Languages as Libraries

Sam Tobin-Hochstadt
Northeastern University

Vincent St-Amour
Northeastern University

Ryan Culpepper
University of Utah

Matthew Flatt
University of Utah

Matthias Felleisen
Northeastern University

Abstract

Programming language design benefits from constructs for extending the syntax and semantics of a host language. While C's string-based macros empower programmers to introduce notational shorthands, the parser-level macros of Lisp encourage experimentation with domain-specific languages. The Scheme programming language improves on Lisp with macros that respect lexical scope.

The design of Racket—a descendant of Scheme—goes even further with the introduction of a full-fledged interface to the static semantics of the language. A Racket extension programmer can thus add constructs that are indistinguishable from "native" notation, large and complex embedded domain-specific languages, and even optimizing transformations for the compiler backend. This power to experiment with language design has been used to create a series of sub-languages for programming with first-class classes and modules, numerous languages for implementing the Racket system, and the creation of a complete and fully integrated typed sister language to Racket's untyped base language.

This paper explains Racket's language extension API via an implementation of a small typed sister language. The new language provides a rich type system that accommodates the idioms of untyped Racket. Furthermore, modules in this typed language can safely exchange values with untyped modules. Last but not least, the implementation includes a type-based optimizer that achieves promising speedups. Although these extensions are complex, their Racket implementation is just a library, like any other library, requiring no changes to the Racket implementation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

1. Growing Many Languages

I need to design a language that can grow. — Guy Steele, 1998

Virtual machines inspire language experimentation. For example, the Java Virtual Machine and the .NET CLR attracted many implementors to port existing languages. Their goal was to benefit from the rich set of libraries and runtime facilities, such as garbage collectors and thread abstractions, that these platforms offer. Both platforms also inspired language design projects that wanted to experiment with new paradigms and to exploit existing frameworks; thus Clojure, a parallelism-oriented descendant of Lisp, and Scala, a multi-paradigm relative of Java, target the JVM, while F# is built atop .NET. In all of these cases, however, the platform is only a target, not a tool for growing languages. As a result, design experiments on these platforms remain costly, labor-intensive projects.

To follow Steele's advice on *growing a language* (1998) requires more than a reusable virtual machine and its libraries; it demands an extensible *host language* that supports *linguistic reuse* (Krishnamurthi 2001). Thus, a derived language should be able to reuse the scoping mechanisms of the host language. Similarly, if the host language offers namespace management for identifiers, a language designer should have the freedom to lift that management into an experimental language.

Providing such freedoms to designers demands a range of extension mechanisms. A programmer may wish to manipulate the surface syntax and the AST, interpose a new context-sensitive static semantics, and communicate the results of the static semantics to the backend. Best of all, this kind of work can proceed without any changes to the host language, so that the resulting design is essentially a library that supplements the existing compiler.

In this paper, we present the Racket[1] (Flatt and PLT 2010) platform, which combines a virtual machine and JIT compiler with a programming language that supports extension mechanisms for all phases of language implementation. With Racket's arsenal of extension mechanisms, a programmer may change all aspects of a language: lexicographic and parsed notation, the static semantics, module linking, and optimizations. The Racket development team has exploited this extensibility for the construction of two dozen frequently used languages, e.g., language extensions for classes (Flatt et al. 2006) and ML-style functors (Culpepper et al. 2005; Flatt and Felleisen 1998), another for creating interactive web server applications (Krishnamurthi et al. 2007), two languages implementing logic programming[2] and a lazy variant of Racket (Barzilay and Clements 2005). This paper is itself a program in Racket's documentation language (Flatt et al. 2009).

Racket derives its power from a carefully designed revision of a Scheme-style macro systems. The key is to make language choice specific to each module. That is, individual modules of a program can be implemented in different languages, where the language implementation has complete control over the syntax and semantics of the module. The language implementation can reuse as much of the base language as desired, and it can export as much of the module's internals as desired. In particular, languages can reuse Racket's macro facilities, which is supported with a mechanism for locally expanding macros into attributed ASTs.

* A preliminary version of this work appeared at the Workshop on Scheme and Functional Programming (Culpepper et al. 2007). This work has been supported by the DARPA, Mozilla Foundation, NSERC, and NSF.

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06...$5.00

[1] Formerly known as PLT Scheme.

[2] Schelog (1993) `http://docs.racket-lang.org/racklog`
Datalog (2010) `http://docs.racket-lang.org/datalog`

For simplicity, we demonstrate the idea of true language extensibility via a single example: the Typed Racket implementation. Typed Racket (Tobin-Hochstadt and Felleisen 2008) is a statically typed sister language of Racket that is designed to support the gradual porting of untyped programs. Its type system accommodates the idioms of Racket, its modules can exchange values with untyped modules, and it has a type-driven optimizer—all implemented as plain Racket libraries.

The remainder of this paper starts with a description of Racket's language extension facilities and the challenges posed by the Typed Racket extension. Following these background sections, the paper shows how we use syntactic extension to introduce Typed Racket notation; how we splice the type checker into the tool chain; how we enable inter-language linking; and how we add realistic optimizing transformations via a library. Finally, we relate our approach to existing facilities for meta-programming with static semantics as well as extensible compiler projects.

2. Language Extension in Racket

Racket provides a range of facilities for language extension. In this section, we outline the most relevant background as well as extension mechanisms provided with Racket.

2.1 Macros

From Lisp (Steele Jr. 1994) and Scheme (Sperber et al. 2009), Racket takes hygienic macros as the basis of its syntactic extension mechanisms. In Racket, macros are functions from syntax to syntax, which are executed at compile time. During compilation, Racket's macro expander recursively traverses the input syntax. When it reaches the use of a macro, it runs the associated function, the *transformer*, and continues by traversing the result.

Macros may use arbitrary Racket libraries and primitives to compute the result syntax. The following macro queries the system clock at compile time:

```
(define-syntax (when-compiled stx)
 (with-syntax ([ct (current-seconds)])
   #'ct))
```

This macro computes, at compile time, the current time in seconds, and uses the `with-syntax` form to bind the identifier `ct` to a *syntax object* representing this number. Syntax objects are the ASTs of Racket, and contain syntactic data as well as metadata such as source location information. The `#'` form (analogous to `'` for lists) constructs syntax objects and can refer to identifiers bound by `with-syntax` in its context. In the following function, we use the `when-compiled` macro:

```
(define (how-long-ago?)
  (- (current-seconds) (when-compiled)))
```

Since the `when-compiled` form expands into the current date in seconds *at the time the form is compiled* and since the value of `(current-seconds)` continues to change, the value produced by `how-long-ago?` continually increases

```
> (how-long-ago?)
0
> (sleep 1)
> (how-long-ago?)
1
```

Most macros generate new expressions based on their input:

```
(define-syntax (do-10-times stx)
  (syntax-parse stx
    [(do-10-times body:expr ...)
     #'(for ([i (in-range 10)])
         body ...)]))
> (do-10-times (display "*") (display "#"))
*#*#*#*#*#*#*#*#*#*#
```

The `do-10-times` macro consumes a sequence of expressions and produces code that runs these expressions 10 times. To this end, it first decomposes its input, `stx`, with the `syntax-parse` form (Culpepper and Felleisen 2010), a pattern matcher designed for implementing macros. The pattern requires a sequence of subexpressions, indicated with `...`, named `body`, each of which is constrained to be an `expr`. The resulting syntax is a `for` loop that contains the `body` expressions. Thanks to macro hygiene, if the `body`s use the variable `i`, it is *not* interfered with by the use of `i` in the `for` loop.

2.2 Manipulating Syntax Objects

Since syntax objects are Racket's primary compile-time data structure, roughly analagous to the ASTs of conventional languages, syntax objects come with a rich API.

Constructors and Accessors Besides `with-syntax` and `#'` for constructing and manipulating syntax objects, the remainder of the paper also uses the following forms and functions:

- `syntax->list` converts a non-atomic syntax object to a list;
- `free-identifier=?` compares two identifiers to determine if they refer to the same binding;
- the `` #` `` and `#,` forms implement Lisp's quasiquote and unquote for syntax object construction.

Syntax properties Racket macros can add out-of-band information to syntax objects, e.g., source locations, dubbed *syntax properties*. The `syntax-property-put` and `syntax-property-get` procedures attach and retrieve arbitrary key-value pairs on syntax objects, which are preserved by the macro expander. Thus syntactic extensions may communicate with each other without interfering with each other.

Local Expansion The `local-expand` procedure expands a syntax object to *core Racket*. This explicitly specified core language consists of approximately 20 primitive syntactic forms that implement Racket; see figure 1 for a subset of this grammar. Using `local-expand`, a language extension may analyze an arbitrary expression, even if that expression uses macros. For example, the following macro requires that its argument be a λ expression:

```
(define-syntax (only-λ stx)
  (syntax-parse stx
    [(_ arg:expr)
     (define c
       (local-expand #'arg 'expression '()))
     (define k (first (syntax->list c)))
     (if (free-identifier=? #'#%plain-lambda k)
         c
         (error "not λ"))]))
```

The `only-λ` macro uses `local-expand` on the `arg` subexpression to fully expand it:

```
> (only-λ (λ (x) x))
#<procedure>
> (only-λ 7)
not λ
```

If we add a definition that makes `function` the same as λ, we still get the correct behavior.

```
> (only-λ (function (x) x))
#<procedure>
```

The `only-λ` macro can see through the use of `function` because of the use of `local-expand`.

2.3 Modules and Languages

Racket provides a first-order module system that supports import and export of language extensions (Flatt 2002). For the purposes

```
  mod-form = expr
           | (#%provide provide-spec)
           | (define-values (id) expr)
           | ...

      expr = id
           | (#%plain-lambda (id ...)
                expr ...+)
           | (if expr expr expr)
           | (quote datum)
           | (#%plain-app expr ...+)
           | ...
```

Figure 1: Racket's core forms (abbreviated)

of this paper, two aspects of the module system are crucial. First, modules can export static bindings, such as macros, as well as value bindings, such as procedures, without requiring clients to distinguish between them. Thus, value bindings can be replaced with static bindings without breaking clients. Second, each module is compiled with a *fresh store*. That is, mutations to state created during one compilation do not affect the results of other compilations.

Every module specifies—in the first line of the module—the language it is written in. For example,

```
    #lang racket
```

specifies `racket` as the module's language, and

```
    #lang datalog
```

specifies `datalog` instead, a language extension with different lexical syntax, static and dynamic semantics from plain Racket.

For the purposes of this paper, a *language* *L* is a library that provides two linguistic features:

- a set of bindings for *L*, including both syntactic forms such as `define` and values such as `+`, which constitute the base environment of modules written in the language, and
- a binding named `#%module-begin`, which is used to implement the whole-module semantics of *L*.

The `#%module-begin` form is applied to the entire module before the macro expander takes over. For example, here is the `#%module-begin` form for the `count` language:

```
(define-syntax (#%module-begin stx)
  (syntax-parse stx
    [(#%module-begin body ...)
     #`(#%plain-module-begin
        (printf "Found ~a expressions."
                #,(length
                   (syntax->list
                     #'(body ...))))
        body ...)]))
```

The `#%plain-module-begin` form is the base module wrapper, adding no new static semantics.

When this language is used, it prints the number of top-level expressions in the program, then runs the program as usual. For example, consider the following module written in `count`:

```
  #lang count
  (printf "*~a" (+ 1 2))
  (printf "*~a" (- 4 3))
```

When run, this module prints:

```
  Found 2 expressions.*3*1.
```

The `#%module-begin` language mechanism allows a language author to implement arbitrary new whole-module static semantics using macro rewriting.

3. Typed Racket as a Library

Typed Racket combines a type system for enriching Racket modules with sound type information and a mechanism for linking typed and untyped modules. Its implementation not only benefits from, but demands, as much linguistic reuse as possible. After all, Typed Racket must implement the same semantics as Racket plus purely syntactic type checking; there is no other way to provide a smooth migration path that turns Racket programs into Typed Racket programs on a module-by-module basis. And the best way to implement the same semantics is to share the compiler.

At the same time, linguistic reuse poses novel challenges in addition to those faced by every implementer of a typed language. In this section, we explain these challenges with examples; in the remainder of the paper we explain our solutions. As for the particular challenges, we show how to:

1. annotate bindings with type specifications;
2. check type correctness in a context-sensitive fashion;
3. type check programs written in an extensible language;
4. integrate type checking with separate compilation of modules;
5. provide safe interaction with untyped modules; and
6. optimize programs based on type information.

3.1 Layering Types on an Untyped Language

Consider this trivial Typed Racket program:

```
  #lang typed/racket
  (define: x : Number 3)
```

It specifies that `x` has type `Number`. To implement `define:`, Typed Racket *reuses* the plain `define` provided by Racket. Thus we must associate the type declaration for `x` out-of-band.

Further, although modern languages come with a wide variety of syntactic forms, most can be reduced to simpler forms via rewrite rules implemented as macros, e.g.:

```
  (define-syntax (let: stx)
    (syntax-parse stx
      [(let ([x:id : T rhs:expr]) body:expr)
       #'((λ: ([x : T]) body) rhs)]))
```

For Typed Racket, these rewriting rules must preserve the specified type information without interfering with the implementation of the typechecker.

These are two instances of Typed Racket's linguistic reuse. This linguistic reuse is comprehensive through all layers of Typed Racket. Modules in Typed Racket are simply Racket modules. The same is true of functions and variables, which map directly to their Racket equivalents. At the intermediate level, Typed Racket reuses Racket's binding forms, such as `define` and λ, with additions for the specification of types as described above. At the lowest level, runtime values are shared between Racket and Typed Racket.

Returning to our example, the plain Racket definition form is:

```
    (define x 3)
```

There is no place in such a definition for a type annotation. Hence, the macro must store the type information out-of-band. Fortunately, syntax properties allow for `define:` to record precisely such additional information:

```
  (define-syntax (define: stx)
    (syntax-parse stx
      [(define: name:id : ty rhs:expr)
       (with-syntax
           ([ann-name
             (syntax-property-put
               #'name 'type-annotation #'ty)])
         #'(define ann-name rhs))]))
```

Here, ann-name is the original name, but with a syntax property indicating that it is annotated with the type ty. With this implementation, later stages of processing can read the type annotation from the binding, but the type annotation does not affect the behavior of Racket's define, allowing reuse as desired.

3.2 Challenges

Next we turn to the challenges of other processing phases.

Context-sensitive Checking Type checking is inherently context-sensitive. For example, this Typed Racket program:

```
#lang typed/racket
(: f (Number -> Number))
(define (f z) (sqrt (* 2 z)))
(f 7)
```

relies on contextual information about the type of f when checking the function application (f 7). Tracking this information demands an implementation that typechecks the entire module, rather than just the syntax available at a particular program point; in other words, it is a whole-module analysis.

Checking an Extended Language Typed Racket programmers expect to use the numerous libraries provided by Racket, many of which provide syntactic abstractions. For example, this module uses match, a syntactic form implemented in a library written in plain Racket, rather than a primitive form as in ML or Haskell, but nonetheless indistinguishable from a language primitive:

```
#lang racket
(match (list 1 2 3)
  [(list x y z)
   (+ x y z)])
```

Naturally, Typed Racket programmers want to reuse such convenient libraries, ideally without modifications, so that the above becomes a valid Typed Racket module by using typed/racket.

Therefore, we must either extend our typechecker to handle match, or translate match into a simpler form that the typechecker understands. The former solution works for existing language extensions, but in Racket, programmers can write new language extensions at any time, meaning that Typed Racket cannot possibly contain a catalog of all of them.

Supporting Modular Programs Racket programs consist of multiple modules; Typed Racket therefore supports modules as well. The types of bindings defined in one module must be accessible in other typed modules without the need to repeat type declarations. For example:

```
#lang typed/racket ;; module server
(: add-5 : Integer -> Integer)
(define (add-5 x) (+ x 5))
;; export ‘add-5’ from the module
(provide add-5)

#lang typed/racket ;; module client
(require server)
(add-5 7) ;; type checks correctly
```

Because modules are compiled separately, the server module is compiled before the client module, but must communicate the static information about the type of add-5 to client. Therefore, compiling a Typed Racket module must produce a *persistent* record of the types of exported bindings.

Integrating with Untyped Modules Racket comes with hundreds of thousands of lines of libraries, almost all of which are written without types. Effective use of Typed Racket, therefore, requires interoperation with untyped modules. Typed Racket supports such interoperation in both directions. First, typed modules can import untyped bindings by specifying their types:

```
#lang typed/racket
(require/typed racket/file
  [file->lines (Path -> (Listof String))])
(file->lines "/etc/passwd")
```

Second, untyped modules can import typed bindings:

```
#lang racket ;; client
(require server)
(add-5 12) ;; safe use
(add-5 "bad") ;; unsafe use
```

Typed Racket must protect its soundness invariants and check value flow across the boundary between typed and untyped programs. It must also avoid, however, imposing unnecessary dynamic checks between typed modules.

Optimizing with Type Information Type information enables a wide variety of optimizations. Tag checking, ubiquitous in untyped Racket programs, is unnecessary in typed programs. For example, this program need not check that the argument to first is a pair:

```
#lang typed/racket
(: p : (List Number Number Number))
(define p (list 1 2 3))
(first p)
```

Of course, with type information, more significant optimizations are also possible. The following loop is transformed into one that performs only machine-level floating-point computation:

```
#lang typed/racket
(: count : Float-Complex -> Integer)
(define (count f)
  (let loop ([f f])
    (if (< (magnitude f) 0.001)
        0
        (add1 (loop (/ f 2.0+2.0i))))))
```

In particular, the type information should assist the compiler back-end with the treatment of complex and floating-point numbers.

4. A Single-Module Typechecker

This section presents the single-module core of a simply-typed version of Typed Racket. It concludes with an explanation of how to scale the system to the full typechecker of Typed Racket.

4.1 An Example

Assume our language is available from the simple-type library. Thus, we can write modules like the following:

```
#lang simple-type
(define x : Integer 1)
(define y : Integer 2)
(define (f [z : Integer]) : Integer
  (* x (+ y z)))
```

Such a module is first fully expanded to the appropriate core forms and then typechecked. Thus, type-incorrect definitions and expressions signal compile-time errors.

```
(define w : Integer 3.7)
```

typecheck: wrong type in: 3.7

Modules with type errors are not executable.

4.2 Wiring Up the Typechecker

Typechecking a module is a context-sensitive process and therefore demands a whole-module analysis via #%module-begin.

Since Racket provides a rich mechanism for syntactic extension, many important language features are implemented as language extensions. Major examples are pattern matching, keyword arguments, and even simple conditional forms such as case and cond. Providing appropriate type rules for every syntactic extension, however, is clearly impossible, because programmers can invent new extensions at any time. Instead, we consider only the

```
(define-syntax (#%module-begin stx)
  (syntax-parse stx
    [(_ forms ...)
     (with-syntax ([(_ core-forms ...)
                    (local-expand #'(#%plain-module-begin forms ...) 'module-begin '())])
       (for-each typecheck (syntax->list #'(core-forms ...)))
       #'(#%plain-module-begin core-forms ...))]))
```

Figure 2: The Top-level Driver

```
(define (typecheck t [check #f])
  (define the-type
    (syntax-parse t
      [v:identifier (lookup-type #'v)]
      [(quote n:number) (cond [(exact-integer? (syntax-e #'n)) IntT]
                              [(flonum? (syntax-e #'n)) FloatT]
                              [else NumberT])]
      [(if e1 e2 e3)
       (typecheck #'e1 BooleanT)
       (unless (equal? (typecheck #'e2) (typecheck #'e3))
         (type-error "if branches must agree"))
       (typecheck #'e3)]
      [(#%plain-lambda formals body:expr)
       (define formal-types (map type-of (syntax->list #'formals)))
       (for-each add-type! (syntax->list #'formals) formal-types)
       (make-fun-type formal-types (typecheck #'body))]
      [(#%plain-app op . args)
       (define argtys (map typecheck (syntax->list #'args)))
       (match (typecheck #'op)
         [(struct fun-type (formals ret))
          (unless (and (= (length argtys) (length formals)) (andmap subtype argtys formals))
            (type-error "wrong argument types" t))
          ret]
         [t (type-error "not a function type" #'op)])]
      [(define-values (id) rhs)
       (add-type! #'id (type-of #'id)) (typecheck #'rhs (type-of #'id))]))
  (when (and check (not (subtype the-type check))) (type-error "wrong type" t))
  the-type)
;; get the type of a binding
(define (type-of id)
  (unless (syntax-property-get id 'type) (type-error "untyped variable" id))
  (parse-type (syntax-property-get id 'type)))
```

Figure 3: The Typechecker

small fixed set of core forms of figure 1, and reduce all other forms to these *before* type checking.

Given an implementation of the typechecker, we must connect it to the program so that it receives appropriate fully-expanded syntax objects as input. The basic driver is given in figure 2. The driver is straightforward, performing only two functions. Once we have fully expanded the body of the module, we typecheck each form in turn. The typechecker raises an error if it encounters an untypable form. Finally, we construct the output module from new core forms, thus avoiding a re-expansion of the input.

The strategy of reducing syntactic sugar to core forms is common in many other languages, and even specified in the standards for ML (Milner et al. 1997) and Haskell (Marlow 2010). In a language with syntactic extension, we need more sophisticated support from the system to implement this strategy, and that support is provided in Racket by `local-expand`.

For successful typechecking, we must also provide an initial environment. The initial environment specifies types for any identifiers that the language provides, such as +, as well as the initial type names. Finally, we provide the `define` and λ binding forms that attach the appropriate syntax properties for type annotations to the bound variables, as described in section 2.1.

4.3 Typechecking Syntax

Figure 1 displays the grammar for our simple language. It is a subset of the core forms of full Racket. Modules consist of a sequence of *mod-form*s, which are either expressions or definitions.

Figure 3 specifies the typechecker for this core language. The `typecheck` function takes a term and an optional result type. Each clause in the `syntax-parse` expression considers one of the core forms described in figure 1.

Two aspects of the typechecker are distinctive. First, the type environment uses a mutable table mapping identifiers to types based on their binding; the table is accessed with `lookup-type` and updated with `add-type!`. Shadowing is impossible because identifiers in fully-expanded Racket programs are unique with respect to the entire program. We update the type environment in the clauses for both `#%plain-lambda` and `define-values`. Using an identifier-keyed table allows reuse of the Racket binding structure without having to reimplement variable renaming or environments.

The second distinctive feature of the typechecker is the `type-of` function. It reads the syntax properties attached by forms such as `define:` in section 3.1 with a known key to determine the type the user has added to each binding position.

4.4 Scaling to Typed Racket

While this module-level typechecker is simple, the full implementation for Typed Racket employs the same strategy. The important differences concern mutual recursion and complex definition forms. Mutual recursion is implemented with a two-pass typechecker: the first pass collects definitions with their types, and the second pass checks individual expressions in this type context. Complex declarations, such as the definition of new types, are also handled in the first pass. They are recognized by the typechecker, and the appropriate bindings and types are added to the relevant environments.

Of course, the Typed Racket type system is much more complex (Strickland et al. 2009; Tobin-Hochstadt and Felleisen 2010) than the one we have implemented here, but that complexity does not require modifications to the structure of the implementation—it is encapsulated in the behavior of `typecheck` on the core forms.

5. Modular Typed Programs

The typechecker in section 4 deals only with individual modules. To deal with multiple modules, we must both propagate type information between typed modules, and also persist type information in compiled code to support separate compilation.

For the first point, we reuse some Racket infrastruture. Namespace management in a modular language is a complex problem, and one that Racket already solves. In particular, identifiers in Racket are given globally fresh names that are stable across modules during the expansion process. Since our type environment is keyed by identifiers, type environment lookup reuses and respects Racket's scoping. An identifier imported from one module maintains its identity in the importing module, and therefore the typechecker is able to look up the appropriate type for the binding.

The second step is to maintain the type environment across compilations. Since each module is compiled in a separate and fresh store, mutations to the type environment do not persist between compilations. Therefore, Typed Racket must incorporate the type environment into the residual program, because that is the only persistent result of compilation. Our strategy — due to Flatt (2002) — is to include code in the resulting module that populates the type environment every time the module is required.

In our example system, we implement this by adding a single rewriting pass to the `#%module-begin` form for Typed Racket. Expressions and definitions are left alone; an appropriate compile-time declaration is added for each export:

```
(provide n) ; The original export
; ==> (is rewritten into)
(with-syntax ([t (serialize (type-of #'n))])
 #`(begin
     (#%provide n) ; The core export form
     (begin-for-syntax
       (add-type! #'n t))))
```

The resulting code uses `#%provide` to maintain the original export. The type declaration is wrapped in `begin-for-syntax`, meaning that it is executed at compile time to declare that `n` is mapped to the serialization of its type in the type environment.

```
(define-syntax (require/typed stx)
  (syntax-parse stx
    [(_ module [id ty])
     #`(begin-ignored
        ; Stage 1
        (require (only-in module
                          [id unsafe-id]))
        ; Stage 2
        (begin-for-syntax
         (add-type! #'id (parse-type #'ty)))
        ; Stage 3
        (define id
          (contract
           #,(type->contract
              (parse-type #'ty))
           unsafe-id
           (quote module)
           'typed-module)))]))
```

Figure 4: Import of Untyped Code

6. Safe Cross-Module Integration

A module in Typed Racket should have access to the large collection of untyped libraries in Racket. Conversely, our intention to support gradual refactoring of untyped into typed systems demands that typed modules can export bindings to untyped modules. However, untyped programs are potentially dangerous to the invariants of typed modules. To protect these invariants, we automatically generate run-time contracts from the types of imported and exported bindings (Tobin-Hochstadt and Felleisen 2006).

However, a large library of untyped modules is useless if each must be modified to work with typed modules. Therefore, our implementation must automatically wrap imports from untyped code, and protect exports to untyped code, *without* requiring changes to untyped modules. Further, communication between typed modules should not involve extra contract checks, since these invariants are enforced statically.

6.1 Imports from Untyped Modules

Typed Racket requires the programmer to specify the types of imports from untyped modules:

```
  (require/typed file/md5
    [md5 (Bytes -> Bytes)])
```

This specification imports the `md5` procedure, which computes the MD5 hash of a byte sequence. The procedure can be used in typed code with the specified type, and the type is converted to a contract and attached to the procedure on import. This translation and interposition is implemented in the `require/typed` form; see figure 4 for an implementation for our `simple-type` language.

The implementation of `require/typed` works in three stages. Stage 1 imports the specified identifier, `id` from `module` under the new name `unsafe-id`. Stage 2 parses and adds the specified type to the table of types with `add-type!`. Finally, stage 3 defines the identifier `id` as a wrapper around `unsafe-id`. The wrapper is a generated contract and establishes a dynamically enforced agreement between the original module (named `module`) and the typed module (with the placeholder name).[3] The entire output is wrapped in `begin-ignored` so that the type checker does not process this meta-information.

In our example, we would now be able to use the `md5` function in typed code according to the specified type, getting a static type error if `md5` is applied to a number, for example. Conversely, if the

[3] In practice, type checking renders the domain contract superfluous.

`file/md5` library fails to return a byte string value, a dynamic contract error is produced, avoiding the possibility that the typed module might end up with a value that it did not expect.

6.2 Exports to Untyped Modules

Unlike imports into a typed module, exports from such a module pose a serious problem because the Typed Racket language implementation is not necessarily in control of the use site. After all, an exported identifier may be used in both typed *and* untyped contexts. Since typed modules statically verify that uses of typed identifiers accord with their types, no contracts are necessary for exports to such modules. In contrast, exports from typed to untyped modules require the insertion of dynamic checks to ensure type safety.

To implement this behavior without cloning every module, we adopt a novel two-stage module compilation strategy. First, each export is replaced with an indirection that chooses whether to reference the contracted or plain version of the exported binding. Second, the compilation of *typed* modules sets a flag inside the `#%module-begin` transformer before expansion of the module's contents; the exported indirections choose which version to reference based on this flag. Since each module is compiled with a fresh state, this flag is only set during the compilation of typed modules—untyped modules have no way to access it. Therefore, during the compilation of untyped modules, the export indirections correctly choose the contract-protected version of bindings. The compilation of typed modules, in contrast, see the set version of the flag and are able to use the uncontracted versions of bindings.

Implementation Exported identifiers are rewritten in the same stages as imported identifiers. Since this rewriting occurs *after* typechecking, however, it is performed by the `#%module-begin` form, just as declarations are added to exports:

```
(#%provide n) ; An export of n
; ==> (is rewritten to)
#`(begin
    ... the declaration from section 5 ...
    ; Stage 1
    (define defensive-n
      (contract
       #,(type->contract (typecheck #'n))
       n 'typed-module 'untyped-module))
    ; Stage 2
    (define-syntax (export-n stx)
      (if (unbox typed-context?)
          #'n #'defensive-n))
    ; Stage 3
    (provide (rename-out [export-n n])))
```

First, we define a `defensive` version of the identifier `n`, which uses a contract generated from the type of `n`. Second, we define an `export-n` version of the identifier `n`, which selects between `n` and `defensive-n` depenending on the value of `typed-context?`. Third, we provide `export-n` under the name `n`, making the indirection transparent to clients of the typed module.

The second part comes in the definition of the language:

```
(define-syntax (#%module-begin stx)
  (set-box! typed-context? #t)
  ... check and transform stx, as in figure 2 ...
  ... rewrite provides, as above ...)
```

The initial flag setting means that the expansion of the module, and in particular the expansion of the indirections imported from other typed modules, see the `typed-context?` flag as set to `#t`.

Finally, because the `typed-context?` flag is accessible only from the implementation of the `simple-type` language, it is simple to verify that the flag is only set to `#t` in the `#%module-begin` form. Therefore, the implementation can rely on this flag as an indicator, without the possibility that untyped code might be able to deceive the typechecker.

6.3 Scaling to Typed Racket

The full implementation of module integration in Typed Racket follows the strategy outlined. The major complication is the export of macros and other static information from typed modules. Since macros from typed modules can refer to internal identifiers not protected by contracts, expanding such macros in untyped modules could potentially allow untyped modules to violate the invariants of typed modules. Therefore, Typed Racket currently prevents macros defined in typed modules from escaping into untyped modules.

7. Optimization via Rewriting

Source-to-source transformations can express large classes of optimizations, which makes it possible to apply them in the front end of the compiler. With the right language extension mechanisms, we can express these transformations as libraries and use them to build competitive optimizers.

7.1 Compiler architecture

Since Typed Racket is built as a language extension of Racket and the Racket compiler is for an untyped language, Typed Racket has to apply typed optimizations before handing programs to the Racket compiler. In contrast, compilers for typed languages can keep track of types across all phases in the compiler and use them to guide optimizations. Hence, Typed Racket features a type-driven optimization pass after typechecking. This optimization pass transforms the code that the front end of Typed Racket generates, using the validated and still accessible type information.

To support realistic optimizers as libraries, the host language must provide ways for language extensions to communicate their results to the compiler's backend. As part of its language extension features, Racket exposes unsafe type-specialized primitives.[4] For instance, the `unsafe-fl+` primitive adds two floating-point numbers, but has undefined behavior when applied to anything else. These type-specialized primitives are more efficient than their generic equivalents; not only do these primitives avoid the run-time dispatch of generic operations, they also serve as signals to the Racket code generator to guide its unboxing optimizations.

Typed Racket's optimizer generates code that uses these primitives. In figure 5, we show an excerpt from the optimizer that specializes floating-point operations using rewrite rules; the optimizer rewrites uses of generic arithmetic operations on floating-point numbers to specialized operations.

7.2 Scaling to Typed Racket

Typed Racket uses the same techniques as the simple optimizer presented here, but applies a wider range of optimizations. It supports a number of floating-point specialization transformations, eliminates tag-checking made redundant by the typechecker and performs arity raising on functions with complex number arguments.

7.3 Results

Untyped Racket is already competitive among optimizing Scheme compilers. The addition of a type-driven optimizer makes a noticeable difference and makes it an even more serious contender.

We show the impact of our optimizer on micro-benchmarks taken from the Gabriel (1985) and Larceny (Clinger and Hansen 1994) benchmark suites and the Computer Language Benchmark Game,[5] as well as on large benchmarks: the pseudoknot (Hartel

[4] Initially these primitives were provided because some programmers wanted to hand-optimize code.

[5] `http://shootout.alioth.debian.org`

```
(define (optimize t)
  (syntax-parse t
    [(#%plain-app op:id e1:expr e2:expr)
     (with-syntax ([new-op (if (and (equal? FloatT (type-of #'e1))
                                    (equal? FloatT (type-of #'e2)))
                               (cond [(free-identifier=? #'op #'+) #'unsafe-fl+]
                                     [(free-identifier=? #'op #'-) #'unsafe-fl-]
                                     [else #'op])
                              #'op)])
       #`(#%plain-app new-op #,(optimize #'e1) #,(optimize #'e2)))]
    ... structurally recur on the other forms ...) )
```

Figure 5: The Optimizer

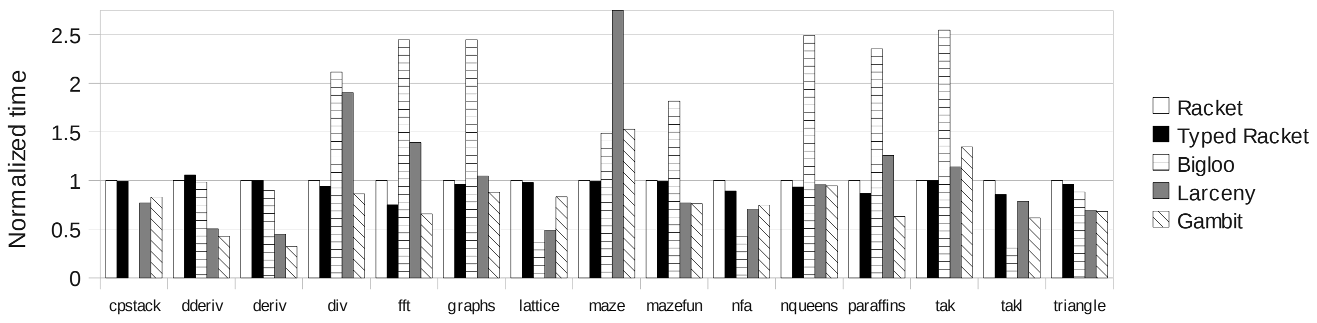

Figure 6: Results on the Gabriel and Larceny benchmarks (smaller is better)

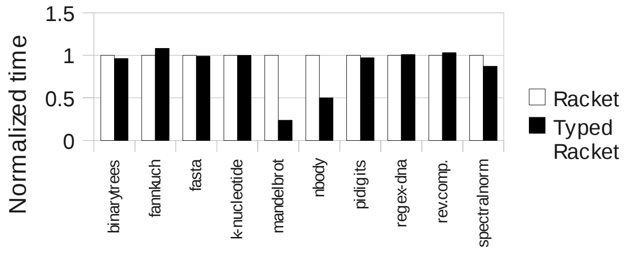

Figure 7: Results on the Computer Language Benchmark Game (smaller is better)

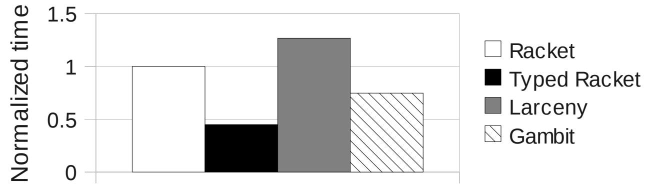

Figure 8: Results on pseudoknot (smaller is better)

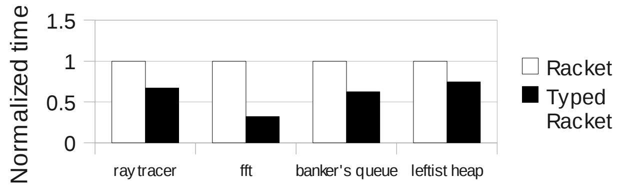

Figure 9: Results on large benchmarks (smaller is better)

et al. 1996) floating-point benchmark, a ray tracer, an industrial strength FFT and the implementation of two purely functional data structures (Prashanth and Tobin-Hochstadt 2010). Each benchmark comes in two versions: the original version and a translation to Typed Racket. The typed versions have type annotations and extra predicates where required to typecheck the program.

The typed version runs in Racket 5.0.2 using our type-driven optimizer and the untyped version in Racket 5.0.2, Gambit 4.6.0 (Feeley and Miller 1990), Larceny 0.97 (Clinger and Hansen 1994) and Bigloo 3.5a (Serrano and Weis 1995), using the highest safe optimization settings in each case. Benchmarks from the Computer Language Benchmarks Game and our sample applications use Racket-specific features and cannot be measured with other Scheme compilers. Bigloo fails to compile the `cpstack` and pseudoknot benchmarks. All our results are the average of 20 runs on a Dell Optiplex GX270 running GNU/Linux with 1GB of memory.

These benchmarks fall into two categories: benchmarks where Racket is already competitive with other well-known optimizing Scheme compilers, and benchmarks where Racket does not perform as well. In some cases where Racket is slower than the competition, Typed Racket's optimizer helps bridge the gap. For instance, it is responsible for a 33% speedup on the `fft` benchmark and a 123% speedup on pseudoknot. The large applications benefit even more from our optimizer than the microbenchmarks.

The results presented in this section demonstrate (a) that the Racket compiler and runtime are already competitive with leading compilers for Scheme, and (b) that Typed Racket's optimizer, written entirely as a library, nonetheless provides noticeable speedups on both widely-used benchmarks and existing Racket programs.

8. Related Work

While many individual aspects of our extensible language have long histories, some are novel and so is the combination as a whole. In this section, we sketch the history of each of these aspects.

8.1 Extensible Static Semantics

Macros have a long history in Lisp and Scheme (Steele Jr. and Gabriel 1993; Dybvig et al. 1992; Kohlbecker 1986). Several aspects of Racket's macro system improve on prior techniques.

First, the `#%module-begin` macro offers control over the entire module. Many Lisp programmers have approximated it with explicit wrappers. Second, many Lisp systems provide analogues of `local-expand` under the names `expand` or `macroexpand`. These features are more limited, however. Most importantly, they do not compose with other macros (Culpepper and Felleisen 2010). Third, Chez Scheme (Dybvig 2009) provides a `define-property` form, which simulates identifier-keyed tables for the same purposes described in section 5. Chez Scheme's form is less general than Racket's: while `define-property` would support Typed Racket's maintenance of type environments, it would not work for the interoperability mechanism described in section 6.2.

Fisher and Shivers (2006; 2008) present a system for handling analysis of programs written in an extensible language. Specifically, their system can implement static analyses, such as type systems, on top of extensible languages, without requiring the analysis to know about every possible language extension. The system employs a dispatch mechanism so that each extension can answer the static analysis question about its own use. This approach is extensible and expressive, but inherently unsound, because language extensions may provide analysis results that disagree with their runtime behavior. Our solution using `local-expand` relies on the inference of high-level properties from core forms and respects soundness, though it imposes limits on some macro uses in language extensions.

In the functional-logic programming world, the Ciao Prolog system (Hermenegildo et al. 2008) comes close to Racket. It also allows programmers to annotate their programs with new static assertions and to control when these assertions are checked. This strategy accommodates customizable static semantics and enabling new optimizations. In contrast to our system, Ciao builds the assertion language into the compiler, which furthers tight integration with its existing static analyzers and compiler, but prevents users from developing truly new semantics and whole program checking.

The ArBB and Rapidmind (Ghuloum et al. 2010) tools take the approach of embedding languages for novel execution models within C++. In contrast with the approach we present, their embedding demands the use of the C++ type system and necessitates the creation of ad-hoc replacements for fundamental language features such as 'if' and 'for' statements, thus limiting linguistic reuse. Additionally, because individual constructs see only a portion of the program, the scope for an analysis is necessarily local, in contrast to Typed Racket which can check an entire module.

8.2 Extensible Compilers

Since compilers are valuable and complex pieces of software, many projects have designed extensible compilers. Extensibility means that programmers can add front-ends, backends, optimizations, analyses, or other plug-ins (Bravenboer and Visser 2004; Bachrach and Playford 2001; Cox et al. 2008; Nystrom et al. 2003). All of these systems differ significantly from our implementation of a language as a library. First, they require interoperation at the level of the compiler rather than the language. Language extensions in Racket are implemented as Racket programs, operate on Racket programs, and produce Racket programs; they operate without any reliance on the details of the underlying compiler or runtime. This higher level of abstraction means that the compiler can be modified without breaking existing extensions, and extensions need not depend on the specialized representations and analyses in the compiler implementation.

Second, extensible compilers are not libraries in the same sense as ordinary libraries. Instead, they are plug-ins to a different application—the compiler, which may be written in a different language using a different architecture. This means compiler plugins cannot take advantage of the same library distribution mechanisms, debugging mechanisms, tools, and the rest of the software infrastructure around the language. This difference makes them both more difficult to develop and to use for programmers, thus limiting language experimentation.

8.3 Rewriting-based Optimization

The specific optimization techniques described in section 7 are not new to Typed Racket. Kelsey (1989) presents a transformational compiler that expresses optimizations as rewritings from the source language, an extended continuation-passing style lambda calculus, to itself. These transformations are encoded inside the compiler, however, which places them off-limits for programmers.

The Glasgow Haskell Compiler (Peyton Jones 1996) also expresses optimizations as rewrite rules. Unlike Kelsey's work, GHC makes it possible for programmers to supplement these rules with their own. To preserve safety, the compiler enforces that all rewrite rules must preserve types; that is the result of applying a rewrite rule must be of the same type as the the original term. In contrast to Racket, the GHC rewrite rules are expressed in a limited domain-specific language.

It would be impossible to implement the arity-raising transformation mentioned in section 7.2 in the GHC framework; such optimizations are instead built-in to GHC.

9. Conclusion

In *Growing a Language*, Steele writes "if we grow the language in these few ways, then we will not need to grow it in a hundred other ways; the users can take on the rest of the task." In this paper, we have explained how Racket's small number of language extensibility mechanisms empower the application programmer to grow even a highly sophisticated language. While these mechanisms continue the long tradition of Lisp and Scheme macros, they also go far beyond macros. Most importantly, Racket programmers can use `#%module-begin` to create libraries that process modules in a context-sensitive manner—without interfering with macros. Further, the syntax system can use attributed ASTs to communicate out-of-band information. As a result, programmers can interpolate context-sensitive analyses and source-based optimizations where the latter may exploit the result of the former.

One running example, Typed Racket, illustrates the range of Racket's extensibility mechanism. Specifically, the implementation of Typed Racket covers almost all aspects of language experimentation, from the front end to the back end. As our benchmarks demonstrate, the current backend of Typed Racket delivers competitive performance. What reduces the performance from "highly competitive" to just "competitive" is the lack of Racket APIs for processing intermediate representations and/or JIT compiler information. Until such APIs are available, compiler optimizations for language extensions remain limited to source-level transformations.

Acknowledgments

Robby Findler and Shriram Krishnamurthi have contributed to this design over many years. Numerous members of the Northeastern PRL provided feedback on earlier drafts of this paper. Mitch Wand suggested the title.

Bibliography

Jonathan Bachrach and Keith Playford. The Java syntactic extender. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 31–42, 2001.

Eli Barzilay and John Clements. Laziness Without All the Hard Work. In *Proc. Works. Functional and Declarative Programming in Education*, pp. 9–13, 2005.

Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 365–383, 2004.

John Clements, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. Fostering Little Languages. *Dr. Dobb's Journal*, pp. 16–24, 2004.

William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label or a simple optimizing compiler for Scheme. In *Proc. Conf. on LISP and functional programming*, pp. 128–139, 1994.

Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proc. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 244–254, 2008.

Ryan Culpepper and Matthias Felleisen. Debugging hygienic macros. *Science of Computer Programming* 75(7), pp. 496–515, 2010.

Ryan Culpepper and Matthias Felleisen. Fortifying macros. In *Proc. International Conf. on Functional Programming*, pp. 235–246, 2010.

Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In *Proc. Conf. Generative Programming and Component Engineering*, pp. 373–388, 2005.

Ryan Culpepper, Sam Tobin-Hochstadt, and Matthias Felleisen. Advanced Macrology and the Implementation of Typed Scheme. In *Proc. Scheme and Functional Programming*, 2007.

R. Kent Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2009.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation* 5(4), pp. 295–326, 1992.

Marc Feeley and James S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proc. Conf. on LISP and Functional Programming*, pp. 119–130, 1990.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. Building Little Languages With Macros. *Dr. Dobb's Journal*, pp. 45–49, 2004.

David Fisher and Olin Shivers. Static semantics for syntax objects. In *Proc. International Conf. on Functional Programming*, pp. 111–121, 2006.

David Fisher and Olin Shivers. Building language towers with Ziggurat. *J. of Functional Programming* 18(5-6), pp. 707–780, 2008.

Matthew Flatt. Composable and compilable macros: you want it when? In *Proc. International Conf. on Functional Programming*, pp. 72–83, 2002.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: closing the book on ad-hoc documentation tools. In *Proc. International Conf. on Functional Programming*, pp. 109–120, 2009.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 236–248, 1998.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Proc. Asian Symp. on Programming Languages and Systems*, pp. 270–289, 2006.

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

Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press, 1985.

Anwar Ghuloum, Amanda Sharp, Noah Clemons, Stefanus Du Toit, Rama Malladi, Mukesh Gangadhar, Michael McCool, and Hans Pabst. Array Building Blocks: A Flexible Parallel Programming Model for Multicore and Many-Core Architectures. *Dr. Dobb's Journal*, 2010.

Pieter H. Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, Peter Baumann, Marcel Beemster, Emmanuel Chailloux, Christine H. Flood, Wolfgang Grieskamp, John H. G. van Groningen, Kevin Hammond, Bogumi Hausman, Melody Y. Ivory, Richard E. Jones, Jasper Kamperman, Peter Lee, Xavier Leroy, Rafael D. Lins, Sandra Loosemore, Niklas Röjemo, Manuel Serrano, Jean-Pierre Talpin, Jon Thackray, Stephen Thomas, Pum Walters, Pierre Weis, and E.P. Wentworth. Benchmarking implementations of functional languages with "pseudoknot" a float-intensive benchmark. *J. of Functional Programming* 6(4), pp. 621–655, 1996.

Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro Lopez, José F. Morales, and German Puebla. An overview of the Ciao multiparadigm language and program development environment and its design philosophy. In *Concurrency, Graphs and Models*. Springer-Verlag, pp. 209–237, 2008.

Richard Andrew Kelsey. *Compilation by Program Transformation*. PhD dissertation, Yale University, 1989.

Eugene Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD dissertation, Indiana University, 1986.

Shriram Krishnamurthi. *Linguistic Reuse*. PhD dissertation, Rice University, 2001.

Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computing* 20(4), pp. 431–460, 2007.

Simon Marlow. Haskell 2010 Language Report. 2010.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, Revised Edition*. MIT Press, 1997.

Nathaniel Nystrom, Michael Clarkson, and Andrew Myers. Polyglot: an extensible compiler framework for Java. In *Proc. International Conf. on Compiler Construction*, pp. 138–152, 2003.

Simon L Peyton Jones. Compiling Haskell by program transformation a report from the trenches. In *Proc. European Symp. on Programming*, pp. 18–44, 1996.

Hari Prashanth K R and Sam Tobin-Hochstadt. Functional data structures for Typed Racket. In *Proc. Works. Scheme and Functional Programming*, pp. 1–7, 2010.

Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Proc. Static Analysis Symp.*, pp. 366–381, 1995.

Michael Sperber, Matthew Flatt, Anton Van Straaten, R. Kent Dybvig, Robert Bruce Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme. *J. of Functional Programming* 19(S1), pp. 1–301, 2009.

Guy L. Steele Jr. *Common Lisp: The Language*. Second edition. Digital Press, 1994.

Guy L. Steele Jr. Growing a language, Keynote at OOPSLA 1998. *Higher-Order and Symbolic Computation* 12(3), pp. 221–236, 1999.

Guy L. Steele Jr. and Richard P. Gabriel. The evolution of Lisp. In *Proc. Conf. on History of Programming Languages*, pp. 231–270, 1993.

T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical Variable-Arity Polymorphism. In *Proc. European Symp. on Programming*, 2009.

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage refactoring: from scripts to programs. In *Proc. Dynamic Languages Symp.*, pp. 964–974, 2006.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Proc. Symp. on Principles of Programming Languages*, pp. 395–406, 2008.

Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proc. International Conf. on Functional Programming*, pp. 117–128, 2010.