Meta-tracing makes a fast Racket

Carl Friedrich Bolz  
Software Development Team  
King’s College London  
cbolz@gmx.de

Tobias Pape  
Hasso-Plattner-Institut  
Potsdam  
Tobias.Pape@hpipotsdam.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 five times faster than Gambit, three times 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 two times slower than Racket, eight times slower than Gambit, and six 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 [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. If 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 final language implementation. Rigo and Pedroni [2006] and Bolz and Tratt [2013] describe an efficient implementation of Python, called PyPy, using meta-tracing.

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 demonstrate 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 don’t 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 [Boz 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 run more than a certain number of times.

Because the produced machine code is linear, it needs to encode potential control flow branches. This is done with guards, a special kind of instruction that encode 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, machine code 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 \( \lambda \)-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 standard functional compiler transformations, including lambda-lifting and inlining, as well as simpler transformations such as constant propagation and dead code elimination. In particular, significant inlining is performed in this pass, including 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 [Holzle 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. 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 efficient implementation from such a high-level interpreter.

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). 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. When a transition produces a value, the machine 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.

3.1 Expressions

The abstract syntax tree (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
continue

```python
cont = BeginCont(self.ast, i+1, env, self.prev)
```

Continuations, which form a stack implemented as a linked list, encapsulate what the interpreter must do after the current AST is fully evaluated to a value. The chain of continuations is equivalent to the evaluation 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 is passed on to the previous continuation.

```python
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
```

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.

```python
class ConsEnv(Env):
    # constructor omitted
    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 self.prev.lookup(sym, env_structure.prev)
```

Pycket also simplifies the environment and closure representations to handle simple recursive functions, as is common in functional language implementations. For example Pycket represents a simplified form of the letrec rewriting for lambdas of Waddell et al. [2005].

### 3.4 Values

Values in Pycket are represented straightforwardly as a number of classes, such as fixnum, flonum, bool, cons, vector, etc. We specialize the representation of vectors to enable unboxed storage for homogeneous vectors of fixnums or flonums using a technique called storage strategies [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 chose one of several specialized representations of cons cells 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 [Boz et al., 2009]. Since 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 of the interpreter main loop (line 3).

A further set of hints can be applied almost everywhere. RPython assumes that all instances are mutable. Since ASTs, continuations, and environments are immutable, they can be marked as such. Furthermore, many functions in the interpreter contain loops which should be unrolled while tracing, specified by a function decorator.

A small further optimization is the creation of size-specialized classes for many classes in the interpreter, such as the environment and continuations. This removes one layer of indirection by putting the content directly into the class instead of a referenced array.

4.1 Allocation-Removal

The main optimization that the JIT compiler performs after it has traced an interesting piece of code is that it removes allocations of immediate data structures that have a predictable life-time [Boz et al., 2011a]. This optimization is particularly important for Pycket. When just running the interpreter without meta-tracing, a lot of data-structures are continually allocated: Every call to an interpret method will allocate a new triple as its return value, new continuations are instantiated all the time. Most of the time these don’t 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 no allocations of environments or continuations.

For recursive functions where the recursive call is not in tail position the situation is slightly more interesting. The JIT compiler will still produce a loop from the start of the function to the recursive call. However, since there is something left to do after the base case is reached, every iteration of the loop actually 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 proceeds 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 Environment Lookup

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 structure of the environment at every point in the AST. Since environments are linked lists of arrays of values, an environment structure is 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

<table>
<thead>
<tr>
<th>Table 1: Extreme runtimes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Gambit mean</td>
</tr>
<tr>
<td>Larceny mean</td>
</tr>
<tr>
<td>Pycket mean</td>
</tr>
<tr>
<td>Racket mean</td>
</tr>
<tr>
<td>--------------------------</td>
</tr>
<tr>
<td>ctk</td>
</tr>
<tr>
<td>866 ± 34 ms</td>
</tr>
<tr>
<td>1993 ± 21 ms</td>
</tr>
<tr>
<td>304 ± 2 ms</td>
</tr>
<tr>
<td>38911 ± 85 ms</td>
</tr>
<tr>
<td>--------------------------</td>
</tr>
<tr>
<td>fibc</td>
</tr>
<tr>
<td>712 ± 5 ms</td>
</tr>
<tr>
<td>1782 ± 29 ms</td>
</tr>
<tr>
<td>1070 ± 9 ms</td>
</tr>
<tr>
<td>26123 ± 83 ms</td>
</tr>
<tr>
<td>--------------------------</td>
</tr>
<tr>
<td>pi</td>
</tr>
<tr>
<td>660 ± 9 ms</td>
</tr>
<tr>
<td>44704 ± 166 ms</td>
</tr>
<tr>
<td>629 ± 2 ms</td>
</tr>
<tr>
<td>505 ± 3 ms</td>
</tr>
</tbody>
</table>

on line 5 is unrolled and the condition on line 6 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. This technique 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.

5. RESULTS

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

**Hardware** The processor used was an Intel Xeon E5410 (Harperstown) clocked at 2.33 GHz with 2x6 MB cache; 16 GB of RAM were available. All runs are un-parallelized, 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. The benchmarking framework ReBench\(^1\) was used to carry out all execution collection of measurements. RPython as of revision 88e5cd5b5 served for translating Pycket.

**Implementations** Racket v6.0, Gambit v4.7.2, Larceny v0.97, and Pycket as of revision aa845425 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 compiled 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 two times slower to two times faster than Racket, in six instances even faster than Gambit. The geometric mean of all measurements compared suggests that Pycket is about as fast as Gambit, 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 ctk 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

\(^1\)https://github.com/smarr/ReBench
emphasizes bignum performance, Larceny is much slower than the other implementations.

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], Biglou [Serrano and Weis, 1995], Larceny [Clinger and Hansen, 1994], 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 S, a closely-related language. Further research on S 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. For other strict functional languages, such as OCaml, JIT compilers exist [Starynkevitch, 2004, Meurer, 2010], however, they are typically auxiliary to the usually much faster AOT compiler implementations. JIT compilation has also become an important in the implementation of JavaScript (see for example [Hölttä, 2013]) and thus a core part of modern web browsers.

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\(^2\) 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 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 2× slower than GHC on the small benchmarks. The thesis reports on [Schilling, 2013].

The core idea of meta-tracing, which is tracing not the program directly, but the interpreter running the program was pioneered by DynamoRIO [Sullivan et al., 2003].

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 2 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.

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

\(^2\)http://luajit.org
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.

Acknowledgements Bolz is supported by the EPSRC Cooler grant EP/K01790X/1. Siek is supported by NSF Grant 1360694. We’d like to thank Anton Gulenko for implementing storage strategies.

References


and Technology Trondheim, 2013.
### Table 2: All benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Gambit</th>
<th>Larceny</th>
<th>Pycket</th>
<th>Racket</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mean</td>
<td>error</td>
<td>mean</td>
<td>error</td>
</tr>
<tr>
<td>ack</td>
<td>85 ms</td>
<td>± 0</td>
<td>73 ms</td>
<td>± 0</td>
</tr>
<tr>
<td>array1</td>
<td>603 ms</td>
<td>± 5</td>
<td>217 ms</td>
<td>± 0</td>
</tr>
<tr>
<td>cpstk</td>
<td>993 ms</td>
<td>± 19</td>
<td>883 ms</td>
<td>± 2</td>
</tr>
<tr>
<td>ctk</td>
<td>866 ms</td>
<td>± 34</td>
<td>1993 ms</td>
<td>± 21</td>
</tr>
<tr>
<td>deriv</td>
<td>784 ms</td>
<td>± 6</td>
<td>809 ms</td>
<td>± 7</td>
</tr>
<tr>
<td>diviter</td>
<td>581 ms</td>
<td>± 15</td>
<td>423 ms</td>
<td>± 4</td>
</tr>
<tr>
<td>divrec</td>
<td>831 ms</td>
<td>± 8</td>
<td>960 ms</td>
<td>± 5</td>
</tr>
<tr>
<td>earley</td>
<td>744 ms</td>
<td>± 23</td>
<td>1094 ms</td>
<td>± 1</td>
</tr>
<tr>
<td>fft</td>
<td>624 ms</td>
<td>± 15</td>
<td>844 ms</td>
<td>± 3</td>
</tr>
<tr>
<td>fib</td>
<td>1910 ms</td>
<td>± 1</td>
<td>1273 ms</td>
<td>± 10</td>
</tr>
<tr>
<td>fibc</td>
<td>712 ms</td>
<td>± 5</td>
<td>1782 ms</td>
<td>± 29</td>
</tr>
<tr>
<td>fibp</td>
<td>1050 ms</td>
<td>± 5</td>
<td>1624 ms</td>
<td>± 4</td>
</tr>
<tr>
<td>gbch</td>
<td>1938 ms</td>
<td>± 8</td>
<td>770 ms</td>
<td>± 13</td>
</tr>
<tr>
<td>mbrot</td>
<td>835 ms</td>
<td>± 7</td>
<td>1153 ms</td>
<td>± 5</td>
</tr>
<tr>
<td>nqueens</td>
<td>1013 ms</td>
<td>± 17</td>
<td>1093 ms</td>
<td>± 7</td>
</tr>
<tr>
<td>paraffins</td>
<td>905 ms</td>
<td>± 7</td>
<td>1763 ms</td>
<td>± 9</td>
</tr>
<tr>
<td>perm9</td>
<td>1202 ms</td>
<td>± 4</td>
<td>668 ms</td>
<td>± 5</td>
</tr>
<tr>
<td>pi</td>
<td>660 ms</td>
<td>± 9</td>
<td>44704 ms</td>
<td>± 166</td>
</tr>
<tr>
<td>pnpoly</td>
<td>777 ms</td>
<td>± 12</td>
<td>964 ms</td>
<td>± 2</td>
</tr>
<tr>
<td>primes</td>
<td>1311 ms</td>
<td>± 4</td>
<td>5253 ms</td>
<td>± 46</td>
</tr>
<tr>
<td>puzzle</td>
<td>972 ms</td>
<td>± 3</td>
<td>1375 ms</td>
<td>± 5</td>
</tr>
<tr>
<td>simplex</td>
<td>755 ms</td>
<td>± 18</td>
<td>1555 ms</td>
<td>± 19</td>
</tr>
<tr>
<td>sum</td>
<td>489 ms</td>
<td>± 0</td>
<td>517 ms</td>
<td>± 0</td>
</tr>
<tr>
<td>sumfp</td>
<td>566 ms</td>
<td>± 2</td>
<td>1046 ms</td>
<td>± 21</td>
</tr>
<tr>
<td>sumloop</td>
<td>518 ms</td>
<td>± 0</td>
<td>779 ms</td>
<td>± 5</td>
</tr>
<tr>
<td>tak</td>
<td>1048 ms</td>
<td>± 2</td>
<td>935 ms</td>
<td>± 1</td>
</tr>
<tr>
<td>takl</td>
<td>718 ms</td>
<td>± 1</td>
<td>779 ms</td>
<td>± 0</td>
</tr>
<tr>
<td>triangl</td>
<td>1539 ms</td>
<td>± 5</td>
<td>1785 ms</td>
<td>± 5</td>
</tr>
</tbody>
</table>