The Racket Manifesto*

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi
Eli Barzilay, Jay McCarthy, Sam Tobin-Hochstadt

Abstract

The creation of a programming language calls for guiding principles that point the developers to goals. This article spells out the three basic principles behind the 20-year development of Racket. First, programming is about stating and solving problems, and this activity normally takes place in a context with its own language of discourse; good programmers ought to formulate this language as a programming language. Hence, *Racket is a programming language for creating new programming languages*. Second, by following this language-oriented approach to programming, systems become multi-lingual collections of interconnected components. Each language and component must be able to protect its specific invariants. In support, *Racket offers protection mechanisms to implement a full language spectrum*, from C-level bit manipulation to soundly typed extensions. Third, because Racket considers programming as problem solving in the correct language, *Racket also turns extra-linguistic mechanisms into linguistic constructs*, especially mechanisms for managing resources and projects. The paper explains these principles and how Racket lives up to them, presents the evaluation framework behind the design process, and concludes with a sketch of Racket's imperfections and opportunities for future improvements.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases design guidelines, language generation, full-spectrum language

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Racket, Historically Speaking

In 1995, we set out to create an outreach project for novice programmers. After observing students in labs over the course of a year, we understood that *nobody* could teach Scheme in an hour and then focus on the essence of computing and programming—contrary to an opinion widely held by instructors due to the numerous courses based on MIT's *Structure and Interpretation of Computer Programs* [1]. What was needed, instead, was a teaching language suitable for instructing beginners. Furthermore, emacs, vi, and similar editors were overwhelming students who had never programmed before, despite their special modes for editing and interacting with Scheme. We therefore also wanted a pedagogical programming environment for our beginners, not just a re-appropriated power tool for professionals.

After we decided to implement our own teaching language and its environment, we still wanted to demonstrate that Scheme was an excellent implementation vehicle for these projects. We thought Scheme's macro system would help us experiment with language designs. The language also appeared to be a perfect match for constructing a simple interactive development environment (IDE); after all, many Lisp courses taught how to create a read-eval-print loop, and a lot of emacs was written in Lisp.

* Over 20 years, this work was partially supported by our host institutions: Rice University, University of Utah, Brown University, University of Chicago, Northeastern University, Northwestern University, Brigham Young University, and Indiana University as well as several funding agencies and foundations: AFOSR, CORD, Darpa, the Department of Education's FIPSE program, the ExxonMobile Foundation, Microsoft, the Mozilla Foundation, NSF, the Texas Advanced Technology Program.

Using Scheme as a starting point turned out to be an acceptable choice, but we soon found we needed a lot more. We discovered that more was needed than a hygienic and pattern-oriented variant of Lisp's old macros. In the same vein, the development of a large system exposed the difference between having safe abstractions and mimicking them via `lambda`. Finally, implementing an IDE also called for executing arbitrary programs under the control of our own program—and this goal clarified that Scheme did not come with the means for managing resources and for ensuring the security of the hosting program.

Over time, we adapted Scheme to serve our needs. We built a syntax extension system on top of Scheme's macros, added mechanisms for the creation of safe abstractions, and turned features of the surrounding operating system into linguistic constructs so that we could program resource administrators and security containers. In time, our language became a full-fledged tool for the working software engineer. By 2010, our dialect of Scheme had evolved so much that we renamed it to Racket [19] to let the world know that we had something different.

2 The Principles of Racket

While we have reported on the pedagogic aspect of the project elsewhere [7], this paper presents the design principles behind Racket and illustrates with concrete examples how they affect the reality of its implementation. It groups these principles under three slogans:

1. *Racket is about creating new programming languages quickly.*
   Programming is a form of problem solving. A proper approach uses the language of the domain to state the problem and to articulate solution processes. In support of this mode of programming, Racket helps programmers create and quickly deploy new languages. In particular, the mechanisms for creating and deploying languages must be contained within the language itself. Once Racket is installed, there must be no need to step outside to use one of its new languages. This principle is in stark contrast to the numerous external tools and command-line pre-processors that are used to create (embedded) domain-specific languages.
2. *Racket provides building blocks for strong protection mechanisms.*
   If programming is about solving problems in the correct language, systems will necessarily consist of interconnected components in several different languages. Due to the connections, values flow from one linguistic context into another. Since languages are charged with providing and preserving invariants, the creators of languages must have the power to protect the languages' invariants. By implication, Racket must come with mechanisms that enable programmers to protect individual components from their clients.
   For this reason, Racket comes with the proper building blocks to set up or construct protection mechanisms at any level, all the way from C to languages with sound, higher-order type systems, and any mixture in between.
3. *Racket turns extra-linguistic mechanisms into linguistic constructs.*
   When programmers must resort to extra-linguistic mechanisms to solve a problem, the chosen language has failed them. Even if it is not always obvious how to fix such failures, programming language researchers ought to accept the general idea and try to work on finding the proper linguistic mechanisms. Due to Racket's uses, the language currently internalizes several resource-management mechanisms that are often found in the underlying operating system. Similarly, this philosophy prohibits the idea of "projects," as found in other IDEs, because this also externalizes resource management, linking, and other aspects of program creation.

Evaluating the use of such principles must take place in a feedback loop that encompasses more than the compiler for the language. In Racket's case, the feedback loop's evaluation stage contains a range of software systems, especially DrRacket [9], the Racket IDE.

Sections 3 through 5 explain the principles in depth: language-oriented programming, protection mechanisms for full-spectrum programming, and services-as-constructs. Section 6 introduces Racket's feedback loop in some detail and how it helps us use the guidelines to turn principles into reality. Finally section 7 puts the principles in perspective, pointing out in particular where they remain goals and the research needed to reach those goals.

Listing 1 A Racket module

```
#lang racket                                                    demo.rkt

(provide
  ;; type Video = [Listof Image]
  ;; Natural -> Video
  walk-simplex)

;; ---------------------------------------------------------------------
(require "small.sim" 2htdp/image)

;; Natural -> Video
(define (walk-simplex timing)
  ... (maximizer #:x 2) ...)
```

3 Racket is a Programming-Language Programming Language

Racket is a programming language. Actually, at first glance it looks like a family of conventional languages, including a small untyped, mostly-functional by-value language (`racket/base`), a batteries-included extension (`racket`), and a typed variant (`typed/racket`).

Like all programming languages, plain Racket forces the programmer to formulate solutions to problems in terms of its built-in programming constructs. But, Racket is also a member of the Lisp family, which has always insisted on stating solutions in the most appropriate language, one suited to the problem domain. As Hudak [21] puts it, "domain-specific languages are the ultimate abstractions."

Following this reasoning, each program component is articulated in the Racket-based programming language that is best suited for the problem it solves. If the language is not available, the Racket programmer creates it, possibly even for a single module. To support this kind of system building, Racket is a programming-language programming language.

Listings 1 and 2 illustrate the principle. The first module in listing 1 uses the `racket` language, which is specified in the so-called `#lang`—pronounced "hash lang"—line. The module provides a single function; the comments inside the `provide` specification informally state a type definition and a function signature in terms of this type definition. To implement this function, the module uses `(require "small.sim")` to import functionality from the module in listing 2 and then defines its own functions.

The box in the top right of a listing specifies the filename. It is not a part of the code. The astute reader will notice that this violates the principle of keeping everything in the language.

The creator of the module in listing 2 prefers a domain-specific language, because the module's purpose is to synthesize a function for a simplex, and the most natural way to specify the latter is to state a collection of linear inequalities. The comments below the `#lang` line in listing 2 state that the module exports a single function, `maximizer`. Concretely, the

Listing 2 A module for describing a simplex shape

```
#lang simplex                                                  small.sim

;; provides: synthesized function maximizer:
;;     #:x Real -> Real
;;     #:y Real -> Real

#:variables x y

3 * x + 5 * y <= 10
3 * x - 5 * y <= 20
```

`#:variables` specification and the following inequalities determine the `maximizer` function. When called as `(maximizer #:x n)`, the function produces the maximal `y` value; conversely, `(maximizer #:y m)` delivers the maximal `x` value.

In support of this kind of language-oriented programming, Racket provides a syntax extension system that borrows elements from Scheme's macro system [4, 23, 24] but also improves on it in several different directions. First, the Racket syntax extension system is about defining languages [12, 25, 26], not just extending an existing language with new linguistic constructs. For example, Racket's class system [16], its first-class components [14], and its language of (loop) comprehensions are just such sub-languages, though their constructs are indistinguishable from Racket's core features. Naturally, a Racket-based language is just a module whose exports make up a new language. These exports must include certain features and may otherwise come with any syntactic constructs and run-time values deemed necessary. The module may define these exports or may import and re-export them from an existing language. Hence, a language module can easily add features to, or subtract them from, an existing language.

Second, the syntax extension system also allows a language module to redefine the meaning of existing constructs. Take function application, for example. Like Lisp, a Racket function application is just a pair of parentheses around the function and its arguments:

```
(f a ...)
```

Racket's syntax system elaborates surface syntax to kernel syntax:

```
(#%app f a ...)
```

The keyword `#%app` is Racket's internal sign post for the function application syntax—and a language can re-define its meaning. Here is a simplistic re-definition:

```
#lang racket
(provide (rename-out [call #%app]) ...)

(define-syntax-rule
  (call f a ...)
  ;; rewrites to
  (if (check-in-defines f) (#%app f a ...) (signal-error f a ...)))
```

This module defines the syntactic abbreviation `call`. A use of `call` expands to an `if` expression that checks a property of `f` and, if it holds, uses the *imported* application syntax (underlined) to create a function application; otherwise it signals an error. On export, `call` is renamed to `#%app`, meaning when another module specifies this module as its language, the compiler uses the `call` syntax to elaborate the module's function applications, e.g.,

```
(g b ...)
-- compiles to--> (#%app g b ...)
== equivalent  == (call g b ...)
-- compiles to--> (if (check-in-defines g) (#%app g b ...) (signal-error g b ...))
```

That is, the final code uses plain `racket`'s `#%app` construct to evaluate the function application—and that is regular call-by-value function application.

This example is inspired by the teaching languages [6]. In particular, the first-order functional teaching language uses it to check whether the function position is a name defined by the program or the language so that it can produce novice-friendly error messages when something else shows up. However, the pattern is used much more widely. For instance, the FrTime language uses this same mechanism to create a dataflow variant of call-by-value [2].

Third, Racket's syntax extension system grants a language-defining module access to the entire syntax tree for a guest module, not just individual nodes in the syntax tree. This access allows the collaboration between the rewriting rule for `#%app` and `define` in the above example. Indeed, this kind of communication smoothly generalizes to complex context-sensitive analysis tasks and, in particular, allows for the implementation of a rather conventional type checker [42].

Fourth, Racket comes with a library that supports the programmatic creation of lexers and parsers [34]. It is thus possible for a language implementation to transform conventional syntax into regular S-expression syntax and to subject this result to the conventional syntax extension system and its rewriting rules. See listing 2 where the implementor of a domain-specific language prefers an ASCII-mathematics notation. Importantly, the separation of parsing from the syntax extension naturally creates an interface between unrelated parts of language design—notation and meaning—and thus enables language engineers to factor the work into two independent components: design of surface notation and meaning.

Finally, Racket insists on separating the various stages of language processing, particularly enforcing a strict separation of compile-time from run-time code. For example, the rewriting rules generate pure syntax and may not embed other language values inside this syntax. Similarly, since the world of Racket languages is actually an inverted pyramid of languages atop languages, each language-processing module may have side-effects—and these side-effects must be insulated from the rest of the language-processing pipeline.

In sum, Racket's toolbox empowers programmers to create new languages quickly and thus enables language-oriented program design. The key to this achievement is to improve over Lisp and Scheme's approaches: Racket carefully stages syntax elaboration [12], eliminating Lisp's problematic `eval-when-where` approach; it enables the quick derivation of new languages from existing ones; and it enables the introduction of conventional syntax.

4 Racket Covers a Full Programming Language Spectrum

An abstraction enforces invariants. Languages are abstractions, and their creators must have the means to build the necessary enforcement mechanisms—especially when components in these languages end up in an interconnected, multi-lingual contexts. Since Racket is a language for building programming languages, it supplies the building blocks for the construction of enforcement mechanisms, too. Indeed, Racket's building blocks allow the creation of a spectrum of languages, and Racket programmers may safely compose components written in various elements of this spectrum.

To get a sense of what these enforcement requirements may mean, consider the kinds of languages a Racket programmer may build. As the literature on domain-specific languages

Listing 3 A Racket module using the foreign-function interface

```
#lang racket                                                      ffi.rkt

(provide
 ;; [Vectorof [Vectorof Real]] -> [Vectorof Real]
 simplex)

;; ---------------------------------------------------------------------
(require ffi/unsafe)

(define lib-simplex (ffi-lib "./coin-Clp/lib/libClp"))

(define (simplex M)
  ... (-simplex-set ...) ...)

(define -simplex-set
  (get-ffi-obj "simplex" lib-simplex (_fun _bytes -> _void)))
```

suggests [20], these constructions are often thin veneers over efficient C-level implementations. To support this kind of language, Racket comes with a foreign interface that allows parenthesized C-level programming. Programmers can refer to a C library, import functions and data structures, and wrap these imports in Racket values. Listing 3 shows an example of a module that imports functions from the `coin-Clp` simplex library and defines regular `racket` functions around them.

At the other end of the spectrum, a Racket programmer might wish to annotate an existing module with explicit types and expect type soundness. Doing just that is possible with `typed/racket`. Listing 4 illustrates how to transform the module from listing 1 into a typed one. Adding types moves knowledge out of comments into a statically checked sub-language, which proves the comments' validity and thus "hardens" [43] the component, because the invariants of the typed language are properly protected as its values flow into untyped components of the world.

Listing 4 A Typed Racket module

```
#lang typed/racket                                         demo-typed.rkt

(provide walk-simplex)

(: walk-simplex (-> Natural Video))

;; ---------------------------------------------------------------------
(require/typed 2htdp/image [#:opaque Image image?])
...
(define-type Video [Listof Image])

(define (walk-simplex timing)
  ... (maximizer #:x 2) ...)
```

While Racket does not automatically protect such flows of values, it comes with the tools to build invariant-enforcement mechanisms. Technically, it provides Miller's proxy mechanism [32] tailored to the needs of a Racket language builder. Racket's proxies come in two tiers: chaperones and impersonators [38]. Each monitors access to an underlying value to guarantee basic invariants. Programmers can create customized proxies that monitor access to functions, immutable values, mutable structures and objects—without ever getting in the way of other operations on the wrapped values and objects.

Listing 5 A Racket module with contracts

```
#lang racket                                             demo-contract.rkt

(provide
  (contract-out
    [walk-simplex (-> natural-number/c (listof image?))]))

;; -----------------------------------------------------------------
(require "small.sim" 2htdp/image)

(define (walk-simplex timing)
  ... (maximizer #:x 2) ...)
```

Racket offers a comprehensive contract system implemented with the proxy mechanisms. The contracts allow components to express Eiffel-style first-order assertions [31]. The introduction of contract boundaries smoothly generalizes these first-order statements to Racket's higher-order setting. That is, with contracts a component can advertise promises and obligations on values such as closures [10], objects, classes [36], and modules [37].

Listing 6 A contract for a first-class class in Racket

```
#lang racket                                            class-contract.rkt

(define MBTA/c
  (class/c
    [find-path
      ;; (find-path f t) finds paths from f to t
      (-> station/c station/c [listof path/c])]
    ...))
  (define (station? s) ...)
  (define path/c ...)

(provide
  ;; does the given value represent a T station?
  station?

  (contract-out
    [mbta%
     ;; represent the state of the MBTA with search functionality
     MBTA/c]

    [read-mbta-graph
     ;; an MBTA/c factory
     (-> (object/c mbta%))]))
```

Listing 5 shows how to express the comments from listing 1 into a contract. The conventional prefix syntax of the contract says that `walk-simplex` is a function, that this function accepts only natural numbers (`0`, `1`, `2`, and so on), and that it returns a list of images (checked with the `image?` predicate from the library). Racket checks this first-order contract in the expected way: if a client module applies the function to something other than a natural number, the client is blamed for a violation; if `walk-simplex` ever returns something other than a list of images, `contract.rkt` is blamed; and if there is no use, no error message is ever signaled even if `walk-simplex` were defined to return a string.

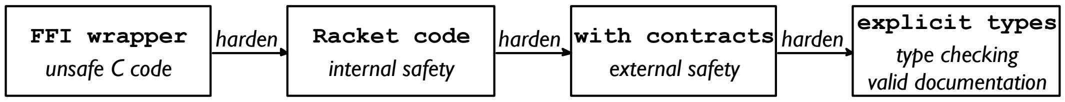
The extract of a module in listing 6 illustrates the use of higher-order contracts. Its header introduces two contracts and a flat predicate and exports the latter plus a contracted version of a class and its factory.

Currently, `typed/racket` [41] is the most important "client" of proxies and contracts. From a mechanical perspective, `typed/racket` is like `simplex`. From the teleological one, the two languages radically differ from each other; `typed/racket` is a sibling of `racket`, not just an arbitrary language implemented in the Racket world. As mentioned, its purpose is to help programmers harden untyped modules by equipping them with types.

While type checking guarantees consistency within the module and with respect to other typed modules, translating types of exported values into run-time contracts ensures a general form of type soundness, known as Tobin-Hochstadt's Blame Theorem.[1] For example, when a typed module exports a function on integers to an untyped module, the latter must not apply the function to a string; similarly, if a function on integer-valued functions flows from the typed to the untyped world, the latter must promise not to apply it to complex functions.

Listings 4 and 5 demonstrate this types-to-contracts translations in a concrete manner. The type of `walk-simplex` in the former translates to the contract shown in the latter. While this translation is straightforward for the functional core, extending this work to Racket's class-oriented fragment is the fruit of a multi-year research project [39, 40]. This extension implements both novel contract mechanisms for Racket's first-class classes as well as critical performance enhancements in the types-to-contracts translator.

Listing 7 The Racket language spectrum



With contracts and types, Racket includes a full spectrum of programming languages and, importantly, allows programmers to incrementally harden their "scripts" into programs. Initially, a Racket programmer may write a "script" in the traditional sense, that is, a thin layer around a C library. Assuming the choice of C is not performance-critical, the programmer could move the code to Racket, gaining operational and memory safety for the module in return. The third step would be the addition of contracts to the exports of this library to protect interactions between clients and the library. Finally, a programmer may equip the code with explicit, statically checked types, which creates validated documentation, improves

[1] As a library-based language, `typed/racket` is on the same footing as other libraries in the Racket ecosystem. Thus it cannot defend its invariants as thoroughly as typed languages such as Java or OCaml. Closing the remaining loopholes to enable more complete guarantees is ongoing research.

the performance[2] and maintainability of the code, and may reveal subtle mistakes. Listing 7 summarizes the hardening process in the Racket language spectrum diagrammatically.

While both contracts and types play a central role in this hardening process, the development of `typed/racket` is far more interesting from a linguistic perspective. Equipping `racket` with a type system is a challenging task. Programmers who use dynamically typed languages superimpose their own reasoning system as they design their code. It is fair to call this reasoning system a type system. Often this informal type system resembles naive set theory; at other times it incorporates elements from several different type systems.

The design challenge for `typed/racket` is to bring all of these informal type systems together in one framework—without introducing incompatibilities and contradictions. After all, when programmers harden a project, they do not want to modify their code to accommodate the type checker. Worse, any such modification might introduce a mistake or change the behavior of the program in undesirable ways. Because of the desire to allow incremental and selective hardening, it is also critical for `typed/racket` to preserve the semantics of `racket`.[3]

5 Racket Internalizes Extra-Linguistic Mechanisms

While many programming problems originate in a "real" world, program development is also a problem domain. As such, tools that support programming deserve a language of their own. Compiler writers take this idea seriously; for example, Dybvig and his group have developed a language for stating compilers as nano-scale transformations and used it for both educational [35] and commercial purposes [22]. When it comes to program development or program execution, however, IDEs resort to mechanisms from the surrounding operating system. They force programmers to develop programs in project contexts, delegate program execution to operating systems, and use *external* tools to inspect programs and their execution states. Racket's focus on languages as the key to problem solving points to the alternative solution of turning these extra-linguistic mechanisms into linguistic constructs [17].

To appreciate this domain, consider the original problem of building a pedagogic IDE for novice programmers. Clearly, the emphasis on pedagogy and novices prohibits the use of "projects;" students should be able to type in programs without any knowledge about computers and to run these programs without leaving the IDE they use. By implication, the IDE runs student programs under its control. Students make mistakes, though, and one common mistake is to launch a diverging program, that is, a program that consumes unbounded amounts of time, memory, or other resources (e.g., file ports, database handles, network connections; sometimes the access may be via instructor-provided libraries). Similarly, novices want to find mistakes in programs, meaning their instructors want to show them how to step through a program's execution. Finally, when a student submits a program to some homework server, this program must run in a security context that prohibits it from inspecting other students' solutions, attacking the server, and so on.

A close look at these requirements immediately suggests several areas of concern. Due to its design feedback loop, Racket includes the following external mechanisms as constructs at the moment: *inspectors*, which establish a hierarchy of access rights; *threads* that can be shut down from the outside; *sandboxes*, which restrict access to services; *custodians*,

[2] Performance enhancements can be realized under certain conditions; in general, types-as-contracts may reduce performance and often require performance tuning.

[3] The design of `typed/racket`'s type system is a complicated, but separate problem. A full discussion of this design would not illuminate the explanation of Racket's design principles, which is why we reserve this topic for a future paper.

which manage file handles, sockets, and database connections; *eventspaces*, which deal with GUI resources and events; and several more. The remainder of this section sketches two of these capabilities—inspectors and custodians—and how providing them inside the language provides fine-grained control over inspection and resources.

Listing 8 Inspection in Racket, part 1

```
#lang racket/base                                                  inspector.rkt

(define the-inspector (current-inspector))
(define sub-inspector (make-inspector the-inspector))

(define v
  (parameterize ([current-inspector sub-inspector])
    (dynamic-require "inspected.rkt" instance-of-s)))
```

Listing 9 Inspection in Racket, part 2

```
#lang racket/base                                                  inspected.rkt

(provide instance-of-s)
(struct s (fld))
(define instance-of-s (s 1))
```

Listings 8 and 9 demonstrate how Racket turns program inspection into a linguistic construct. Ordinarily, a Racket structure declaration like the one for `s` in listing 9 defines several functions: a constructor `s`, a field accessor `s-fld`, and a predicate `s?`. Unless a module exports the field accessor, instances of `s` are *opaque* to other modules in the system, i.e., other modules cannot view, access, or mutate the content of field `fld` in an instance. For example, dynamically loading module `inspected` from listing 9, retrieving `instance-of-s`, and printing it would reveal no information:

```
> (dynamic-require "inspected.rkt" 'instance-of-s)
#<s>
```

When the Racket IDE dynamically loads and evaluates a student program, however, it needs to have access to structure information for printing, stepping, and debugging.

To address these needs, Racket evaluates modules under a hierarchy of *inspectors*. If two modules run under the same inspector or incomparable inspectors in the hierarchy, they cannot view, access, or mutate each others structures unless they explicitly grant these rights via `provide`s of the respective functions. In contrast, if module `A` runs under the control of inspector i and another module `B` runs under the control of an inspector j that is below i, `A` can inspect `B`'s structures—whether `B` grants these rights or not.

Consider the module in listing 8, which concretely illustrates how inspectors work. The module creates a reference to the current inspector, that is, the inspector under whose supervision it executes. It then makes another inspector; the new one is below `make-inspector`'s argument, which is the module's current inspector. The module then uses `parameterize` to set the value of the `current-inspector` to this newly created inspector for the duration of the evaluation of

```
(dynamic-require "inspected.rkt" 'instance-of-s)
```

As a result, the value of `v` is a *transparent* instance of `s`, which is defined in `inspected` but exported without access methods. Hence, when `inspector` is loaded into the read-eval-print loop of DrRacket, `v` prints as `(s 1)`.

Listing 10 Programming operating-systems patterns in Racket

```
#lang racket                                                    universe.rkt
...
(define (launch-many-worlds* . th*)
  ;; allocate resources of th ... in the currently active custodian
  (define cc (current-custodian))
  ;; allocate resources of launch-many-worlds in new custodian c*
  (define c* (make-custodian))
  (define ch (make-channel))
  (parameterize ([current-custodian c*])
    ...
    (channel-put ch
      (list i (parameterize ([current-custodian cc]) (th)))))
  ;; th ... send values to channel ch;
  ;; if any of these is an exception structure, shut down
  ...
  (when (exn? x)
    (custodian-shutdown-all c*)
    (raise x))
  ...)
```

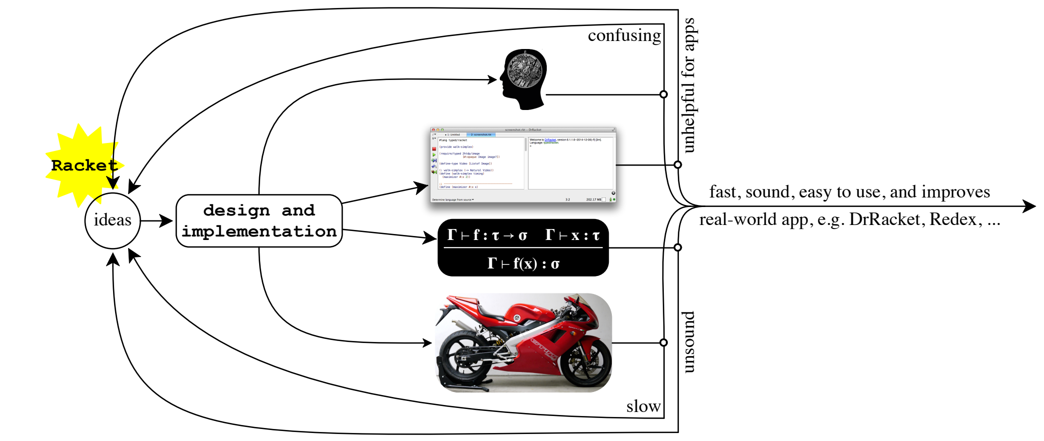
Listing 10 presents an example of resource administration, another operating-system service turned into a Racket construct. It displays the essence of the `launch-many-worlds` function, which is used to run students' distributed programs [8] in parallel. The function consumes an arbitrary number of thunks and runs them in parallel until all of them have produced a proper value or one of them has signaled an exception. Since the function itself consumes resources, it uses two *custodians*: its caller's—to manage the resources of the given thunks—and a new one—to manage its own resources, mostly threads. If any of these thunks raise an exception, the latter custodian is shut down and all of `launch-many-world`'s resources are released. For a more sophisticated pattern of killing threads safely, see Flatt and Findler's work on "kill safety" [15].

Finally, Racket also internalizes other aspects of its context. Dating back to the beginning, Racket programs can programmatically link modules [14] and classes [18]. In conventional languages, programmers must resort to extra-linguistic tools to abstract over such linguistic constructs; only ML-style languages and some scripting languages make modules and classes programmable, too.

6 The Racket Design Feedback Loop

The design of a language must take place in the context of a feedback loop. Like the feedback loop for many programming languages, Racket's feedback loop contains soundness theorems [40], performance evaluations [39], and usability studies [28]. As listing 11 shows, however, the Racket feedback loop also includes a number of software applications While the preceding sections already indicate the role that DrRacket played for developing, honing, and checking the principles, the creation of large and complex domain-specific languages such as Redex [5, 30], Scribble [13], Slideshow [11] and others have had an equally significant impact.

Listing 11 The Racket design feedback loop


Racket
ideas
design and implementation
confusing
unhelpful for apps
fast, sound, easy to use, and improves
real-world app, e.g. DrRacket, Redex, ...
Γ ⊢ f : τ → σ   Γ ⊢ x : τ
Γ ⊢ f(x) : σ
unsound
slow

Redex is probably the most sophisticated client of Racket's syntax extension system. It employs the latter at two levels: to compile the Redex language of grammar, type, and semantics definitions and as a target for the compilation. As such, Redex has stretched and expanded the syntax extension system.

Scribble is a domain-specific language for creating Racket documentation. Unlike the documentation system of conventional languages, a Scribble program can refer to, and compute with, bindings from a Racket library. As a result, programmers can easily create intensively cross-referencing manuals, language guides, and books in such a way that each occurrence of an identifier is *automatically* linked to its documentation. In fact, a Scribble file is just a Racket module, so Scribble documents come with all the benefits of other Racket code—including separate compilation, a feature absent in numerous document markup processors. Integrating Scribble with Racket exposed a gap in, and thus forced an expansion of, the phase separation model of Racket's syntax extension system [12].

Finally, Slideshow is both a domain-specific language for programming presentations and a graphical tool for displaying them. For a linguist with an awareness of the language of discourse, designing a language for the programmatic creation of presentations is a natural step. Presenters want a single point of control: they want parametrized re-use of slides, slide elements, and other concepts that are most easily expressed with a language but are difficult to obtain in a WYSIWYG tool. A programmer-as-presenter also has the natural desire to evaluate code within a presentation, possibly even the presentation itself [17]. Because of this combination, Slideshow's construction plays almost the same role in the feedback loop of design as the DrRacket IDE.

In general, all of these applications challenge the linguist in the problem-solving programmer. Each poses several different kinds of problems, best articulated and solved with problem-specific languages. In all cases, the purpose of the languages is to provide a protected and enforced abstraction. Equally important, they all need fine-grained control over resource-management mechanisms that are usually found outside of the language. Their existence and their designs both confirm the Racket principles and illustrate them, so Racketeers frequently consult these applications when they contribute new languages or new concepts to existing languages in the realm of Racket.

7 Racket, the Future: From Imperfections to Research Opportunities

Racket's design principles have produced a programming language that

- enables the rapid creation of new languages for specific problem areas and thus enables language-oriented programming;
- supports a full spectrum of general-purpose programming languages with various conventional degrees of safety; and
- internalizes mechanisms from its system context into linguistic constructs for fine-grained, programmable control.

Turning principles into reality almost always yields an incomplete, and possibly even flawed, product.[4] Racket is no exception, but we consider these imperfections as opportunities for future research. The remainder of this paper sketches some of them.

Racket's key advantage is its syntax extension system. It makes experienced programmers extremely productive, but it comes with an extraordinarily steep learning curve. Its syntax elaboration algorithm is hard to understand; its toolbox is large and complex; and it has some brittle, unexplored corners that occasionally trip up even experienced programmers. The situation calls for simplifications of the syntax system and for the creation of a smooth ramp for the toolbox (in terms of both tools and documentation).

In addition, the syntax extension system does not allow for a separation of concerns, and programs suffer from this. For example, many programming languages allow programmers to separate specifications from implementations. In conventional Racket, contracts play the role of specifications, functions implement them, and programmers may choose to separate the two concerns in a module. No such separation exists for the syntax system. While Culpepper's dissertation [3] research has made some progress in this direction, a lot more work on separating syntax specifications from syntax implementations is needed. Realizing both will greatly improve Racket's support for the principle of language-oriented programming.

Besides a language, modern programmers need an ecosystem. Indeed, many programmers equate languages with their ecosystems. For Racket, this equation means that the creation of a new language ought to include the derivation of an IDE from DrRacket. To some extent, DrRacket can already support new languages automatically, e.g., with on-line syntax checking and simple refactoring actions. For other tools, such as a syntax-directed stepper, this process would need a significant amount of work and comes without guidance or automation.[5]

The currently available enforcement mechanisms give rise to a full spectrum of conventional programming languages: Typed Racket, Racket with contracts, Racket, and `ffi/unsafe` Racket. Although this spectrum is expressive, it lacks power at both ends. To achieve full control over its context, Racket probably needs access to assembly languages on all possible platforms (from hardware to the web's JavaScript). How to integrate this power in a portable manner is unclear. To realize the full power of types, Racket will have to be equipped with dependent types. Tobin-Hochstadt and his Typed Racket group are currently working on first steps in this direction, focusing on numeric constraints in `typed/racket`. When a Racket program uses vectors, its corresponding typed variant type-checks what goes into these vectors and what comes out, but like ML or Haskell, indexing is left to a (contractual) check in the run-time system. Integrating Xi and Pfenning's form of programming with numeric constraints [44] into `typed/racket` is a natural step beyond plain types.

[4] The same caveat applies to the design *process*, not only its result, but covering the positives and negatives of the design process is beyond the scope of this paper.

[5] Spoofax [27] comes with domain-specific languages for the generation of IDE tools, but also relies on extra-linguistic mechanisms.

More generally, Racket's spectrum of languages creates a multi-lingual world for programmers, though with far more structure than currently found in practice. Even though Matthews and Findler [29] have studied the basics of multi-lingual programs, their theory covers only a small part of this world. Our work on Racket clearly calls for extensions of this result in several directions, including the sound interaction between by-value and by-name (or lazy) variants of Racket, typed and dependently typed variants, and so on. We expect that studying how to protect verified code as it is co-mingled with other kinds of code will yield new insights into Racket's safety mechanisms.

Racket must also broaden its horizon and consider security concerns, both as an enforcement action but also as an application of the third principle. While sandboxes address some of the security concerns of running a student program in a homework submission server, properly addressing this problem calls for articulating security policies and enforcing them in a system. Moore, et al. [33] recently presented Shill, a secure scripting language implemented atop Racket. Their work exposed serious gaps between Racket's principle of language-oriented programming and its implementation as well as in Racket's approach to enforcing security. Once again, we consider these weaknesses an opportunity to improve Racket and expect to study these problems in the near future.

Acknowledgements We wish to thank Claire Alvis, Ryan Culpepper, John Clements, Stephen Chang, Richard Cobbe, Greg Cooper, Christos Dimoulas, Bruce Duba, Carl Eastlund, Burke Fetscher, Cormac Flanagan, Kathi Fisler, Tony Garnock-Jones, Paul Graunke, Dan Grossman, Kathy Gray, Casey Klein, Jacob Matthews, Scott Owens, Greg Pettyjohn, Jon Rafkind, Vincent St-Amour, Paul Steckler, Stevie Strickland, James Swaine, Asumu Takikawa, Kevin Tew, Neil Toronto, and Adam Wick for helping us hone the guidelines.

In addition to the reviewers, Ben Greenman, Vincent St-Amour, Asumu Takikawa made several suggestions on improving the presentation.

References

1 Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

2 Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308, 2006.

3 Ryan Culpepper. Fortifying macros. *J. Functional Programming*, 22(4–5):439–476, 2012.

4 R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

5 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* The MIT Press, 2009.

6 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs.* MIT Press, 2001.

7 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *J. Functional Programming*, 14(4):365–378, 2004.

8 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A Functional I/O System. In *International Conference on Functional Programming*, pages 47–58, 2009.

9 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *J. Functional Programming*, 12(2):159–182, 2002.

10 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, pages 48–59, 2002.

11 Robert Bruce Findler and Matthew Flatt. Slideshow: Functional presentations. *J. Functional Programming*, 16(4–5):583–619, 2006.

12 Matthew Flatt. Composable and compilable macros: You want it *when?* In *International Conference on Functional Programming*, pages 72–83, 2002.

13 Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the book on ad hoc documentation tools. In *International Conference on Functional Programming*, pages 109–120, 2009.

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

15 Matthew Flatt and Robert Bruce Findler. Kill-safe synchronization abstractions. In *Programming Language Design and Implementation*, pages 47–58, 2005.

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

17 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 *International Conference on Functional Programming*, pages 138–147, 1999.

18 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Principles of Programming Languages*, pages 171–183, 1998.

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

20 Martin Fowler. *Domain-specific Languages*. Addison-Wesley, 2010.

21 Paul Hudak. Domain specific languages. In Peter H. Salas, editor, *Handbook of Programming Languages*, volume 3, pages 39–60. MacMillan, Indianapolis, 1998.

22 Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *International Conference on Functional Programming*, pages 343–350, 2013.

23 Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *Lisp and Functional Programming*, pages 151–161, 1986.

24 Eugene E. Kohlbecker and Mitchell Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *Principles of Programming Languages*, pages 77–84, 1987.

25 Shriram Krishnamurthi. *Linguistic Reuse*. PhD thesis, Rice University, 2001.

26 Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From macros to reusable generative programming. In *International Symposium on Generative and Component-Based Software Engineering*, pages 105–120, September 1999.

27 Kats C.L. Lennart and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *Object-Oriented Programming Systems, Languages & Applications*, pages 444–463, 2010.

28 Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Technical Symposium on Computer Science Education*, pages 499–504, 2011.

29 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems*, 31(3):1–44, 2009.

30 Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Rewriting Techniques and Applications*, pages 2–16, 2004.

31 Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

32 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

33 Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. Shill: A secure shell scripting language. In *Operating Systems Design and Implementation*, pages 183–199, 2014.

34 Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin Mcmullan. Lexer and parser generators in Scheme. In *Scheme and Functional Programming*, pages 41–52, 2004.

35 Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass framework for compiler education. *J. Functional Programming*, 15(5):653–667, 2005.

36 T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa, and Matthias Felleisen. Contracts for first-class classes. *ACM Trans. Program. Lang. Syst.*, 35(3):11:1–11:58, 2013.

37 T. Stephen Strickland and Matthias Felleisen. Contracts for first-class modules. In *Dynamic Language Symposium*, pages 27–38, 2009.

38 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In *Object-Oriented Programming Systems, Languages & Applications*, pages 943–962, 2012.

39 Asumu Takikawa, Daniel Feltey, Sam Tobin-Hochstadt, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Towards practical gradual typing. In *European Conference on Object-Oriented Programming*, 2015. To appear.

40 Asumu Takikawa, Stevie Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Object-Oriented Programming Systems, Languages & Applications*, pages 793–810, 2012.

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

42 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Programming Language Design and Implementation*, pages 132–141, 2011.

43 Tobias Wrigstad, Patrick Eugster, John Field, Nate Nystrom, and Jan Vitek. Software hardening: A research agenda. In *Script to Program Evolution*, pages 58–70, 2009.

44 Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Programming Language Design and Implementation*, pages 249–257, 1998.

Image References

Head Logo image in listing 11 is remixed from Nuri Aydoğdu's under AY SA 3.0.
Motorcycle image in listing 11 is remixed from Klaus Nahr's under AT SA 2.0.