Rebuilding Racket on Chez Scheme (Experience Report)

MATTHEW FLATT, University of Utah, USA
CANER DERICI, Indiana University, USA
R. KENT DYBVIG, Cisco Systems, Inc., USA
ANDREW W. KEEP, Cisco Systems, Inc., USA
GUSTAVO E. MASSACCESI, Universidad de Buenos Aires, Argentina
SARAH SPALL, Indiana University, USA
SAM TOBIN-HOCHSTADT, Indiana University, USA
JON ZEPPIERI, independent researcher, USA

We rebuilt Racket on Chez Scheme, and it works well—as long as we're allowed a few patches to Chez Scheme. DrRacket runs, the Racket distribution can build itself, and nearly all of the core Racket test suite passes. Maintainability and performance of the resulting implementation are good, although some work remains to improve end-to-end performance. The least predictable part of our effort was how big the differences between Racket and Chez Scheme would turn out to be and how we would manage those differences. We expect Racket on Chez Scheme to become the main Racket implementation, and we encourage other language implementers to consider Chez Scheme as a target virtual machine.

CCS Concepts: • **Software and its engineering** → **Functional languages; Software evolution**.

Additional Key Words and Phrases: Racket, Scheme

ACM Reference Format:
Matthew Flatt, Caner Derici, R. Kent Dybvig, Andrew W. Keep, Gustavo E. Massaccesi, Sarah Spall, Sam Tobin-Hochstadt, and Jon Zeppieri. 2019. Rebuilding Racket on Chez Scheme (Experience Report). *Proc. ACM Program. Lang.* 3, ICFP, Article 78 (August 2019), 15 pages. https://doi.org/10.1145/3341642

1 STARTING A RACKET

Racket started in 1995 as a fusion of two off-the-shelf C/C++ libraries: a Scheme interpreter (Benson 1994) and a cross-platform GUI toolkit (Smart 1995). The intent was to assemble enough of a Scheme implementation to host a graphical pedagogical programming environment. The programming environment became DrRacket, and the interpreter mash-up evolved into the modern Racket core.

Although combining existing libraries is a sensible way to produce new software, picking a C-implemented interpreter for Racket does not, in retrospect, look like a well-informed choice. Starting with a slow interpreter encouraged the creation of more C code, even when the new parts included a compiler, JIT, and runtime extensions that ultimately improved Racket's performance. The main Racket distribution now consists of roughly 1.2M lines of Racket (including 270k lines of

Authors' addresses: Matthew Flatt, University of Utah, USA, mflatt@cs.utah.edu; Caner Derici, Indiana University, USA, cderici@indiana.edu; R. Kent Dybvig, Cisco Systems, Inc. USA, dyb@cisco.com; Andrew W. Keep, Cisco Systems, Inc. USA, akeep@cisco.com; Gustavo E. Massaccesi, Universidad de Buenos Aires, Argentina, gustavo@oma.org.ar; Sarah Spall, Indiana University, USA, sjspall@iu.edu; Sam Tobin-Hochstadt, Indiana University, USA, samth@cs.indiana.edu; Jon Zeppieri, independent researcher, USA, zeppieri@gmail.com.

2475-1421/2019/8-ART78
https://doi.org/10.1145/3341642

documentation source), but that code is still supported by roughly 200k lines of C. Large parts of Racket's implementation remain in C only because the original interpreter was in C, and all of that C code is relatively difficult to maintain.

Experience porting various subsystems from C/C++ to Racket—notably the cross-platform graphics and GUI layer in 2010 and the macro expander in 2016—has confirmed that Racket-implemented libraries are easier to maintain and modify, unsurprisingly. The obvious next step is to migrate the compiler and runtime system itself to a more maintainable form. Again, building on existing technology is better than starting from scratch.

There are many virtual machines that a language implementer might choose to target, but the major ones are not well suited to host a functional programming language. Most artificially limit the continuation to a fixed-size call stack, preventing a programmer from using the direct, recursive style that naturally matches a list- or tree-shaped data declaration. Some have grudgingly tacked on a tail-call instruction, but first-class continuations are right out. Most provide numerical support only in the form of floating-point numbers and small integers, leaving out arbitrary precision arithmetic. The functional-programming community sorted out these issues decades ago.

Chez Scheme became available as an open-source implementation in mid-2016. It is certainly a better-informed starting point for building a functional language, and it is an especially good match for Racket. Selecting a compiler and runtime to drop into an existing ecosystem is a different proposition than picking a base for a new language, and while Chez Scheme and Racket implement similar languages, they are different enough that success was not guaranteed. Whether and how to manage mismatches between Chez Scheme and Racket was the least predictable part of our effort, and so we concentrate on that aspect of the conversion in this experience report.

Our experience suggests that other implementations of functional programming languages could benefit from targeting Chez Scheme. While our efforts required changes to Chez Scheme, some of those may be useful to other implementers, and most of the rest are due to aiming for a very high level of compatibility with an existing system.

2 REBUILDING OVERVIEW

Figure 1 illustrates both the rebuilding task and the motivation for Racket on Chez Scheme (a.k.a. Racket CS). The leftmost column represents the content of the `racket` executable in the current Racket release; except for the macro expander, it is implemented in C. The middle column represents Chez Scheme, including its boot files; Chez Scheme has a small kernel that is written in C, but it is mostly implemented in Scheme. The rightmost column represents the new Racket implementation on Chez Scheme; besides Chez Scheme's implementation, it includes a compatibility layer that is implemented in Scheme, a C-implemented `rktio` layer that abstracts over operating-system facilities (similar to libraries like `libuv`[1]), and additional Racket-specific functionality that is implemented in Racket.

The "expander" layer at the top of both the leftmost and rightmost columns implements Racket's module and macro system, and it is the same implementation in both cases. The output of the macro expander is a set of `linklet` forms, where a small layer immediately below the "expander" layer manages compilation and evaluation of `linklet` forms. We discuss the `linklet` form in section 3. For Racket CS, the "schemify" layer converts a Racket `linklet` to a Chez Scheme `lambda`, which is then handled by the Chez Scheme compiler.[2]

[1]https://github.com/libuv/libuv

[2]Racket modules sometimes generate extremely large `linklet` forms. In that case, Racket CS interprets the outer layer of the schemified linklet and compiles only smaller, interior `lambda` forms.

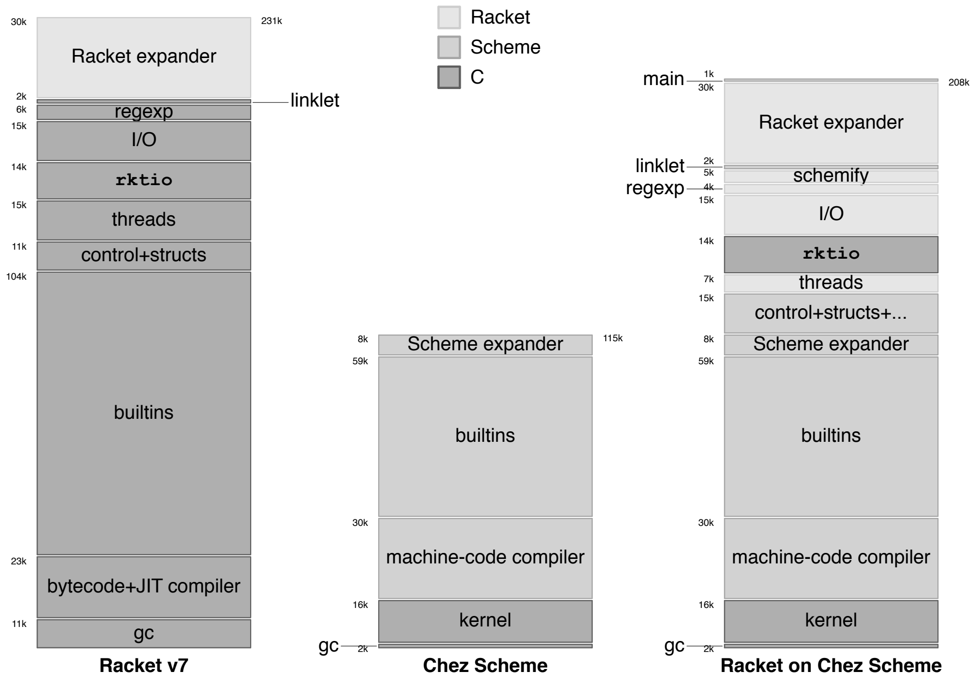
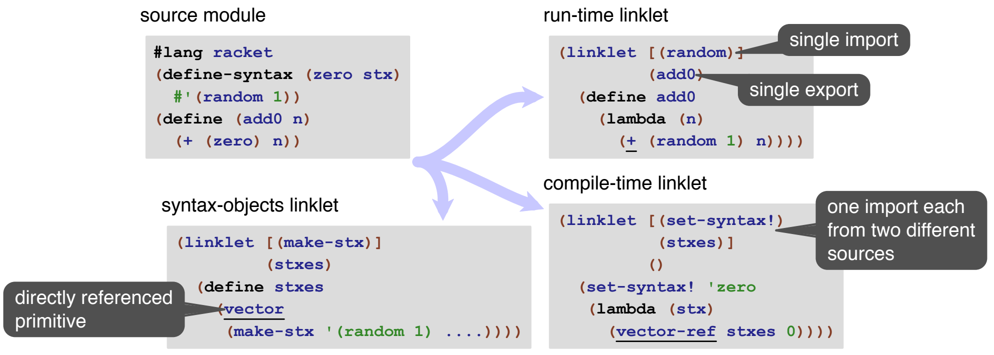


Fig. 1. Comparing the traditional Racket, Chez Scheme, and Racket CS implementations. Numbers to the left of each block are rough lines of code as measured with `wc -l`, and they add up to the number at the top right of each column. Anecdotally, relative lines of code consistently approximate relative functionality.

The layers depicted in figure 1 are mostly conceptual, except that the Racket-implemented layers correspond to distinct subsystems that can be separately compiled and tested. The “builtins” layer in each column represents a broad collection of primitive datatypes, including numbers (fixnums, flonums, exact rationals, and complex numbers), lists, strings, hash tables, records, procedures, continuations, and more. The “control+structs” layers represent Racket’s full API for delimited continuations, impersonators and chaperones, structure-type properties, and related reflective operations; for Racket CS, some of those augment or replace variants from “builtins.” The Racket CS “I/O” layer similarly replaces I/O APIs from Chez Scheme’s “builtins” with an implementation that uses `rktio` and cooperates with Racket threads. Racket threads are userspace threads with a rich system of synchronous events that is based on Concurrent ML (Reppy 1999), but the “threads” layer also includes Racket’s places and futures, which provide access to OS-level concurrency.

To a first approximation, porting Racket to Chez Scheme means developing the layers that are unique to the rightmost column of figure 1. The effort triggered changes that are already reflected in the leftmost column, such as moving parts into a stand-alone `rktio` library. More significantly, the rightmost column relies on a Chez Scheme with about 30 changes and patches. We attempted to minimize those changes, and we detail many of the trade-offs involved with those modifications in section 4.



```
#lang racket
(define-syntax (zero stx)
  #'(random 1))
(define (add0 n)
  (+ (zero) n))
```

```
(linklet [(random)]
         (add0)
  (define add0
    (lambda (n)
      (+ (random 1) n))))
```

```
(linklet [(make-stx)]
         (stxes)
  (define stxes
   (vector
     (make-stx '(random 1) ....))))
```

```
(linklet [(set-syntax!)
          (stxes)]
         ()
  (set-syntax! 'zero
   (lambda (stx)
     (vector-ref stxes 0))))
```

Fig. 2. Example expansion of a Racket module into linklets.

3 LINKLETS AND BOOTSTRAPPING

Racket's macro and module system is responsible for elaborating source programs into a core language that is consumed by the compiler. A module can not only implement syntax that is to be used in other modules, it can contain macros that extend the language used in the module's own body. The macro expander strictly separates run-time and expansion phases (and meta-expansion phase, etc.), so a single module can correspond to multiple bundles of code. For example, run time and compile time are implemented as distinct code bundles. Literal syntax objects, which are a generalization of S-expressions to accommodate binding information, bridge those two worlds, so they live in yet another code bundle.

The code bundles produced from a module use a core language that is similar to the core for any functional language, i.e., the λ-calculus with a handful of syntactic extensions. Instead of using a `lambda` form directly, however, each code bundle produced by Racket's macro expander is a `linklet` form, which consumes and produces variables that have names and are potentially mutable, instead of consuming and producing values. Figure 2 sketches the expansion of an example Racket module into a set of linklets. A simple module's expansion produces one to three linklets, but submodules or higher expansion phases can generate additional linklets.

The imports to a linklet are grouped into sets of variables, where each set will be provided by a potentially distinct linklet instance. When a linklet is instantiated, its body definitions and expressions are evaluated, and the exported subset of the defined variables are packaged up in a result linklet instance, which can be provided in turn to future linklet instantiations. By making the concepts of variables, imports, and exports explicit, the macro expander can cooperate with an underlying compiler to support cross-module optimizations (which turn into cross-linklet optimizations). Cross-module optimization in Racket CS is implemented by the schemify layer, while it is part of the lower-layer bytecode compiler in the existing Racket implementation.

Besides using core syntactic forms, a `linklet` body can directly refer to primitive functions like `vector-ref` and `+`. Those direct references allow the underlying compiler to recognize and optimize references to system primitives. Racket linklets rely on a large set of primitives—roughly 1500 of them. In the case of building Racket on Chez Scheme, we get most of those primitives for free, since a shared Lisp and Scheme heritage means that Chez Scheme already implements the majority of primitives that Racket needs. Racket- and Scheme-implemented layers provide the rest.

A Racket-implemented layer of Racket CS must be translated to Scheme to run on top of Chez Scheme. Naturally, that translation works by running it through the expander (using some existing Racket implementation), which produces a set of linklets. Then, the subset of linklets that corresponds to the layer's run-time implementation can be flattened into a single linklet, and the flattened linklet can be translated to Scheme by the schemify compiler. The macro expander and schemify can run on themselves to generate the full sets of layers. Each layer is wrapped as a Chez Scheme library, and then the set of libraries is compiled together using whole-program optimization in unsafe mode and without debugging information.

4 LANGUAGE MISMATCHES

Figure 3 provides a summary of the various ways that Racket CS initially needed different behavior from Chez Scheme. The mismatches were found by running Racket programs and the core Racket test suite as described in section 6. Some of the mismatches were resolved through schemify or the compatibility library that acts as a layer between Chez Scheme and the rest of Racket. Some mismatches were resolved by adding or changing functionality in Chez Scheme in a way that seems generally useful, and many of those changes have been merged into the main Chez Scheme implementation. Other changes to Chez Scheme are either controversial or heavyweight compared to the expected benefit for applications other than Racket, so those are organized as Racket-specific patches to Chez Scheme. A small number of those patches are marked as "for now," which means that a patch is convenient given that other patches are needed, but alternative solutions may be possible—including just accepting the mismatch. Finally, some mismatches already appear to be acceptable in the long run.

4.1 Evaluation Rules

Left-to-Right Evaluation. In Racket, a function-call expression always evaluates its argument subexpressions left-to-right. Chez Scheme follows the Scheme standard (Sperber et al. 2007), which does not specify the order of evaluation for subexpressions in a function call. This difference is managed in Racket CS by transforming a function-call form to a sequence of nested `let`s, since a `let`'s right-hand side is always evaluated before the body form. The schemify layer of Racket CS performs this transformation, and to avoid expanding code too much or unnecessarily constraining the compiler, schemify does not perform the transformation if it can determine that order does not matter.

`letrec` *and Multiple Returns.* Schemify similarly resolves a difference with `letrec`, where the Scheme standard makes the result unspecified for the following program if calling `get-f` captures a continuation that is used to return a second time.

```
(letrec ([g (lambda () f)]
         [f (get-f)])
  (g))
```

Racket specifies the behavior of this program in terms of the allocation of variable locations for `g` and `f`, and schemify implements that specification by transforming the expression to a conventional combination of `let` and `set!`. Again, the transformation should apply only when necessary, and limiting this transformation requires an analysis of `letrec` bindings in schemify, including whether variables are potentially referenced before they have a value. That analysis duplicates one that is already present in Chez Scheme, but the analysis is not onerous, and it also supports a transformation to guard potential references before initialization; the explicit guard ensures that an error reports the source name of the variable, which is otherwise mangled by macro expansion.

Delimited Continuations. Racket's support for first-class control includes delimited and composable continuations (Flatt et al. 2007). Chez Scheme provides just `call/cc`, but the Chez Scheme developers have a long record of work on continuations (Dybvig et al. 2007; Hieb et al. 1994;

Evaluation Rules		
Left-to-right evaluation	*change*	resolved by schemify
`letrec` and multiple returns	*change*	resolved by schemify
Delimited continuations	*addition*	resolved by library
Continuation marks	*addition*	patch Chez Scheme for Racket only
Preserving non-tail calls	*change*	patch Chez Scheme for Racket only
Structures and Procedures		
Applicable structures and other properties	*addition*	resolved by schemify and library
Procedure arity and name reflection	*addition*	patch Chez Scheme for Racket only
Procedure approximate result arity	*addition*	patch Chez Scheme for Racket only
Core Datatypes		
Immutable pairs	*addition*	resolved by library
Immutable vectors and strings	*addition*	modify Chez Scheme
Chaperones and impersonators	*addition*	resolved by library
Partial hash-table iteration	*addition*	modify Chez Scheme
Immutable hash tables and `eq?` hash codes	*addition*	resolved by library
Numbers		
Arithmetic special cases, such as `(/ 0 ....)`	*change*	modify Chez Scheme
Left-associative `+`, `*`, and variants	*change*	patch Chez Scheme, for now
`eqv?` on `+nan.0`	*change*	patch Chez Scheme, for now
`eq?` on flonums	*change*	patch Chez Scheme, for now
Single- and extended-precision flonums	*addition*	accept mismatch
Compilation		
Eager line/column source-location tracking	*addition*	modify Chez Scheme
Permissive library recompilation	*addition*	patch Chez Scheme for Racket only
Type reconstruction for optimization	*addition*	patch Chez Scheme for Racket only
Faster boot-file loading	*change*	patch Chez Scheme for Racket only
Flonum unboxing	*change*	accept mismatch, for now
Memory Management		
Ephemerons	*addition*	modify Chez Scheme
Ordered and unordered finalization	*addition*	patch Chez Scheme for Racket only
Memory accounting	*addition*	patch Chez Scheme for Racket only
Debugging backreferences	*addition*	patch Chez Scheme for Racket only
Phantom byte strings	*addition*	patch Chez Scheme for Racket only
Incremental garbage collection	*change*	accept mismatch, for now
Foreign-Function Interface		
Foreign-pointer representation	*addition*	resolved by library
C `struct` arguments and returns	*addition*	modify Chez Scheme
Foreign-thread activation	*addition*	modify Chez Scheme
Compare-and-set	*addition*	modify Chez Scheme
Locked versus immobile memory	*change*	accept mismatch
Exported C API	*change*	accept mismatch

Fig. 3. Summary of mismatches between Racket and Chez Scheme.

Hieb and Dybvig 1990), so it's no coincidence that the implementation is well suited to delimited control. Specifically, Chez Scheme internals include an operation to truncate a captured continuation, and Racket CS uses that operation to delimit continuations. Instead of exposing `call/cc` and `dynamic-wind` directly, Racket CS wrappers implement prompt-sensitive variants of those operations. Overall, the implementation is similar to previously reported strategies for delimited control based on metacontinuations (Danvy and Filinski 1990; Dybvig et al. 2007).

Continuation Marks. In addition to operations for capturing and restoring continuations, Racket provides *continuation marks* for reflecting on them (Clements and Felleisen 2004; Flatt et al. 2007). Continuation marks play an important role in Racket for implementing dynamic binding, exception handling, debugging facilities (Clements et al. 2001; Li and Flatt 2017), profiling (Andersen et al. 2019), and contracts. The syntactic form for installing a continuation mark,

```
(with-continuation-mark
   key-expr value-expr
   body)
```

associates the result of `key-expr` to `value-expr` in the current continuation frame, replacing any existing association for the key. Crucially, `body` remains in tail position with respect to the `with-continuation-mark` form, which is why continuation marks cannot be implemented simply by wrapping `body` with *push* and *pop* operations. Functions such as `current-continuation-marks` and `continuation-mark-set-first` provide efficient access to marks; those functions are used, for example, when accessing a dynamic binding, finding an exception handler, or reporting an exception trace.

Continuation marks can be implemented as part of the delimited-continuation implementation, but a library-based implementation does not perform well enough (and becomes a bottleneck in certain uses of contracts, for example). Part of the problem is that using `call/cc` to access the current continuation frame typically requires allocating a closure for the argument to `call/cc`. Another problem is that `call/cc` reifies a continuation in a way that allows it to be applied multiple times, while an implementation of `with-continuation-mark` needs only a one-time continuation. Finally, a library implementation of `with-continuation-mark` is difficult for the compiler to optimize—for example, to turn into a simple *push* and *pop* wrapper when that could work for a `body` expression.

Instead of adding a `with-continuation-mark` form to Chez Scheme, we added the procedure `call-adding-continuation-attachment` for associating a single attachment value to the current continuation and the procedure `call-with-current-continuation-attachment` to access the attachment value for the current continuation frame. Having a single value does not compose well compared to a key–value mapping, but the key–value mapping can be added in a library layer. Meanwhile, the compiler can recognize the continuation-attachment operations and treat them specially, much as it recognizes and treats specially `call-with-values`. The result is a continuation marks implementation that performs on par with the existing Racket implementation and is 3-4 times as fast as a library-based implementation.

Preserving Non-Tail Calls. Scheme and Racket guarantee that evaluating an expression E_1 in tail position with respect to an enclosing expression E_2 does not extend the continuation of E_1 (although subexpressions of E_2 may extend the continuation). Proper handling of tails calls is one of the big enablers of compilation from Racket to Chez Scheme. While proper tail-call handling is a guarantee of asymptotic behavior with respect to memory use, in a language with continuation marks, it becomes a semantic guarantee about the marks that are associated with a continuation.

Conversely, an expression E_1 that is *not* in tail position with respect to E_2 must extend the continuation as reflected via marks. To implement this non-tail guarantee for Racket programs, we adjusted the Chez Scheme optimizer to prevent it from transforming an expression like `(let ([x (f)]) x)` to just `(f)` when nothing more is known about `f` or about the surrounding context. Otherwise, “simplifying” the expression that way could change the behavior of continuation-mark operations in tail position within `f`. If `f` is known not to adjust or inspect continuation marks before returning, or if the `let` form is in a non-tail position with no wrapping `with-continuation-marks`, then the transformation is allowed.

A second and related reason not to perform the transformation is that `(f)` may produce multiple values. Depending on the surrounding context, the simplification may turn a result-arity exception into a permitted production of multiple values. Racket must reliably produce an exception in that case, so Chez Scheme's optimizer has been constrained to perform the transformation only when it will affect neither result-arity checking nor continuation-mark operations.

4.2 Structures and Procedures

Racket and Chez Scheme support similar constructs for declaring new structure (i.e., record) types and creating structure instances. They also support similar compiler optimizations for structure predicates and selectors. Racket further imitates Chez Scheme's `case-lambda` form to support multi-arity procedures, so Racket's core `lambda` and `case-lambda` forms map directly. However, Racket supports additional reflective operations on procedures and structures, including an option to make structure instances behave as procedures.

Applicable Structures and Other Properties. Racket supports an association of arbitrary properties to structure types. The properties are specified when the structure type is created. Associating property values to Chez Scheme structure types is straightforward, because they can be attached to the property list of the globally unique symbol that is created for each structure type.

Racket's built-in `prop:procedure` property enables an instance of a structure type with the property to be applied in the same way as a function. The property value implements the structure type's application method. While a `prop:procedure` value can be associated to a structure type in the same way as any other property value, modifying the behavior of function application is less straightforward. Changing every function call in a Racket program to implement a general method send would be prohibitively expensive; for example, changing every function call in the expander's implementation causes it to run at less than half its normal speed.

To support structures that behave as functions, schemify changes

```
(proc-expr arg-expr ...)
```

to

```
((extract-procedure proc-expr) expr-expr ...)
```

for every function call where it cannot resolve *proc-expr* to a known procedure. For the vast majority of function calls, the procedure is known, and no transformation is necessary; for example, among the 37,049 function calls in the macro expander's implementation, at most 857 (2.3%) need the transformation. To better support the cases that must be converted, `extract-procedure` can be inlined, at least for the fast path where the argument *proc-expr* produces a plain procedure. Overall, and especially since Chez Scheme tends to outperform the old Racket implementation for function calls, a rarely needed and inlined `extract-procedure` performs well enough.

Procedure Arity and Name Reflection. Given Racket's original role as a pedagogic programming environment, we committed early in the design to an operation that takes a procedure and reports the procedure's arity. That way, for example, a higher-order function like `map` can confirm that a given function will work on the expected number of arguments before applying the function, and it can report a helpful error message if not. Reflecting arity information has been helpful for implementing contracts, too.

At the same time, exposing a procedure's arity means that a wrapper procedure like `(lambda args (apply f args))` works less well, because the wrapper claims to accept any number of arguments, although it will only succeed with arguments accepted by *f*. To compensate, Racket provides a `procedure-reduce-arity` function to further wrap a procedure, but with a more specific arity. The pattern for wrapping a procedure *f* becomes

```
(procedure-reduce-arity (lambda args (apply f args))
                        (procedure-arity f))
```

While arity inspection and reduction could be implemented through applicable structures, making applicable structures so pervasive substantially reduces performance. Instead, we extended Chez Scheme with a way to report a procedure's arity, and we added a combination of a wrapper generator and `procedure-reduce-arity` to support efficient redirection of a procedure call to another procedure (i.e., without allocating a list of arguments, as the example wrapper does). The overhead of an extra indirection and jump for an arity-reduced function call is small enough that it's difficult to measure in a microbenchmark, probably due to branch prediction, whereas the slower path triggered by an applicable-structure wrapper causes a function call to be slower by roughly a factor of six.

The newly built-in wrapper facility cannot, unfortunately, improve the performance of applicable structures. Chez Scheme's representation of procedure references and structure references involve different tag bits and object layouts, so a wrapper procedure cannot work as a structure instance.

Procedure Approximate Result Arity. Racket's contract system uses arity reflection to enforce contracts, and it uses operations like `procedure-reduce-arity` to generate wrapped procedures to enforce higher-order contracts. To reduce the amount of wrapping that it performs, the contract system benefits from an operation that reports dynamically when a procedure is known to produce a single result value, even if that report is conservative. We adjusted Chez Scheme's compiler to (often) detect single-valued procedure bodies and record that result for run-time reporting.

4.3 Core Datatypes

Immutable Datatypes. Since they're both descendants of Scheme, Chez Scheme and Racket agree on most of their core datatypes. Unlike Scheme, pairs in Racket are immutable, but enforcing that property for Racket on Chez Scheme is simply a matter of withholding the `set-car!` and `set-cdr!` operations from Racket programs. Racket provides mutable pairs as a separate datatype.

Racket includes both mutable and immutable variants of Unicode strings, byte strings, vectors, and boxes. The same accessors, such as `string-ref`, must work on both mutable and immutable variants, while mutators like `string-set!` must be provided for mutable variants. Simply withholding the mutators is not an option, and adding a wrapper to distinguish different variants seems expensive. We adjusted Chez Scheme to include a mutability bit in the type tags for strings, bytes strings, vectors, and boxes. This extra bit imposes a low extra cost, because testing or non-testing for the bit mostly can be folded into existing masks and tests.

Chaperones and Impersonators. Racket's chaperones and impersonators support interposition on some primitive-datatypes operations, such as procedure application and access or update in hash tables (Strickland et al. 2012). For Racket CS, chaperones are implemented as a library in the "control+structs" layer. Procedure-application chaperoning works through applicables structures. To support interposition on operations like `vector-ref`, the library exports a replacement version that inlines a `vector?` check plus `vector-ref` selection for the fast path and dispatches to a slow path for the general case.

Hash Tables. Racket's mutable hash tables mostly can be implemented in terms of Chez Scheme's hash tables, but implementing stream-like iteration requires a new operation to Chez Scheme to access a bounded number of keys in time proportional to the bound. Racket's persistent hash tables are implemented as a library, where `eq?`-based tables rely on a global, mutable hash table with weakly held keys to map a value to a counter-based hash code, simulating an allocation address.

4.4 Numbers

Racket and Chez Scheme both implement the full Scheme numeric tower, including exact and inexact variants of rational and complex numbers. The two systems are compatible to an especially high degree, even down to choices that are not specified by the standard, such as the result of multiplication between an exact `0` and an inexact number. We made small changes to both Chez Scheme and the old Racket implementation to bring them further into line. The changes touch around 40 lines in Chez Scheme and about 30 lines in Racket (not counting tests for either).

After those changes, some differences remain. One is whether multi-argument `*` and `/` have a specified association; Racket specifies left-associative addition and multiplication, while Chez Scheme leaves the association unspecified. Racket equates all IEEE NaN representations with `eqv?`, while Chez Scheme equates only bit-identical NaNs. Racket preserves object-identity of inexact reals as detectable by `eq?`, while Chez Scheme leaves `eq?` on such numbers unspecified. Racket CS would probably work well enough if we left those differences in place, but the patches to adjust Chez Scheme are small and worthwhile if we have to patch for other reasons.

Finally, in addition to double-precision floating-point numbers, Racket supports single-precision and (on some platforms) extended-precision numbers. Those number variants are infrequently used, and we can do without them for now.

4.5 Compilation

We made a small change to Chez Scheme's compiler to accept eagerly computed line and column locations, instead of always computing them on demand from file offsets. We also adjusted Chez Scheme to allow the recompilation of certain libraries without necessarily having to recompile uses of those libraries; that adjustment facilitates the development of the Racket CS core.

More significantly, we added a type-reconstruction pass to the compiler to enable some optimizations. For example, in the pair-reversing expression `(cons (cdr p) (car p))`, a successful evaluation of `(cdr p)` implies that `p` is a pair, so a non-checking variant of `car` can be used for the second operation of `p`. Previous work added a type reconstruction pass to Chez Scheme already (Adams 2013), but that implementation has not been integrated into the Chez Scheme release. Our new pass is less ambitious, but it enables the optimizations that the old Racket implementation performs, which ensures more consistent performance in a switch to Racket CS.

4.6 Memory Management

Ephemerons, Ordered and Unordered Finalization. In addition to *weak boxes*, which are easily mapped to Chez Scheme's *weak pairs*, Racket supports *ephemerons* (Hayes 1997), which are a kind of "and" for weak references. The main use of ephemerons is to solve the key-in-value problem for weak mappings. We added ephemeron pairs to Chez Scheme in a way that avoids quadratic worst-case behavior.

Racket's main finalization construct is directly based on Chez Scheme's *guardians* (Dybvig et al. 1993). Guardians implement *unordered* finalization, where two objects that are inaccessible can both be finalized, even if each has a finalizer that refers to the other object. Our experience is that unordered finalization is the correct design for most purposes. To implement modules that are backed by foreign libraries, however, *ordered* finalization is also useful, where a reference to an object from a finalizer will prevent that object from being finalized. Ordering allows a foreign-object finalizer to run only when an object is truly inaccessible, and not potentially accessible from a client-program finalizer.

The current Racket implementation provides a limited and unsatisfying form of ordered finalization that is hard-wired to three levels of finalization; references from finalizers at level N prevent finalization at level *N*+1, while finalization is unordered within each level. For Racket CS, we have instead extended Chez Scheme with ordered guardians as an alternative to unordered guardians; a reference from a finalizer in *any* guardian prevents an object from being finalized through an ordered guardian. This new design is more general, and it works for Racket because existing foreign-library bindings accommodate either an unordered or leveled interpretation of finalization.

Memory Accounting, Debugging Backreferences, Phantom Byte Strings, and Incremental Garbage Collection. Programs that are developed in DrRacket run on the same Racket instance as DrRacket itself. To prevent a program under development from consuming so much memory that it terminates the programming environment, Racket supports allocation limits that are tied to a *custodian* (Wick and Flatt 2004), which is a language construct that abstracts the concept of a process-like resource domain (Flatt et al. 1999). Chez Scheme includes a `compute-size` debugging function computes the memory use from a given starting object. We extended that function to add `compute-size-deltas`, which implements the ordering that is needed to assign charges to the correct custodian within a tree of Racket threads.

Racket's `dump-gc-stats` helps in debugging resource leaks, and while Chez Scheme provides a similar `compute-composition` function, the `dump-gc-stats` function is more useful in cases where the relevant root object is not apparent; we found it simplest to extend Chez Scheme's garbage collector to more directly support `dump-gc-stats`. Racket's *phantom byte strings* provide a way to tie external, finalized allocation to Scheme objects for the purpose of memory accounting and triggering garbage collections; adding phantom byte strings to Chez Scheme was straightforward. Racket's garbage collector supports an incremental mode, which is particularly useful for classroom exercises that involve interactive games, but we do without it for now.

4.7 Foreign-Function Interface

Interacting with C-implemented libraries in modern Racket is driven from Racket code using a foreign-function interface (Barzilay and Orlovsky 2004), as opposed to driven by glue code that is written in C. This evolution means that Racket looks similar to Chez Scheme in its foreign-function interface (FFI). Still, a FFI tends to expose some of a host language's implementation details, and incompatibility between Racket and Racket CS seems inevitable. A typical Racket binding to foreign libraries needs adjustments to work in both implementations. Adapting bindings in the main distribution required only modest work, where the wrapped libraries include OpenSSL, libjpeg, libpng, Pango, Cairo, GTK+, Cocoa, Windows system libraries, and more.

Foreign-Pointer Representation and Object Locking. Chez Scheme distinguishes foreign pointers from Scheme objects, while Racket's notion of pointers for foreign calls allows a Racket byte string to be used interchangeably with a foreign pointer, and it also supports the allocation of raw arrays that are not constrained by a pointer-tagging regime. The FFI bridge for Racket CS can mostly manage these differences, but it must reject certain kinds of pointer coercions that cannot work on Chez Scheme. Another difference is that Racket's garbage collector supports an allocation arena of objects that will never be moved by garbage collection but will be reclaimed when they become inaccessible. Chez Scheme supports locking any allocated object, which prevents it both from moving and from being reclaimed. To work with both systems, Racket libraries must use an abstraction that fits both constraints.

C `struct` Arguments and Returns, Foreign-Thread Activation, and Compare-and-Set. While Chez Scheme provides a rich set of features in its FFI, some corners were not yet covered, including support for C functions that have `struct` arguments and return values. Chez Scheme supports OS-level threads, but it was not yet set up to handle calls into Scheme from previously unregistered OS

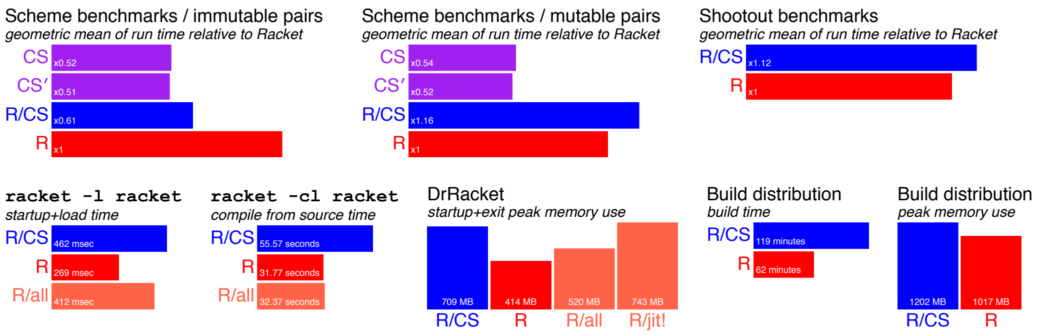


Fig. 4. Performance comparisons. Shorter is better. CS = unmodifed Chez Scheme, CS′ = modified Chez Scheme, R/CS = Racket CS, R = Racket, R/all = Racket with lazy bytecode loading disabled, R/jit! = JIT forced on all bytecode. Benchmarks results show a geometric mean of run times relative to Racket run times, taking the median of three runs for each benchmark. Benchmark sources are in the `racket-benchmarks` package in the Racket GitHub repository. Using Chez Scheme 9.5.3 commit 7df2fb2e77 at `github:cicso/ChezScheme`, modified as commit 6d05b70e86 at `github:racket/ChezScheme`, and Racket 7.3.0.3 as commit ff95f1860a at `github:racket/racket`. Measured on an Intel Core i7-2600 3.4GHz processor running 64-bit Linux.

threads, and no compare-and-set operation was exposed to support simple lock-free synchronization. Additions to cover those gaps have been merged into the main Chez Scheme implementation.

Exported C Interface. Both Racket and Chez Scheme provide an interface from C functions to call directly into the runtime system, instead of the other way around. Due to its history, Racket's exported C interface is large. Most of it could be mapped to Chez Scheme with the help of supporting Racket/Scheme code, but not all of it. We have made no effort to translate Racket's C API for Racket CS, and we currently have no plans to do so.

5 PERFORMANCE

Figure 4 compares a few facets of performance among Chez Scheme, Racket, and Racket CS.[3] The first two plots show relative performance for a set of commonly used Scheme benchmarks, and the results provide evidence that our changes to Chez Scheme have a negligible effect on its performance; Racket CS mostly maintains that performance, except where it introduces a distinct datatype to support mutable pairs (which Racket programmers rarely use). The third plot reports performance on benchmarks derived from the Computer Language Benchmarks Game[4] over its history; Racket CS performs less well here, where the benchmarks rely more heavily on the newly implemented Racket CS layers (represented by boxes toward the top-right of figure 1) that need further improvement. Similar to these benchmarks, production Racket programs tend to perform somewhere between slightly faster and 50% slower on Racket CS.

In full programs instead of benchmarks, the most perceptible performance differences come from longer compile times, larger code sizes, and longer load times—all of which are related to generating machine code instead of bytecode. The plots in the bottom row of figure 4 illustrate the differences and draw out some of the reasons. For example, load time in the current Racket implementation benefits significantly from lazy parsing of bytecode. Working with bytecode also

[3]We provide additional measurements as supplementary material.
[4]https://benchmarksgame-team.pages.debian.net/benchmarksgame/, formerly known as The Great Computer Language Shootout.

reduces the memory footprint of programs like DrRacket. Forcing both eager parsing of bytecode and JIT compilation closes some of the gap. The next-to-last plot in the figure shows a large difference in time required to build the Racket distribution from source; "cheap code" in the current Racket implementation has encouraged the generation of lots of code, often via macros, and the difference in build times reflects various compilation and code costs combined.

Overall, reduced end-to-end performance relative to the current Racket version prevents us from switching immediately to Racket CS as the default implementation. We expect to resolve the difference over time through some combination of further performance improvements and revised expectations from both Racket implementers and users.

6 STATUS AND OUTLOOK

After two years of work, Racket CS currently passes all but 26 of 813,950 tests in the 73k-line core Racket test suite (not counting 1,688 skipped tests of traditional Racket's bytecode optimizer and 2,841 skipped tests of single- and extended-precision floating point operations). The failures involve complex numbers and floating-point precision, error-message differences, and other corners that have little effect on end-user programs (based on bug reports, so far). Success rates are similar for other Racket libraries in the distribution as measured by our continuous-testing service. DrRacket works fully running on Chez Scheme, and Racket CS can build itself from source to full-distribution form.

If our task were "compile Racket to an existing target," then we would not have achieved such a high degree of compatibility. Unlike projects where the goal is to compile to the JVM, JavaScript, or WebAssembly, we have taken the liberty of modifying Chez Scheme to make it an easier target for Racket. Because we are willing to maintain Chez Scheme and any patches needed for Racket CS, and because that maintenance is preferable to working on Racket's existing implementation, this approach meets our goal of moving Racket to a more maintainable footing.

Our evidence for improved maintainability is anecdotal, but we consistently find working on Racket CS easier. For example, the new implementation of delimited continuations became useful almost immediately as an oracle to track down bugs in the previous, decade-old implementation. The new I/O implementation performed poorly at first, but we were able to refactor internal representations and protocols—building a new little language extension for objects, with just the right properties for the representations—in a matter of days, essentially catching up to the performance of the old implementation. Rewriting the macro expander in Racket (which was a prerequisite for porting to Chez Scheme) enlarged the number of people willing to modify the expander from 2 people over 16 years to 6 people over 2 years. Meanwhile, the fact that changes and patches to Chez Scheme were possible speaks to the flexibility and quality of its implementation.

Although our report has concentrated on the obstacles to building Racket on Chez Scheme, the benefits were far more numerous. The key benefit is starting with a robust core for a functional language: closures, compact data representations with full arithmetic, continuations bounded only by heap size, proper handling of tail calls, precise liveness for variables with safe-for-space optimizations, and compilation to high-quality machine code. Racket also relies on access to unsafe operations—to support external optimizations, which are sometimes driven by Typed Racket—plus a capable and convenient foreign-function interface. With the basics taken care of, we were able to concentrate on the details.

ACKNOWLEDGMENTS

The authors are supported by grants from National Science Foundation, Consejo Nacional de Investigaciones Científicas y Técnicas, Universidad de Buenos Aires, and Agencia Nacional de Promoción Científica y Tecnológica, including grants NSF-1823244, PCB No. 2013-1401PCB, UBACYT 20020150100157BA, and PICT No. 201-0381.

REFERENCES

Michael D. Adams. Flow-Sensitive Control-Flow Analysis in Linear-Log Time. PhD dissertation, Indiana University, 2013.

Leif Andersen, Vincent St-Amour, Jan Vitek, and Matthias Felleisen. Feature-Specific Profiling. *Transactions on Programming Languages and Systems* 41(1), 2019.

Eli Barzilay and Dmitry Orlovsky. Foreign Interface for PLT Scheme. In *Proc. Scheme and Functional Programming*, 2004.

Brent W. Benson Jr. libscheme: Scheme as a C Library. In *Proc. USENIX Symposium on Very High Level Languages*, 1994.

John Clements and Matthias Felleisen. A Tail-Recursive Machine with Stack Inspection. *Transactions on Programming Languages and Systems* 26(6), pp. 1029–1052, 2004.

John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an Algebraic Stepper. In *Proc. European Symposium on Programming*, 2001.

Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proc. Lisp and Functional Programming*, 1990.

R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a Generation-Based Garbage Collector. In *Proc. Programming Language Design and Implementation*, 1993.

R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. A Monadic Framework for Delimited Continuations. *Journal of Functional Programming* 17(6), pp. 687–730, 2007.

Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming Languages as Operating Systems (or Revenge of the Son of the Lisp Machine). In *Proc. International Conference on Functional Programming*, 1999.

Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding Delimited and Composable Control to a Production Programming Enviornment. In *Proc. International Conference on Functional Programming*, 2007.

Barry Hayes. Ephemerons: a New Finalization Mechanism. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 1997.

Robert Hieb, Kent Dybvig, and Claude W. Anderson , III. Subcontinuations. *Lisp and Symbolic Computation* 7(1), pp. 83–110, 1994.

Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Proc. Principles and Practice of Parallel Programming*, 1990.

Xiangqi Li and Matthew Flatt. Debugging with Domain-Specific Events via Macros. In *Proc. Software Language Engineering*, 2017.

John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

Julian Smart. *User Manual for wxWindows 1.63: a Portable C++ Toolkit.* 1995. Note: wxWindows is now known as wxWidgets.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (Ed.). The Revised ^{6} Report on the Algorithmic Language Scheme. 2007.

T. Stevie Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 2012.

Adam Wick and Matthew Flatt. Memory Accounting without Partitions. In *Proc. International Symposium on Memory Management*, 2004.