ASTs, continuations, and environments are immutable, they can be marked as such. Finally, many functions in the interpreter contain loops which should be unrolled while tracing, specified by a function decorator (listing 4, line 3).

A small further optimization is the creation of size-specialized classes for many classes in the interpreter, such as `ConsEnv` and continuation frame classes. This removes a layer of indirection by putting the content directly into the class.

4.1 Allocation-Removal

The main optimization that the JIT compiler performs after it has traced a hot piece of code is that it removes allocations of immediate data structures that have a predictable lifetime (Bolz et al. 2011a). This optimization is particularly important for Pycket. When running the interpreter without meta-tracing, a lot of data-structures are continually allocated: every call to an `interpret` method allocates a new triple as its return value and new continuations are instantiated all the time. Most of the time these do not live very long, and the JIT compiler can fully remove the overhead of their use. In particular, simple tail-recursive functions are turned into loops on the machine code level that, most of the time, do not need to allocate environments or continuations.

For recursive functions where the recursive call is not in tail position the situation is slightly more interesting. The JIT compiler still produces a loop from the start of the function to the recursive call. However, because there is something left to do after the base case is reached, every iteration of the loop allocates a continuation. This continuation records what is left to do upon the return of the recursive call. When the base case is eventually reached, these continuations are then activated and removed again. This produces a second loop which effectively unwinds the stack of continuations. In this way, even recursion that cannot be directly mapped to iteration is compiled as two loops in Pycket.

4.2 Lexical Addressing for Free

During the rewriting of the AST, another piece of information is computed for every AST node. Since Racket is lexically scoped, it is possible to determine the static structure of the environment at every point in the AST. Environments are linked lists of arrays of values, thus static environments are a linked list of arrays of names.

In traditional Scheme systems this environment structure is used to assign two numbers to every variable which encode the position of the variable in the stack of frames. In Pycket, this encoding is not necessary. Instead, every time a variable in the environment is examined, the structure of the environment is traversed in parallel to the environment, until the looked-for name is found. While this appears much less efficient than just using the two indices to pick the right place, the JIT compiler will produce the same machine code as if the encoding to numbers was used.

This works because the environment structure is an immutable data structure that is part of the AST. Thus, the JIT can constant-fold all computations that inspect it. In particular, in listing 4, the loop on line 6 is unrolled and the condition on line 7 is constant-folded. The generated code simply traverses the stack of environments to the right one, and then reads the value at the correct index.

The approach is made possible by separating the static part of the environment, its structure, from the data structure that stores different values at runtime. The technique of separating static and dynamic data structure components is common in the context of partial evaluation, and called *binding time improvement* (Jones et al. 1993, Chapter 12). Jørgensen (1992) provides an example of using binding time improvement for environments.

Table 1. Extreme runtimes in milliseconds

	Gambit	Larceny	Pycket	Racket
ctak	859 ±36	1965 ± 15	304 ±1	40774 ±81
fibc	768 ±52	1750 ± 28	1061 ±2	27064 ±94
pi	657 ± 7	44742 ±234	624 ±4	502 ± 2

5. Results

We compared Pycket to Racket and several Scheme implementations to test its performance and therefore our hypothesis.

Hardware We conducted the experiments on an Intel Xeon E5410 (Harpertown) clocked at 2.33 GHz with 2×6 MB cache and 16 GB of RAM. All benchmarks used are single-threaded, hence the number of cores (four) was irrelevant to the experiment. Although virtualized on Xen, the machine was dedicated to the benchmarks.

Software The machine ran Ubuntu 12.04.4 LTS with a 64 bit Linux 3.2.0. We used the benchmarking framework *ReBench*[2] to carry out all executions and measurements. RPython as of revision d86c4a65f830 served for translating Pycket.

Implementations Racket v6.0, Gambit v4.7.2, Larceny v0.97, and Pycket as of revision 10ed5db6a395 were used in the benchmarks. All Gambit programs were compiled with `-D__SINGLE_HOST`.

Benchmarks The benchmark suite consists of the "CrossPlatform" benchmark from Larceny, comprising well-known Scheme benchmarks originally collected for Gambit. We omit those benchmarks that use mutable pairs, as well as those using I/O and threads, which we have not yet implemented.

Methodology Every benchmark was run 10 times uninterrupted at highest priority, in a new process. The runtime (*total time*) was measured *in-system* and, hence, does not include start-up; however, warm-up was not separated, so JIT compiler runtime is included in the numbers. We report the arithmetic mean of the ten runs along with bootstrapped (Davison and Hinkley 1997) confidence intervals showing the 95 % confidence level.

The results are summarized in Figure 1. The runtime per benchmark of each implementation is normalized to Racket. Pycket's performance on individual benchmarks ranges from approximately three times slower to two times faster than Racket, in nine instances even faster than Gambit. The geometric mean of all measurements compared suggests that Pycket is about as fast as Racket, as depicted by the bars labeled "geometric mean" in the figure.

Three benchmarks are not included in the comparison above, as the differences were so extreme that they skew the overall numbers. As Table 1 suggests, Pycket was one to two orders of magnitude faster than Racket for the *ctak* and *fibc* benchmarks. Both make heavy use of continuations, and hence benefit from the CEK-machine nature of Pycket, whereas Racket's implementation of continuations is known to be slow. On these benchmarks, Pycket is close to or faster than Gambit and Larceny. On the *pi* benchmark, which emphasizes bignum performance, Larceny is much slower than the other implementations.

Pycket with the meta-tracing JIT compiler disabled runs generally 40 times slower than the meta-traced verision.

6. Related Work

As mentioned in the introduction, functional languages in general, and Scheme in particular, have a long tradition of highly optimizing AOT compilers. Rabbit, by Steele (1978), following on the initial design of the language, demonstrated the possibilities of continuation-passing style and of fast first-class functions. Subsequent systems such as Gambit (Feeley 2014), Bigloo (Serrano and Weis 1995), Larceny (Clinger and Hansen 1994), Stalin (Siskind

[2] `https://github.com/smarr/ReBench`

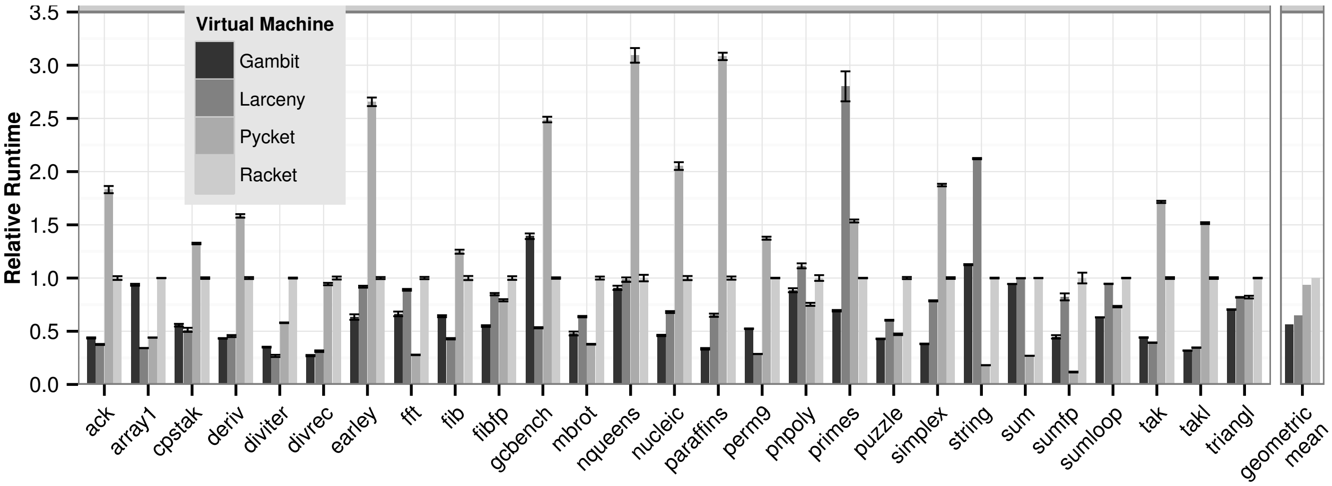

Figure 1. Benchmark runtime results. Each bar shows the arithmetic mean of 10 runs, normalized to Racket.

1999), and Chez (Dybvig 2011) have pioneered a variety of techniques for static analysis, optimization, and memory management, among others. Most other Scheme implementations are interpreters, either directly on the AST or on a bytecode representation. Racket is the only widely used system in the Scheme family with a JIT compiler, and even that is less dynamic than many modern JIT compilers.

Many Lisp implementations provide means for programmers to manually type-specialize code with specialized operations or type declarations. Racket provides these operations, but we have not yet evaluated their effect on performance in Pycket.

JIT compilation has been extensively studied in the context of object-oriented, dynamically-typed languages (Aycock 2003). For Smalltalk-80, Deutsch and Schiffman (1984) developed a JIT compiler from bytecode to native code. Chambers et al. (1989) explored using type specialization and other optimizations in Self, a closely-related language. Further research on Self applied more aggressive type specialization (Chambers and Ungar 1991) and improved the compiler's selection of methods (Hölzle and Ungar 1994).

With the rise in popularity of Java, JIT compilation became a mainstream enterprise, with a significant increase in the volume of research. The Hotspot compiler (Paleczny et al. 2001) is representative of the Java JIT compilers. JIT compilation has also become an important topic in the implementation of JavaScript (see for example (Hölttä 2013)) and thus a core part of modern web browers. For strict functional languages other than Scheme, such as OCaml, JIT compilers exist (Starynkevitch 2004; Meurer 2010), however, they are typically auxiliary to the usually much faster AOT compiler implementations.

As mentioned in the introduction, Mitchell (1970) introduced the notion of tracing JIT compilation, and Gal et al. (2006) used tracing in a Java JIT compiler. Since then, Gal et al. (2009) developed a tracing JIT compiler for JavaScript and LuaJIT[3] is a very successful tracing JIT compiler for Lua. Further work was done by Bebenita et al. (2010) who created a tracing JIT compiler for Microsoft's Common Intermediate Language (CIL) and applied it to a JavaScript implementation in C#. Schilling (2013, 2012) developed a tracing JIT compiler for Haskell based on LuaJIT called *Lambdachine*. Due to Haskell's lazy evaluation, the focus is quite different than ours. One goal of Lambdachine is to achieve deforestation (Wadler 1988; Gill et al. 1993) by applying allocation-removal techniques to traces. Lambdachine is between 50 % faster and two times slower than GHC on the small benchmarks the thesis reports on (Schilling 2013).

The core idea of meta-tracing, which is to trace an *interpreter* running a program rather than a program itself, was pioneered by Sullivan et al. (2003) in DynamoRIO.

There were experiments with applying meta-tracing to a Haskell interpreter written in RPython (Thomassen 2013). The interpreter also follows a variant of a high-level semantics of the Core of Haskell (Launchbury 1993). While the first results were promising, it supports a very small subset of primitives so that not many interesting benchmarks run on it.

7. Future Directions

With approximately two person-months of effort, Pycket has demonstrated that meta-tracing JIT compilers are competitive with mature AOT compilers for classic functional programming languages. RPython's meta-tracing approach takes a simple implementation of the CEK-machine and turns it into fast machine code.

However, much remains to investigate in this direction. As we have seen, Pycket is on-average almost two times slower than Gambit, and significantly slower than that on some benchmarks. We conjecture that this gap can be closed, but more work is required to find out. Additionally, Pycket does not yet implement some of Racket's runtime features, including threads, continuation marks, and delimited control operators. These features may require changes that significantly affect performance, but we conjecture that they do not since they map naturally onto our CEK-machine architecture.

Additionally, a wide variety of further optimization opportunities present themselves. For example, while we implement storage strategies for vectors, all integers stored in environment locations, heap cells, and continuations are boxed—storage strategies may also have a role to play here. They may also allow us to implement Racket's ubiquitous lists in new ways, taking advantage of new functional data structures such as Hash-Array Mapped Tries (Bagwell 2001). We plan to investigate whether tracing provides performance advantages for complex control flow such as that generated by contract checking, objects implemented via macros and structures, or even interpreters written in Racket itself. Finally, since meta-tracing has accelerated the CEK-machine so effectively, these techniques may also apply to other abstract machines.

[3] http://luajit.org

Acknowlegements Bolz is supported by the EPSRC *Cooler* grant EP/K01790X/1. Siek is supported by NSF Grant 1360694. We thank Anton Gulenko for implementing storage strategies.

References

J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.

P. Bagwell. Ideal hash trees. In *Es Grands Champs*. Citeseer, 2001.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. *SIGPLAN*, 35(5):1–12, 2000.

M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. In *Proc. OOPSLA*, pages 708–725, 2010.

C. F. Bolz and L. Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 2013.

C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proc. ICOOOLPS*, pages 18–25, 2009.

C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. *Proc. PEPM*, pages 43–52, 2011a.

C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In *Proc. ICOOOLPS*, page 9:1–9:8, 2011b.

C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. In *Proc. OOPSLA*, pages 167–182, 2013.

C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. *Lisp Symb. Comput.*, 4(3):283–310, 1991.

C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Proc. OOPSLA*, pages 49–70, 1989.

W. D. Clinger and L. T. Hansen. Lambda, the ultimate label or a simple optimizing compiler for Scheme. In *Proc. LFP*, pages 128–139, 1994.

O. Danvy. Three steps for the CPS transformation. Technical Report CIS-92-02, Kansas State University, 1991.

A. C. Davison and D. V. Hinkley. *Bootstrap Methods and Their Application*, chapter 5. Cambridge, 1997.

L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proc. POPL*, pages 297–302, 1984.

R. K. Dybvig. Chez Scheme version 8 user's guide. Technical report, Cadence Research Systems, 2011.

M. Feeley. Gambit-C: A portable implementation of Scheme. Technical Report v4.7.2, Universite de Montreal, February 2014.

M. Felleisen and D. P. Friedman. Control operators, the SECD-machine and the lambda-calculus. In M. Wirsing, editor, *Proc. 2nd Working Conf. on Formal Description of Programming Concepts - III*, pages 193–217. Elsevier, 1987.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. PLDI*, pages 502–514, 1993.

M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. URL `http://racket-lang.org/tr1/`.

A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proc. VEE*, pages 144–153, 2006.

A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proc. PLDI*, pages 465–478. ACM, 2009.

A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proc. FPCA*, pages 223–232, 1993.

M. Hölttä. Crankshafting from the ground up. Technical report, Google, August 2013.

U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proc. PLDI*, pages 326–336, 1994.

N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.

J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Proc. POPL*, pages 258–268, 1992.

J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL*, pages 144–154, 1993.

B. Meurer. OCamlJIT 2.0 - Faster Objective Caml. *CoRR*, abs/1011.1783, 2010.

J. G. Mitchell. *The Design and Construction of Flexible and Efficient Interactive Programming Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970.

M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Proc. JVM*, pages 1–1. USENIX Association, 2001.

A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Proc. DLS*, pages 944–953, 2006.

T. Schilling. Challenges for a Trace-Based Just-In-Time Compiler for Haskell. In *Implementation and Application of Functional Languages*, volume 7257 of *LNCS*, pages 51–68. 2012.

T. Schilling. *Trace-based Just-in-time Compilation for Lazy Functional Programming Languages*. PhD thesis, University of Kent at Canterbury, 2013.

M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Static Analysis*, volume 983 of *LNCS*, pages 366–381. 1995.

J. M. Siskind. Flow-directed lightweight closure conversion. Technical Report 99-190R, NEC Research Institute, Inc., 1999.

M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. *Revised^6 Report on the Algorithmic Language Scheme*. Cambridge, 2010.

B. Starynkevitch. OCAMLJIT – A faster just-in-time OCaml implementation. In *First MetaOCaml Workshop*, Oct. 20 2004.

G. L. Steele. Rabbit: A compiler for Scheme. Technical Report AI-474, MIT, 1978.

G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proc. IVME*, pages 50–57, 2003.

G. L. Sussman and G. L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Technical Report AI-349, MIT, 1975.

E. W. Thomassen. Trace-based just-in-time compiler for Haskell with RPython. Master's thesis, Norwegian University of Science and Technology Trondheim, 2013.

O. Waddell, D. Sarkar, and R. K. Dybvig. Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. *Higher Order Symbol. Comput.*, 18(3-4):299–326, 2005.

P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proc. ESOP*, pages 231–248, 1988.

Table 2. All benchmarks

Benchmark	Gambit		Larceny		Pycket		PycketNoJit		Racket	
	mean	error	mean	error	mean	error	mean	error	mean	error
ack	85 ms ±	0	73 ms ±	0	356 ms ±	5	10312 ms ±	32	195 ms ±	3
array1	594 ms ±	6	217 ms ±	0	279 ms ±	1	39376 ms ±	223	634 ms ±	1
cpstak	991 ms ±	23	913 ms ±	35	2356 ms ±	4	82679 ms ±	391	1779 ms ±	10
ctak	859 ms ±	36	1965 ms ±	15	304 ms ±	1	9447 ms ±	26	40774 ms ±	81
deriv	781 ms ±	3	821 ms ±	18	2861 ms ±	25	85341 ms ±	204	1806 ms ±	12
diviter	568 ms ±	7	435 ms ±	18	939 ms ±	4	104737 ms ±	807	1620 ms ±	6
divrec	842 ms ±	16	970 ms ±	21	2930 ms ±	18	97819 ms ±	1504	3104 ms ±	33
earley	757 ms ±	33	1098 ms ±	7	3176 ms ±	44	72730 ms ±	848	1196 ms ±	9
fft	632 ms ±	22	848 ms ±	5	265 ms ±	2	54706 ms ±	276	954 ms ±	8
fib	1909 ms ±	1	1278 ms ±	8	3714 ms ±	27	117401 ms ±	2132	2976 ms ±	43
fibc	768 ms ±	52	1750 ms ±	28	1061 ms ±	2	17831 ms ±	46	27064 ms ±	94
fibfp	1051 ms ±	6	1624 ms ±	1	1515 ms ±	7	47600 ms ±	1001	1916 ms ±	25
gcbench	1965 ms ±	37	751 ms ±	8	3511 ms ±	34	27192 ms ±	76	1410 ms ±	7
mbrot	865 ms ±	33	1150 ms ±	4	682 ms ±	1	59947 ms ±	246	1804 ms ±	18
nqueens	1006 ms ±	4	1091 ms ±	5	3425 ms ±	19	142284 ms ±	373	1108 ms ±	25
nucleic	655 ms ±	2	967 ms ±	2	2917 ms ±	37	26723 ms ±	41	1421 ms ±	20
paraffins	920 ms ±	18	1787 ms ±	38	8467 ms ±	31	62351 ms ±	385	2746 ms ±	31
perm9	1216 ms ±	7	665 ms ±	1	3194 ms ±	34	48332 ms ±	386	2323 ms ±	5
pi	657 ms ±	7	44742 ms ±	234	624 ms ±	4	1594 ms ±	26	502 ms ±	2
pnpoly	772 ms ±	8	972 ms ±	9	656 ms ±	1	87984 ms ±	677	872 ms ±	17
primes	1323 ms ±	14	5355 ms ±	287	2934 ms ±	29	87414 ms ±	498	1910 ms ±	4
puzzle	972 ms ±	3	1368 ms ±	3	1069 ms ±	17	127085 ms ±	681	2268 ms ±	16
simplex	746 ms ±	4	1537 ms ±	6	3665 ms ±	13	80971 ms ±	165	1956 ms ±	11
string	174 ms ±	1	328 ms ±	0	28 ms ±	0	28 ms ±	0	155 ms ±	0
sum	490 ms ±	0	518 ms ±	1	140 ms ±	0	82006 ms ±	406	519 ms ±	1
sumfp	567 ms ±	4	1044 ms ±	22	148 ms ±	0	41359 ms ±	543	1267 ms ±	47
sumloop	518 ms ±	0	777 ms ±	1	601 ms ±	6	87337 ms ±	914	822 ms ±	2
tak	1049 ms ±	2	934 ms ±	1	4078 ms ±	7	136038 ms ±	1719	2377 ms ±	14
takl	716 ms ±	0	779 ms ±	1	3414 ms ±	5	215217 ms ±	651	2252 ms ±	13
triangl	1537 ms ±	4	1788 ms ±	6	1792 ms ±	29	161541 ms ±	1770	2183 ms ±	5