

Pycket: A Tracing JIT For a Functional Language

Spenser Bauman Carl Friedrich Bolz Robert Hirschfeld
Vasily Krilichev Tobias Pape Jeremy Siek Sam Tobin-Hochstadt

Abstract

We present Pycket, a high-performance tracing JIT compiler for Racket. Pycket supports a wide variety of sophisticated Racket features, such as contracts, structures, continuations, parameters, classes, and more. On average over a standard suite of benchmarks, *Pycket outperforms existing compilers*, both Racket’s JIT and other highly-optimizing Scheme compilers. Finally, Pycket provides much better performance for proxies than existing systems, *dramatically reducing the overhead of contracts and gradual typing*.

Pycket is implemented using the RPython meta-tracing framework (originally created for PyPy), which automatically generates a tracing JIT compiler based on a high-level interpreter. All optimization is performed at runtime and the JIT operates directly on core Racket forms. The interpreter is based on the well-known CEK abstract machine, enabling support for proper tail calls and first-class continuations. In the setting of a functional language, the most significant challenge to achieving high performance with a tracing JIT compiler is identifying hot loops to be optimized—we discuss multiple strategies and measure their impact.

1. Introduction

Contemporary high-level languages like Java, JavaScript, Haskell, ML, Lua, Python, and Scheme rely on sophisticated compilers to produce high-performance code. Two broad traditions have emerged in the implementation of such compilers. In functional languages, such as Haskell, Scheme, ML, Lisp, or Racket, ahead-of-time (AOT) compilers perform substantial static analysis, assisted by type information available either from the language’s type system or programmer declarations. In contrast, dynamic, object-oriented languages, such as Lua, JavaScript, and Python, following the tradition of Self, are supported by just-in-time (JIT) compilers which analyze the execution of programs and dynamically compile to machine code.

While both of these approaches have produced notable progress, each comes with drawbacks. In particular, for dynamically typed functional languages, high performance traditionally requires the addition of type information, whether in the form of declarations (Common Lisp), type-specific operations (Racket), or an additional static type system (Typed

Racket). Furthermore, AOT compilers have so far been unable to remove the overhead associated with highly-dynamic programming patterns, such as the dynamic checks used to implement contracts and gradual types. In these situations, programmers often compromise with respect to software engineering best practices to achieve better performance.

To address these drawbacks and to explore a blank spot in the compiler design space, we present **Pycket**, a JIT compiler for Racket, a dynamically typed mostly functional language descended from Scheme. Pycket is implemented using the RPython meta-tracing framework, which automatically generates a JIT compiler from an interpreter written in a restricted subset of Python.

To demonstrate the effectiveness of Pycket, consider a function computing the dot product of two vectors in Racket:

```
(define (dot u v) (for/sum ([x u] [y v]) (* x y)))
```

This implementation uses a Racket comprehension, which iterates in lockstep over u and v , binding their elements to x and y , respectively. The `for/sum` operator performs a summation over the values generated in each iteration, in this case the products of the vector elements. This `dot` function works over arbitrary sequences (lists, vectors, specialized vectors, etc) and uses generic arithmetic. Dot product is at the core of many numerical algorithms (Demmel 1997). We use this function as a running example throughout the paper.

In Racket, the generality of `dot` comes at a cost. If we switch from general to floating-point specific vectors and specialize the iteration and numeric operations, we increase performance by a $6\times$. On the other hand, if we increase safety by adding contracts, checking that the inputs are vectors of floats, the performance is cut in half.

In Pycket, the generic version runs at almost the same speed—the *overhead of generic sequences, vectors, and arithmetic is eliminated*. In fact, the code generated for the inner loop is identical. The execution speed is 5 times faster than the straightforwardly specialized version in Racket and 1.5 times faster than most manually optimized code we wrote for Racket. Furthermore, we present an optimization technique based on dynamic construction of call graphs which *eliminates the overhead of contracts in dot*—the generated inner loop is identical to `dot` without contracts. While this

optimization is not yet profitable for all programs, it is extremely effective on code produced by contracts.

With Pycket, we depart from traditional Lisp and Scheme compilers in several of ways. First, we do *no AOT optimization*. The only transformations statically performed by Pycket are converting from core Racket syntax to A-normal form (ANF) and converting assignments to mutable variables to heap mutations. We present an overview of Pycket’s architecture and major design decisions in section 2.

Second, Pycket performs aggressive run-time optimization by leveraging RPython’s *trace-based compilation* facilities. With trace-based compilation, the runtime system starts by interpreting the program and watching for hot loops. Once a hot loop is detected, the system records the actual instructions executed in the loop and optimizes the resulting straight-line trace. Subsequent executions of the loop use the optimize trace instead of the interpreter. Thus, Pycket automatically sees through indirections due to pointers, objects, or higher-order functions. We present background on tracing in section 3 and optimizations to the interpreter in section 5.

Trace-based compilation poses a significant challenge for functional languages: What constitutes a loop? The only looping mechanism in Racket is the function call, but not all function calls create loops. Tracking only tail-recursive functions detects too few loops; tracking every repeated line of code detects too many. Section 4 describes two techniques: one that addresses the false loop problem and a call graph based technique that is highly effective for contracts.

Overall, we make the following contributions.

1. We describe the first high-performance JIT compiler for a dynamically-typed functional language.
2. We show that tracing JIT compilation works well for eliminating the overhead of proxies and contracts.
3. We show that our combination of optimizations eliminates the need for manual specialization.

We validate these contributions with an extensive evaluation of each contribution, presented in section 6. Our results show that Pycket is the fastest compiler among several mature, highly-optimizing systems such as Bigloo, Gambit, and Larceny on their own benchmark suite, and that its performance on contracts is substantially better than Racket and other JIT compilers. Finally, we show that specialization is not needed for good performance in Pycket.¹

2. Pycket Primer

This section presents the architecture of Pycket but defers the description of the JIT to section 3. Pycket is an implementation of Racket, but is built in a way that generalizes to other dynamically-typed, functional languages. Fundamentally, Py-

cket is an implementation of the CEK machine (Felleisen and Friedman 1987), but scaled up from the lambda calculus to most of Racket, including macros, assignment, multiple values, modules, structures, continuation marks and more. While this precise set of features will not be present in other languages, the need to handle higher-order functions, dynamic data structures, control operators, and dynamic binding is common to many languages.

To describe the implementation of Pycket, consider again the dot product function from the introduction:

```
(define (dot u v) (for/sum ([x u] [y v]) (* x y)))
```

This example presents several challenges. First, `for/sum` and `define` are macros, which must be expanded to core syntax before interpretation. Second, these macros rely on runtime support functions from libraries, which must be loaded to run the function. Third, this loop is implemented with a tail-recursive function, which must avoid stack growth. We now describe our solutions to each of these challenges in turn.

Macros & Modules Almost every form in a Racket program is a macro, which is expanded into a few *core forms* implemented by the runtime system (Tobin-Hochstadt et al. 2011). To support macros, we could implement the Racket macro system, including its interaction with modules (Flatt 2002) in Pycket; instead we reuse Racket’s macro expander.

To run a Racket program,² Pycket uses Racket to macro-expand all the modules used in a program and write the resulting forms and metadata to files. Pycket then reads the serialized representation, parses it to an AST, and executes it. Adopting this technique enables Pycket to handle the full Racket language while focusing on the key research contributions.

Assignment conversion and ANF Once a module is expanded to core Racket and parsed from JSON, Pycket performs two transformations on the AST. First, the program is converted to ANF, ensuring that all non-trivial expressions are named (Danvy 1991; Flanagan et al. 1993). Strictly speaking, this is an AOT optimization for Pycket, the only one we perform, even though the traditional use of ANF is to enable further analysis and optimization, which we do not do. We discuss below why ANF improves performance and the specific challenges in an interpreter only context.

Next, we convert all mutable variables (those that are the target of `set!`) into heap-allocated cells. This is a common technique in Lisp systems, and Racket performs it as well. This approach allows environments to be immutable mappings from variables to values, and localizes mutation in the heap. Additionally, we record the static environment at each AST node on the node; see section 5.1 for how this is used.

CEK states and representation With our program in its final form, we now inject it into a CEK machine triple with an empty continuation and environment, and execute the

¹ This paper builds on preliminary work presented by Bolz et al. (2014) at the Workshop on Dynamic Languages and Applications. This paper reports major improvements with respect to performance, coverage of the Racket language, and breadth of benchmarks, including programs with contracts.

² Pycket supports programs written as modules but not an interactive REPL.

following (slightly simplified) interpreter main loop until completion:

```
try:
    while True:
        ast, env, cont = ast.interpret(env, cont)
except Done, e:
    return e.values
```

This is a piece of RPython code, that continuously transforms an `(ast, env, cont)` CEK triple into a new one, by calling the `interpret` method of the `ast`, with the current environment `env` and continuation `cont` as an argument. This process goes on until the continuation is the empty continuation, in which case a `Done` exception is raised, which stores the return value.

Here the `ast` is the result of A-normalization and assignment conversion. Environments and continuations are straightforward linked lists of frames, although environments store only values, not variable names, using the static environment information to aid lookup. Continuation frames are the frames from the CEK machine, extended to handle Racket's additional core forms such as `begin` and `letrec`.

Each continuation also contains information for storing *continuation marks*, a Racket feature supporting stack inspection and dynamic binding. Because each continuation frame is represented explicitly and heap-allocated in the interpreter, rather than using the conventional machine stack, first-class continuations as created by `call/cc` are straightforward to implement, and carry very little run-time penalty, as the results of section 6 show. In this, our runtime representation resembles that of Standard ML of New Jersey (Appel and MacQueen 1991).

This approach also makes it trivial to implement proper tail calls, as required by Racket and the Scheme standard (Sperber et al. 2010). On the other hand, one complication is that runtime primitives that call Racket procedures must be written in a variant of continuation-passing style.

Contracts and Chaperones One distinctive feature of Racket is the extensive support for higher-order software contracts (Findler and Felleisen 2002). Software contracts allow programmers to specify preconditions, postconditions, and invariants using the programming language itself. This enables debugging, verification, program analysis, random testing, and gradual typing, among many other language features. Higher-order contracts allow greater expressiveness, scaling these features to modern linguistic constructs, but at the cost of wrappers and indirections, which often entail noticeable performance overhead.

In Racket, contracts are implemented using the *chaperones* proxying mechanism (Strickland et al. 2012), and make heavy use of Racket's structure system. These are the most complex parts of the Racket runtime system that Pycket supports—it provides comprehensive implementations of both. This support is necessary to run both the Racket

standard library and most Racket programs. Our implementations of these features follow the high-level specifications closely. In almost all cases, the tracing JIT compiler is able to nonetheless produce excellent results.

Primitives and values Racket comes with over 1 400 primitive functions and values; Pycket implements close to half of them. These range from numeric operations, where Pycket implements the full numeric tower including bignums, rational numbers, and complex numbers, to regular expression matching, to input/output including a port abstraction. As of this writing, more than half of the non-test lines of code in Pycket implement primitive functions.

One notable design decision in the implementation of primitive values is to abandon the Lisp tradition of pointer tagging. Racket and almost all Scheme systems, along with many other language runtimes, store small integers (in Racket, up to 63 bits on 64-bit architectures) as immediates, and only box large values, taking advantage of pointer-alignment restrictions to distinguish pointers from integers. Some systems even store other values as immediates, such as symbols, characters, or even cons cells. Instead, all Pycket values are boxed, including small integers. This has the notable advantage that Pycket provides excellent performance on the full range of machine integers, but means that without JIT optimization, even simple programs perform poorly.

Limitations While Pycket is able to run a wide variety of existing Racket programs out of the box, it is not a complete implementation. The most notable absence is concurrency and parallelism: Racket provides threads, futures, places, channels, and events; Pycket implements none of these. Given the CEK architecture, the addition of threads (which do not use OS-level parallelism in Racket) should be straightforward. However, true parallelism in RPython-based systems remains a work in progress. Other notable absences include Racket's FFI, and large portions of the IO support, ranging from custom ports to network connectivity.

3. Background on Tracing JITs and RPython

Having described the basic architecture of Pycket, the next few sections explain how a high-level interpreter is turned into an optimizing JIT. We will again use the dot product from the introduction as an example.

A tracing JIT compiler optimizes a program by identifying and generating optimized machine code for the common execution paths. The unit of compilation for a tracing JIT is a loop body, requiring a heuristic to identify loops during interpretation. For the dot function, the identified loop is the tail-recursive call in the code generated by `for/sum`.

When a hot loop is identified, the JIT starts tracing the loop. The tracing process records the operations executed by the interpreter for one iteration of the loop. The JIT then optimizes the instruction sequence and generates machine code which will be used on subsequent iterations of the loop.

```

try:
    while True:
        driver.jit_merge_point()
        prev = ast
        ast, env, cont = ast.interpret(env, cont)
        if ast.should_enter:
            driver.can_enter_jit()
except Done, e:
    return e.values

```

Figure 1. Interpreter main loop with hints

During tracing, the JIT inserts guards into the trace to detect when execution diverges from the trace and return control to the interpreter. Frequently taken fall back paths are also candidates for tracing and optimization. In the dot function, guards are generated for the loop termination condition (one for each sequence). Additional tests, such as dynamic type tests, vector bounds checks, or integer overflow checks, are optimized away.

The RPython (Bolz et al. 2009; Bolz and Tratt 2013) project consists of a language and tool chain for implementing dynamic languages. The RPython language is a subset of Python amenable to type inference and static optimization. The tool chain translates an interpreter, implemented in RPython (“Restricted Python”), into an efficient virtual machine, inserting the necessary runtime components, such as a garbage collector and JIT compiler, automatically. The translation process generates a control flow graph of the interpreter and performs type inference. This representation is gradually simplified to a low-level intermediate representation that is easily translated to machine code and is suitable for use in the tracing JIT.

Tracing JITs typically operate directly on a representation of the program; in contrast, the JIT generated by RPython operates on a representation of the interpreter; that is, RPython is a *meta-tracing JIT*. To make effective use of the RPython JIT, the interpreter source must be annotated to help identify loops in the interpreted program, and to optimize away the overhead of the interpreter.

For Pycket, we annotate the main loop of the CEK interpreter as in figure 1: The annotations indicate that this is the main loop of the interpreter (`jit_merge_point`) and that AST nodes marked with `should_enter` are places where a loop in the interpreted program might start (`can_enter_jit`). At these places the JIT inspects the state of the interpreter by reading the local variables and then transfers control to the tracer.

In a conventional tracing JIT, loops can start at any back-edge in the control-flow graph. In contrast, Pycket requires special care to determine where loops can start because the control flow of functional programs is particularly challenging to determine; see section 4 for the details.

3.1 Generic RPython optimizations

RPython performs a large number of optimizations to traces. These optimizations are generic and not specialized to Pycket, but they are essential to the performance of Pycket.

Standard Optimizations RPython’s trace optimizer includes a suite of standard compiler optimizations, such as common-subexpression elimination, copy propagation, constant folding, and many others (Ardö et al. 2012). One advantage of trace compilation for optimization is that the control-flow graph of a trace is just a straight line. Therefore, the optimizations and their supporting analyses are implemented in two passes over the trace, one forward pass and one backward pass.

Inlining Inlining is a vital compiler optimization for high-level languages, including both functional and OO systems. In a tracing JIT compiler such as RPython, *inlining comes for free* from tracing (Gal et al. 2006). A given straight-line trace will include the the inlined code from any functions called during the trace. This includes Racket-level functions as well as runtime system functions (Bolz et al. 2009). The highly-aggressive inlining produced by tracing is one of the keys to its successful performance: it eliminates function call overhead and exposes opportunities for other optimizations.

Loop invariant code motion Another common compiler optimization is loop-invariant code motion. RPython implements this in a particularly simple way, by peeling off a single iteration of the loop, and then performing its standard suite of forward analyses to optimize the loop further (Ardö et al. 2012)³. Because many loop-invariant computations are performed in the peeled iteration, they can then be omitted the second time, removing them from the actual loop body.

The style of code generated by Pycket is so allocation and computation intensive that it relies more heavily on this optimization than PyPy does. In some cases, the RPython JIT was not able to optimize some Pycket loops, requiring changes to the optimizer to handle those cases.

Allocation removal The CEK machine allocates a vast quantity of objects, each of which would nominally appear in the heap without optimization. This ranges from the tuple holding the three components of the machine state, to the environments holding each variable, to the continuations created for each operation. For example, a simple two-argument multiply operation, as found in dot, will create and use 3 continuation frames. Additionally, since both integers and floating-point numbers are boxed, unlike in typical Scheme implementations, many of these allocations must be eliminated to obtain high performance. Fortunately, RPython’s optimizer is able to see and remove allocations that do not live beyond the scope of a trace (Bolz et al. 2011).

³Developed originally by Mike Pall in the context of LuaJIT.


```

loop header
  label(p3, f58, i66, i70, p1, i17, i28, p38, p48)
  guard_not_invalidated()
loop termination tests
  i71 = i66 < i17
  guard(i71 is true)
  i72 = i70 < i28
  guard(i72 is true)
vector access
  f73 = getarrayitem_gc(p38, i66)
  f74 = getarrayitem_gc(p48, i70)
core operations
  f75 = f73 * f74
  f76 = f58 + f75
increment loop counters
  i77 = i66 + 1
  i78 = i70 + 1
jump back to loop header
  jump(p3, f76, i77, i78, p1, i17, i28, p38, p48)

```

Figure 2. Optimized trace for dot inner loop

3.2 An optimized trace

The optimized trace for the inner loop of dot is shown in figure 2. Traces are represented in SSA form (Cytron et al. 1991). Variable names are prefixed according to type, with p for pointers, f for floats, and i for integers. Several aspects of this trace deserve mention. First, the loop header and final jump appear to pass arguments, but this is merely a device for describing the content of memory—no function call is made here. Second, we see that there are two loop counters, as generated by the original Racket program. More sophisticated loop optimizations could perhaps remove one of them. Third, note that nothing is boxed—floating point values are stored directly in the array, the sum is stored in a register, and the loop counters are as well. This is the successful result of the allocation removal optimization. Third, note that no overflow, tag, or bounds checks are generated. Some have been hoisted above the inner loop and others have been proved unnecessary. Finally, the `guard_not_invalidated()` call at the beginning of the trace does not actually become a check at run-time—instead, it indicates that some other operation might invalidate this code, forcing a recompile.

4. Finding Loops

As described in the introduction, determining where loops start and stop in a functional language, especially one implemented using the CEK machine, is challenging. In this section, we describe two approaches to detect appropriate loops, both of which are dynamic variants of the “false loop filtering” technique (Hayashizaki et al. 2011). The first approach is almost always profitable, and we use it in our standard configuration for evaluation. The second has more limited

applicability—it enables high performance on proxies and contracts, but can interfere with performance elsewhere.

4.1 Two State Representation

In general, the beginning of every lambda may denote a trace header (i.e. it may be a profitable location to begin tracing). Though many functions encode loops, treating the body of every lambda as a trace header produces many traces that do not encode loops. A non-recursive function called repeatedly in the body of a loop will initiate tracing at the top of the function and trace through its return into the body of the loop. Treating the current AST node as the location in the program does not allow the JIT to distinguish between functions called consecutively (“false loops”) and recursive invocations which encode a loop.

```

(define (g a b) (+ a b))
(define (f a b) (g a b) (g a b) (f a b))

```

Consider the example above, which creates two functions `f` and `g`, both of two arguments. The `g` function computes the sum of its two arguments, while `f` invokes `g` on its arguments twice and then calls itself with the same arguments. Although it never terminates, `f` provides a simple example of a false loop. The `f` function is a tail recursive loop, with two trace headers observed during the execution of this loop: at the beginning of the loops and at each invocation of `g`.

Because `g` is invoked twice as frequently as `f` per iteration, the JIT first sees `g` as hot and begins tracing at one of the invocations of `g`. Regardless of which call site initiates tracing, the trace will consist of one call to `g` along with the part of `f` bridging the two calls. Neither possible trace covers the whole loop in `f`. Instead, the loop is encoded as a single trace, consisting of the body of `g`, then inspecting the continuation to determine the return location and proceeding to trace until the next call to `g`. Continuation inspection is encoded, by tracing, as a sequence of guards, which may be as long as the number of call sites of `g`. This results in frequently taken guards, degrading performance. Because optimizations are only done within a trace, breaking a loop into many traces results in each being less optimized than the equivalent single trace.

One way to “trick” the JIT into tracing a whole loop is to encode the location in the program as a pair of the current and previous AST node (with respect to execution). This distinction only matters at potential trace headers (the top of every lambda body). As such, each trace header is encoded as the body of the lambda along with its call site. The modified CEK loop above stores this additional state in the `prev` variable (figure 1), which is picked up by the JIT at the calls to the methods `jit_merge_point` and `can_enter_jit`.

In the example above, each invocation of `g` would appear to the JIT as separate locations. Tracing will now begin at the first invocation of `g`, but will proceed through the second call and around the loop. Even though tracing did not begin at the “top” of the loop, the trace still covers a whole iteration. Such

a “phase shift” of the loop has no impact on performance. This approach is a simplified version of the one proposed by Hayashizaki et al. (2011). Their proper solution is to look at more levels of the stack to decide whether something is a false loop.

For recursive invocations of a function, this approach introduces little overhead as a function body will only have a few recursive call sites, though one trace will be generated for each recursive call site. Such functions may trace through several calls, as tracing will only terminate at a recursive call from the same call site which initiated the trace.

Detecting loops in this manner also has the potential to generate more traces for a given function: one per call site rather than just one for the function’s body, as each trace is now identified by a call site in addition to the function which initiated the trace. More traces are often beneficial for performance in code with data dependent control flow, as a loop may be encoded as multiple specialized traces with infrequently taken guards between them, as opposed to frequently taken guards encoding the control flow.

4.2 Call Graph

When used in isolation, the approach described in the previous section does not produce satisfactory results for contracts. This is because the approach can be defeated by making calls go through a common call site, as in the following modified version of `g`:

```
(define (g a b) (+ a b))
(define (call-g a b) (g a b))
(define (f* a b) (call-g a b) (call-g a b) (f* a b))
```

In such cases, the common call site hides the loop from the tracer, resulting in the same behavior as the original example. This behaviour is a particular problem for contracts, leading to unsatisfactory results for code that uses them. Contracts use many extra levels of indirection and higher order functions in their implementation. Thus, the one extra level of context used by the two state approach is not consistently able to identify false loops.

To address this problem we developed a further, still experimental, technique. It is not enabled yet in Pycket by default, because it does not help all benchmarks. It uses a more systematic approach to loop detection that makes use of runtime information to construct a call graph of the program. A call graph is a directed graph with nodes corresponding to the source level functions in the program, and edges between nodes *A* and *B* if *A* invokes *B* in operator position. We compute an under-approximation of the call graph at runtime. To achieve this, whenever the interpreter executes an application, the invoked function is inspected, and the lambda expression which created the function is associated with the lambda expression containing the current application. Thus portions of the full call graph are discovered incrementally.

Loop detection in the program then reduces to detecting cycles in the call graph. Cycle detection is also done online;

when adding an edge to the call graph, Pycket checks for a cycle along the newly added edge. When a cycle is detected, a node in the cycle is annotated as potential loop header, which the JIT might then decide to trace. As opposed to the two state representation, this is an opt-in strategy—initially, *no* AST nodes are potential trace headers. If the loop is generated by a tail-recursive loop, simple cycles are all the system needs to worry about. In the example above, `f*` is the only cycle in the call graph, so correctly marked as the only loop.

However, non tail-recursive loops are broken up in the CEK machine into two loops. Consider the example of `append`, which loops over its first argument, producing one continuation frame per `cons` cell. When `append` reaches the end of the first list, the accumulated continuation is applied to `ys`; the application of the continuation will continue, with each continuation `cons-ing` an element onto its argument. Thus, non tail-recursive functions consist of a loop which builds up continuation frames (the “up” recursion) and a loop which tears down continuation frames and performs outstanding actions (the “down” recursion).

To expose this fact to the JIT, the call graph also inspects the continuation after discovering a cycle. If the current continuation frame was generated by the function which created the call graph cycle, the continuation is inspected for the AST element which will be executed upon application of the continuation, and this AST element is also marked as a loop header. For the ANF version of the `append` example, the beginning of the function body would be marked as a loop for an “up” recursion, as would the body of the innermost `let`, containing the application of the `cons` function, which receives the result of the recursive call, performs the `cons` operation, and invokes the previous continuation.⁴

The call graph approach is very good at improving the performance of contract benchmarks. For reasons not yet understood, this approach sometimes slows down non-contract code in significant ways by making the tracer produce much longer traces. Furthermore, the call graph recording and, in particular, cycle detection are not cheap operations, so doing them can itself be a source of slowdown. Overall, the call graph technique produces a 4 % slowdown across the non-contract benchmarks. For the individual benchmarks the results are quite varied, with some benefiting significantly, and some being up to 8.5 times slower.

5. Optimizing Interpreter Data Structures

In this section, we describe a number of independent optimizations we applied to the data structures in Pycket, some novel and some taken from the literature, which contribute significantly to overall performance. For each of these optimizations, we report how they impact performance by comparing Pycket in its standard mode (all optimizations on) to

⁴ A special case is primitive functions that are themselves loops, such as `map`. They must be marked in the interpreter source code so that the JIT generates a trace for them, even though there is no loop in the call graph.

Pycket with the particular optimization off, across our full benchmark suite.

5.1 Optimizing Environments in the presence of ANF

As described in section 2 we translate all expressions to ANF prior to interpretation. This introduces additional let-bindings for all non-trivial subexpressions. Thus, function operands and the conditional expression of an `if` are always either constants, variables, or primitive operations that do not access the environment or the continuation, such as `cons`. The transformation to ANF is not required for our implementation, but significantly simplifies the continuations we generate, enabling the tracing JIT to produce better code.

Traditionally, ANF is used in the context of AOT compilers that perform liveness analysis to determine the lifetime of variables and make sure that they are not kept around longer than necessary. This is not the case in a naive interpreter such as ours. Therefore, ANF can lead to problems in the context of Pycket, since the inserted let-bindings can significantly extend the lifetime of an intermediate expression.

As an example of this problem, the following shows the result of transforming the `append` function to ANF:

```
(define (append xs ys)
  (if (null? xs) ys
      (cons (car xs) (append (cdr xs) ys))))
⇒
(define (append xs ys)
  (let ([test (null? xs)])
    (if test ys
        (let ([head (car xs)])
          (let ([tail (cdr xs)])
            (let ([rec (append tail ys)])
              (cons head rec))))))
```

In the resulting code, `(cdr xs)` is live until the call to `cons`, whereas in the original code that value was only live until the recursive call to `append`. Even worse, the result of the test is live for the body of the `append` function. This problem is not unique to ANF—code written by the programmer can exhibit the same effects. However, ANF makes the problem more common, necessitating a solution in Pycket.

In Pycket, we counter the problem by attempting to drop environment frames early. Dropping an environment frame is possible when the code that runs in the frame does not access the variables of the frame at all, only those of previous frames. To this end, we do a local analysis when building a let AST. The analysis checks whether any of the outer environments of the `let` hold variables not read by the body of the `let`. If so, those environment frames are dropped when executing the `let`. In the example above, this is the case for the frame storing the `tail` variable, which can be dropped after the recursive call to `append`.

Additionally, if the parent of an environment frame is un-referenced, a new frame is created with just the child frame.

This optimization produces an effect similar to closure conversion, ensuring that closures capture only used variables.

Disabling ANF entirely in the interpreter is not possible, but we can disable Pycket’s environment optimizations. Across the full benchmark suite, Pycket, with all optimizations on, produces a speedup of 68 % over the version with only environment optimization disabled.

5.2 Type and Size Specialization

A number of interpreter-internal data structures store arrays of Racket values. Examples of these are environments and several kinds of continuations, such as those for `let` and function application. These data structures are immutable. Therefore, it is possible to choose, at creation time, a specialized variant of these data structures optimized for the particular data it stores. Simple examination of the arguments to the constructor (all data structures are classes in RPython) suffices to choose the variant.

Pycket uses two kinds of variants. First, we specialize the data structures based on the number of elements, for all sizes between 0 and 10, inclusive. This avoids an indirection to a separate array to store the actual values. More importantly, there are several type-specialized variants. This helps to address the lack of immediate small integers (so-called *fixnums*), as mentioned in section 2. Boxed integers (and other boxed data) makes arithmetic slower, because the results of arithmetic operations must be re-boxed. This is mitigated by storing the *fixnums*’ values directly in a specialized environment or continuation, without the need for an extra heap object.

All of these specializations come into play for the compilation of `dot`. The continuations and environments allocated all contain fewer than 10 values. Also, there are multiple environments that type-specialize based on their contents, such as one that holds the two integer loop counters, enabling further optimization by other parts of the trace optimizer.

In addition to the systematic specialization for continuations and environments, a few special cases of type specialization are directly coded in the representation of data structures. The most important example of these is `cons` cells that store *fixnums* in the `car`. This case again uses an unboxed representation to store the value. The optimization is made possible by the fact that Racket’s `cons` cells are immutable. As an example, figure 3 shows the data layout of a type-specialized `cons` cell that is storing an unboxed *fixnum*.

These specializations combine for significant performance benefits. Across the full benchmark suite, Pycket with all optimizations produces a speedup of 19 % over the version with type- and size-specialization disabled.

5.3 Strategies for Optimizing Mutable Objects

Optimizing mutable objects by specialization is harder than optimizing immutable objects. When the content of the mutable object is changed, the specialized representation might not be applicable any more. Thus a different approach is

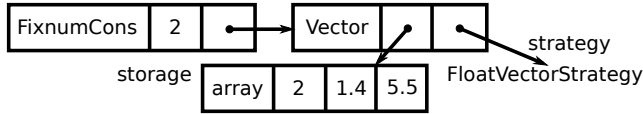


Figure 3. Optimized representation of `(1 . #(1.4 5.5))` using cons specialization and vector strategies

needed to optimize mutable data structures such as Racket’s vectors and hash tables.

For example, one would like to use a different representation for vectors that only store floating point numbers. In practice, many vectors are type-homogeneous in that way. Ideally the content of such a vector is stored in unboxed form, to save memory and make accessing the content quicker. However, because the vector is mutable, that representation needs to be changeable, for example if a value that is not a floating point number is inserted into the vector.

The RPython JIT specializes mutable collections using the *storage strategies* approach that was pioneered in the context of the PyPy project (Bolz et al. 2013). In that approach, the implementation of a collection object is split into two parts, its strategy and its storage. The strategy object describes how the contents of the collection are stored and all operations on the collection are delegated to the strategy.

If a mutating operation is performed that needs to change the strategy, a new strategy is chosen and assigned to the collection. The storage is rewritten to fit what the new strategy expects. As an example, if a vector uses a special floating point storage strategy, the storage is an array of unboxed floating point numbers. If a string is written into vector, the strategy is switched to a general one, that can store all kinds of objects. The storage is then rewritten to be a vector of boxed values. For a large vector, this is an expensive operation, and thus strategies depend for their performance on 1) the hypothesis that representation-changing mutations are rare on large data structures, and 2) the change to a more general strategy is a one-way street (Bolz et al. 2013).

Pycket uses strategies for the following kinds of objects: (a) item vectors are type specialized if they contain all fixnums or all floating-point numbers (flonums); (b) hash tables are type specialized if their keys are all fixnums, bytes, symbols, or strings; (c) strings are specialized according to the kind of characters they store (Unicode or ASCII); (d) cells (used to store mutable variables) are type specialized for fixnums and flonums.

The use of strategies for vectors is a crucial optimization for dot, our running example. When the vectors are allocated with floating-point numbers, they use a strategy specialized for flonums, avoiding unboxing and tag checks for each vector reference in the inner loop. Racket programmers can manually do this type specialization by using `flvectors`, gaining back much of the lost performance. Pycket obviates the need for manual specialization by making generic vectors perform

on par with specialized ones. As an example, figure 3 shows the data layout of a vector using a float strategy, with an array of unboxed floats as storage.

For hash maps the benefits are even larger than for vectors, as, if the key type is known, the underlying implementation can use a much more efficient hashing and comparison function. In particular, since the comparison and hash function of these types is known and does not use arbitrary stack space or `call/cc`, the hash table implementation is simpler.

Strings are mutable in Racket, so they also use storage strategies. Since most strings are never mutated, a new string starts out with a strategy that the string is immutable. If later the string is mutated, it is switched to a mutable strategy. A further improvement of strings is the observation that almost all are actually ASCII strings, even though the datatype in Racket supports the full Unicode character range. Thus a more efficient ASCII strategy is used for strings that remain in the ASCII range. This makes them much smaller in memory (since every character needs only one byte, not four) and makes operations on them faster.

One special case of strategies is used for mutable heap cells which are used to implement mutable variables—those that are the target of `set!`. Quite often, the type of the value stored in the variable stays the same. Thus when writing a fixnum or a floating point number type into the cell, the cell switches to a special strategy that stores the values of these in unboxed form (bringing the usual advantages of unboxing).

Strategies are vital for high performance on benchmarks with mutable data structures. On dot, disabling strategies reduces performance by 75%. For the benchmarks from section 6 that make extensive use of hash tables, disabling strategies makes some benchmarks *18 times slower*.

6. Evaluation

In this section, we evaluate Pycket’s performance to test several hypotheses, as described in the introduction:

1. A highly-dynamic JIT compiler produces performance competitive with mature existing AOT compilers.
2. Tracing JITs perform well for indirections produced by proxies and contracts.
3. Tracing JITs reduce the need for manual specialization.






To test the first hypothesis, we compare Pycket to Racket and 3 highly-optimizing Scheme compilers, across a range of Scheme benchmarks, and to Racket on benchmarks taken from the Racket repository. To test the second hypothesis, we measure Pycket and Racket’s performance on both micro- and macro-benchmarks taken from the paper introducing Racket’s *chaperones and impersonators*, the proxy mechanism underlying contracts, and also test several other implementations on similar benchmarks. In particular, we show how call graph based loop filtering of section 4.2 improves performance. To test the third hypothesis, we compare Py-

cket’s performance on benchmarks with and without type specialization.

Our evaluation compares Pycket with multiple configuration options and multiple other systems on a variety of programs. We present the most important results and include full results in supplemental material.

6.1 Setup

System We conducted the experiments on an Intel Xeon E5-2650 (Sandy Bridge) at 2.8 GHz with 20 MB cache and 16 GB of RAM. Although virtualized on Xen, the machine was idle. All benchmarks are single-threaded. The machine ran Ubuntu 14.04.1 LTS with a 64 bit Linux 3.2.0. We used the framework *ReBench*⁵ to carry out all measurements. RPython as of revision 0e89a463e021 was used for Pycket.

Implementations Racket v6.1.1 , Larceny v0.97 , Gambit v4.7.2 , Bigloo v4.2a-alpha13Oct14 , V8 v3.25.30 (and contracts.js⁶ version 0.2.0), Pypy v2.4.0, and Pycket  as of revision e10b989 were used for benchmarking. Gambit programs were compiled with `-D__SINGLE_HOST`. Bigloo was compiled with `-DLARGE_CONFIG` to enable benchmarks to complete without running out of heap. In a few instances, Bigloo crashed, and in one case Gambit did not compile. These results were excluded from the average.

Methodology Every benchmark was run 3 times uninterruptedly at highest priority in a new process. The execution time was measured *in-system* and, hence, does not include start-up; however, warm-up was not separated, so all times include JIT compilation. We show the arithmetic mean of all runs along with bootstrapped (Davison and Hinkley 1997) confidence intervals for a 95 % confidence level.

6.2 Benchmarks

Larceny cross-platform benchmarks The benchmark suite consists of the “CrossPlatform” benchmark from Larceny, comprising well-known Scheme benchmarks originally collected for Gambit. We increased iteration counts until Pycket took approximately 5 seconds, to lower jitter associated with fast-running benchmarks, and to ensure that we measure peak performance as well as JIT warmup (which is included in all measurements). Also, we moved all I/O out of the timed loop, and omitted a few benchmarks where I/O was the primary feature measured. All of our benchmarks and infrastructure are available at [url removed for blind review](#).

The results are summarized in Figure 4. The runtime per benchmark of each system is normalized to Racket. The geometric mean of all measurements is given in bars at the right of the figure. The chart ends at 3 times execution time relative to Racket for space reasons, but some of the benchmarks on both Pycket and Larceny are between 3 and 4 times slower than Racket. A few benchmarks perform very poorly on some

	Pycket	±	Pycket*	±	Racket	±	V8	±	PyPy	±
Bubble										
direct	689	3	645	3	1382	2	335	0	600	4
chap	11161	144	773	13	6667	6				
proxy							109985	3859	1260	15
unsafe	543	3	520	0	956	1				
unsafe*	541	3	537	3	726	1				
Church										
chap	9209	88	6421	57	37585	137				
contract	11930	4363	1241	3	10093	252	291442	2926		
direct	721	9	760	5	1235	7	2059	11	3381	43
proxy							54089	144	107075	468
Struct										
chap	132	0	135	3	5593	100				
direct	133	3	133	3	527	0	377	0	128	0
proxy							27462	1955	2057	2
unsafe	133	3	133	3	337	0				
unsafe*	132	0	132	0	337	0				
ODE										
direct	2287	10	2452	16	5372	97				
contract	4680	51	3149	7	12074	164				
Binomial										
direct	2199	9	1671	5	2888	35				
contract	23674	331	162594	44756	52811	1141				

Table 1. Execution Times (in ms) for the Chaperone Benchmarks. Pycket* is Pycket with the call graph optimization

systems; this includes *pi* (bignum performance) on Larceny, and *ctak* and *fibc* (continuations) on Racket; we omit these from figure 4 to avoid changing the scale, but include their results in overall calculations.

Pycket’s performance on individual benchmarks ranges from approximately 4 times slower to 260 times faster than Racket; in 12 of 48 cases Pycket is the fastest implementation. On average, Pycket is the fastest system, and significantly faster than Racket.

Shootout benchmarks Next, we consider Racket-specific benchmarks designed for the Computer Language Benchmarks Game. The benchmarks are all taken from the Racket source repository, version 6.1.1. We omit two benchmarks, *regexdna* and *k-nucleotide*, which require regular-expression features Pycket does not support.

On average, Pycket is 22 % faster than Racket on these benchmarks, is faster on 25 of the 40, and is up to 6 times faster on numeric-intensive benchmarks such as *nbody*, *nsievebits*, or *mandelbrot*.

Chaperone benchmarks To measure the impact of our optimizations of contracts and proxies, we re-use the benchmarks from Strickland et al. (2012). We run all of the micro-benchmarks from that paper, and two of the macro-benchmarks.⁷ The results are presented in table 1, with 95% confidence intervals. We show both Pycket’s performance in the standard configuration (first column) as well as the impact of the callgraph optimization (second column).

The micro-benchmarks form the upper part of the table, with the macro-benchmarks below. For each micro-benchmark, we include representative implementations for v8 and PyPy. The *bubble* benchmark is bubble-sort of a vector;

⁵<https://github.com/smarr/ReBench>

⁶<http://disnetdev.com/contracts.coffee/>

⁷All the other macro benchmarks require either the FFI or the meta-programming system at run-time, neither of which Pycket supports.

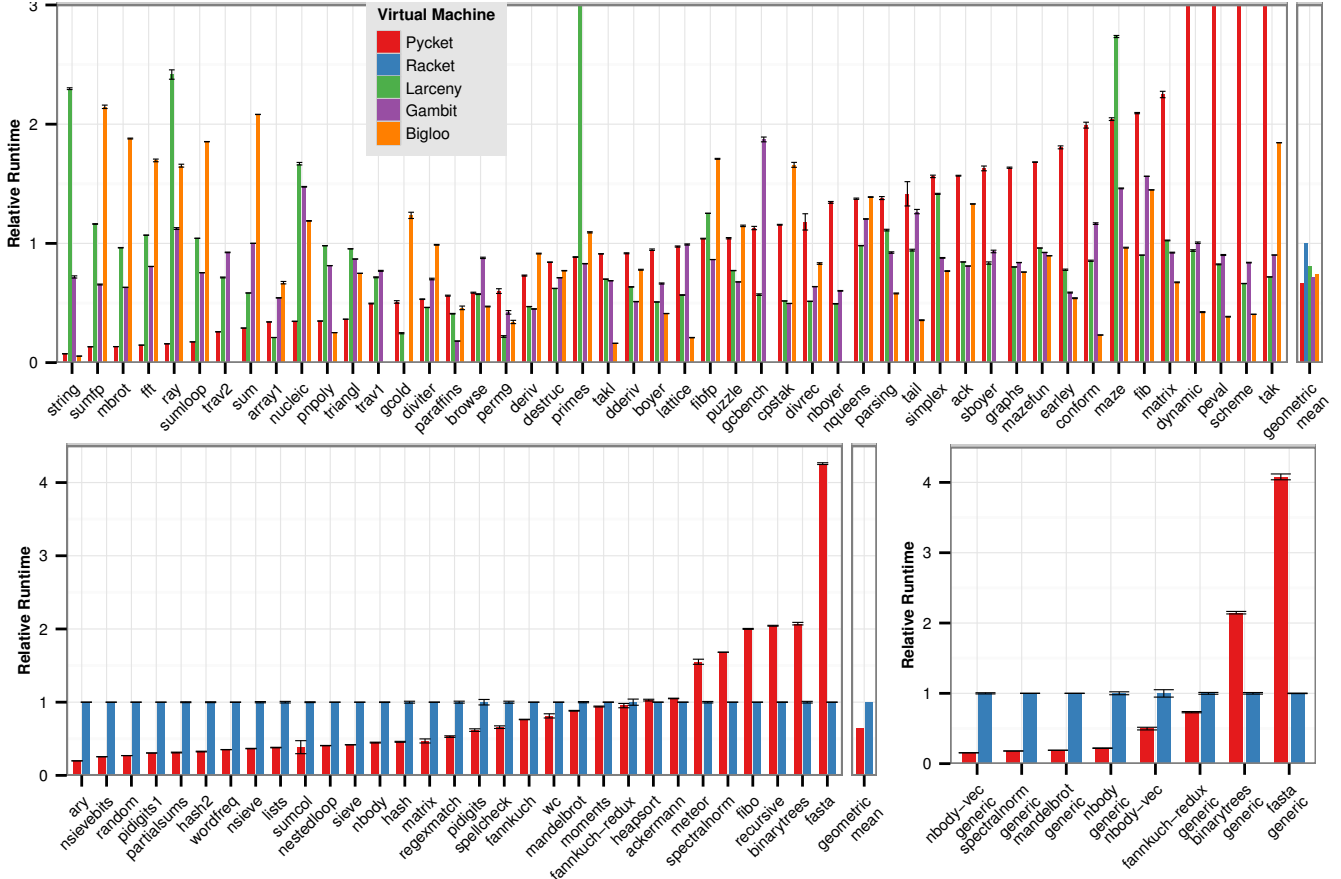


Figure 4. Scheme (above) and Racket (below) benchmarks, with runtimes normalized to Racket. Racket is omitted from the top figure for clarity. Lower is better.

struct is structure access in a loop; and *church* is calculating 9 factorial using Church numerals. Where a *chaperone*, *contract*, or *proxy* annotation is given, we used the language-appropriate mechanism.

For the macro-benchmarks, we show results only for Pycket and Racket, with contracts on and off. The two macro-benchmarks are an ODE solver and a binomial heap, replaying a trace from a computer vision application.

The results show that Pycket is competitive with other JIT compilers on micro-benchmarks, and that Pycket’s performance with the call graph optimization on contracts and proxies is far superior both to Racket’s and to other systems. In many cases, Pycket improves on other systems by factors of 2 to 100, reducing contract overhead to almost 0.

On the ODE benchmark, Pycket with the call graph provides a 4× speedup over Racket, with less than 30% contract overhead. On the binomial heap benchmark, however, Pycket’s performance is problematic. Here, the callgraph optimization slows down Pycket substantially. We believe that both Pycket’s slow performance, and the high contract overhead generally, is due to the large numbers of proxy wrappers generated by this benchmark—as many as 20 on a single object. Racket’s developers plan to address this by reducing the

number of wrappers created in the contract system,⁸ which we believe will fix the poor performance here.

Specialization benchmarks To measure Pycket’s ability to eliminate the need for manual specialization, we constructed generic versions of several of the Racket benchmarks, and compared Racket and Pycket’s performance. In all cases, Pycket’s performance advantage over Racket *improves*. These results are presented in figure 4. Pycket loses only 2 % of its performance when unspecialized, whereas Racket loses 57 %.

6.3 Discussion

Our evaluation results support our hypotheses, although there are still limitations. Pycket is faster than Racket across a broad range of benchmarks, and is competitive with highly-optimizing AOT compilers. Furthermore, Pycket can largely eliminate the need for manual specialization on types that is common in high-performance Racket and Scheme programs. Finally, call graph-based optimization radically reduces the overhead of contracts, eliminating it entirely in some cases.

⁸Robby Findler, personal communication

There are some cases where Pycket is substantially slower. These cases are almost exclusively recursive programs with data-dependent control flow in the style of an interpreter over an AST. Such programs are a known weakness of RPython’s tracing JIT compilers, although we hope to improve Pycket’s performance in the future. Another important source of slowness is JIT warmup. Since Pycket is written in such a high-level style, the JIT optimizers are taxed a lot more than in other RPython interpreters. For a number of slow benchmark JIT optimizations and code generation take up a substantive portion of the benchmark runtime, up to about half of the full execution time. We plan to address this issue by modifying interpreter to be a bit more low-level.

7. Related Work

As mentioned in the introduction, functional languages in general, and Scheme in particular, have a long tradition of 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 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 AST or bytecode interpreters. Racket is the only widely used system in the Scheme family with a JIT compiler.

Many Lisp implementations, including Racket, provide means for programmers to manually type-specialize code with specialized operations or type declarations. We support Racket’s type-specialized vectors and specialized arithmetic operations, as well as unsafe operations (e.g., eliding bounds checks). The results of our evaluation demonstrate that these manual optimizations are less necessary 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 browsers. For strict functional languages other than Scheme, such as OCaml, JIT compilers exist (Starynkevitch 2004; Meurer 2010), however, the AOT compilers for these languages are faster.

Mitchell (1970) introduced the notion of *tracing JIT compilation*, and Gal et al. (2006) used tracing in a Java JIT compiler. 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. Since then, Gal et al. (2009) developed a tracing JIT compiler for JavaScript. LuaJIT⁹ 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 CIL and applied it to a JavaScript implementation in C#. These existing tracing systems, as well as PyPy and other RPython-based systems, differ from Pycket in several ways. First, they have not been applied to functional languages, which presents unique challenges such as first-class control, extensive use of closures, proper tail calls, and lack of explicit loops. Second, these systems all operate on a lower-level bytecode than Pycket’s CEK machine, placing less burden on the optimizer. Third, few AOT compilers exist for these languages, making a head-to-head comparison difficult or impossible.

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.

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 (Launchbury 1993) of Core, the intermediate representation of the GHC compiler. While the first results were promising, it supports a very small subset of primitives leading to limited evaluation. It is unknown how well meta-tracing scales for a realistic Haskell implementation.

8. Conclusion

Pycket is a young system—it has been under development for only a year, yet it is competitive with the best existing AOT Scheme compilers, particularly on safe, high-level, generic code. Furthermore, Pycket is much faster than any other system on the indirections produced by contracts, addressing a widely noted performance problem, and making safe gradual typing a possibility in more systems.

The implementation of Pycket provides two important lessons for JITs for functional languages. First, the issue of finding and exploiting loops requires careful consideration—explicit looping constructs in imperative languages make the optimizer’s life easier. Second, once this issue is addressed, as we describe above, conventional JIT optimizations such as strategies are highly effective in the functional context.

Pycket demonstrates the value of the RPython infrastructure (Marr et al. 2014): We have built in one year and 12 000 LOC a compiler competitive with existing mature systems. We encourage other implementors to consider RPython.

⁹<http://luajit.org>

References

- A. Appel and D. MacQueen. Standard ml of new jersey. In J. Maluszynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 1991.
- H. Ardö, C. F. Bolz, and M. Fijakowski. 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](https://doi.org/10.1145/2384577.2384586). URL <http://doi.acm.org/10.1145/2384577.2384586>.
- J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.
- 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. Fijakowski, and A. Rigo. Tracing the meta-level: PyPy’s tracing JIT compiler. In *Proc. ICIOOLPS*, pages 18–25, 2009.
- C. F. Bolz, A. Cuni, M. Fijakowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. *Proc. PEPM*, pages 43–52, 2011.
- C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. In *Proc. OOPSLA*, pages 167–182, 2013.
- C. F. Bolz, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Meta-tracing makes a fast Racket. In *Workshop on Dynamic Languages and Applications*, 2014.
- 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.
- R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451490, Oct. 1991.
- 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.
- J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 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, Université 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.
- R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, ICFP, pages 48–59, October 2002.
- 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. Composable and compilable macros:: you want it when? In *ICFP ’02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 72–83. ACM Press, 2002.
- 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.
- H. Hayashizaki, P. Wu, H. Inoue, M. J. Serrano, and T. Nakatani. Improving the performance of trace-based systems by false loop filtering. *ACM SIGPLAN Notices*, pages 405–418, 2011. doi: [10.1145/1950365.1950412](https://doi.org/10.1145/1950365.1950412). ACM ID: 1950412.
- 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.
- J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL*, pages 144–154, 1993.
- S. Marr, T. Pape, and W. De Meuter. Are we there yet? Simple language implementation techniques for the 21st century. *IEEE Software*, 31(5):6067, Sept. 2014. ISSN 0740-7459. doi: [10.1109/MS.2014.98](https://doi.org/10.1109/MS.2014.98).
- 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.
- 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. *Revised⁶ 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.
- T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12*, 2012.
- 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.
- E. W. Thomassen. Trace-based just-in-time compiler for Haskell with RPython. Master's thesis, Norwegian University of Science and Technology Trondheim, 2013.
- S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11*, pages 132–141, New York, NY, USA, 2011. ACM.
- P. Wadler. Deforestation: transforming programs to eliminate trees. In *Proc. ESOP*, pages 231–248, 1988.