A Programmable Programming Language

MATTHIAS FELLEISEN, ROBERT BRUCE FINDLER, MATTHEW FLATT, SHRIRAM KRISHNAMURTHI, ELI BARZILAY, JAY MCCARTHY, SAM TOBIN-HOCHSTADT

In addition to libraries and packages, programmers develop and use embedded problem-specific languages as building blocks of software systems. These languages help developers state solutions for the various aspects of the problem in appropriate terms. Complete software systems compose these partial solutions into a coherent whole. Sadly, this form of work is conducted without real support from the underlying programming language.

What this emerging development method calls out for, then, are programming languages that make the creation and use of embedded languages as convenient as the creation of libraries. Implementing this kind of programming language has been the goal of our 20-year-old Racket project. This paper presents the state of the project and sketches how it will proceed.

CCS Concepts: •**Software and its engineering → Language types;** *Context specific languages;* Development frameworks and environments;

ACM Reference format:
Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, Sam Tobin-Hochstadt. 2017. A Programmable Programming Language. 0, 0, Article 0 (2017), 7 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 PROBLEMS VS PROGRAMMING LANGUAGES

In the ideal world, software developers ought to analyze each problem in the language of its domain and then articulate solutions in matching terms. They could thus easily communicate with domain experts and separate problem-specific ideas from the details of general-purpose languages and specific program design decisions.

In reality, however, programmers use a mainstream programming language that someone else picked for them. To compensate for this conflict, they resort to—and on occasion build their own—domain-specific languages embedded within the chosen language (eDSLs). For example, JavaScript programmers employ jQuery for interacting with the DOM and React for dealing with events and concurrency. As developers solve their problems in appropriate eDSLs, they compose these solutions into one system. In short, developers effectively write multi-lingual software in a common host language.[1]

Sadly, multi-lingual eDSL programming currently rests on an ad hoc basis and is rather cumbersome. To create and deploy a language, programmers must usually step outside the chosen language to set up configuration files, run compilation tools, and link in the resulting object-code files. Worse, the host languages fail to support the proper and sound integration of components in different eDSLs. Finally, most available IDEs do not even understand eDSLs or perceive the presence of code written in eDSLs.

The goal of the Racket project is to explore this emerging idea of

language-oriented programming (LOP)

at two different levels. At the *practical* level, our goal is to build a programming language that enables language-oriented software design. This language must facilitate the easy creation of eDSLs, the immediate development of components in these newly created languages, and the integration of components in distinct eDSLs.

At the *conceptual* level, the case for LOP is analogous to the ones for object-oriented programming or for concurrency-oriented programming [3]. The former arose from making the creation and manipulation of objects syntactically simple and dynamically cheap, the latter from Erlang's inexpensive process creation and message passing. Both of these innovations enabled new ways of developing software and triggered research projects. The question is how we will realize LOP and how this will affect the world of software.

The decision to develop a new language, Racket, is partly a historical artifact and partly due to our desire to free ourselves from any unnecessary constraints of industrial mainstream languages while we investigate LOP. The next section spells out how Racket got started, how we honed in on LOP, and what this idea implies.

2 THE PRINCIPLES OF RACKET

The Racket project dates back to January 1995, when we started it as a language for experimenting with pedagogic programming languages [15]. Working on these languages quickly taught us that a language itself is a problem-solving tool. We then soon found ourselves developing different languages for different parts of the project: one (meta-) language for expressing many pedagogic languages; another one for specializing the DrRacket IDE [15]; and a third for managing configurations. In the end, our software was a multi-lingual system—just as described in the introduction.

Racket's guiding principle reflects the insight we gained:

> *Empower programmers to create new programming languages easily and to add them with a friction-free process to a code base.*

With "language" we mean a new syntax, a static semantics, and a dynamic semantics, which usually maps the new syntax to elements of the host language and possibly external languages via an FFI.

For a concrete example, take a look at figure 1. It displays a diagram of the architecture of a recently developed pair of scripting languages for video editing [2].[2] Their purpose is to assist people who turn recordings of conference presentations into YouTube videos and channels. Most of their work is repetitive—adding preludes and postludes, concatenating playlists, and superimposing

[1] The numerous language-like libraries in scripting languages (JavaScript, Python, and Ruby), books such as Fowler and Parson's [20], and web sites such as Tomassetti's [tomassetti.me/resources-create-programming-languages/, last visited May 21, 2017] are evidence for the desire of programmers to use and develop eDSLs.

DOI: 10.1145/nnnnnnn.nnnnnnn

[2] The video language, including an overview of the implementation, is available as a use-case artifact at ccs.neu.edu/racket/pubs/#icfp17-acf.

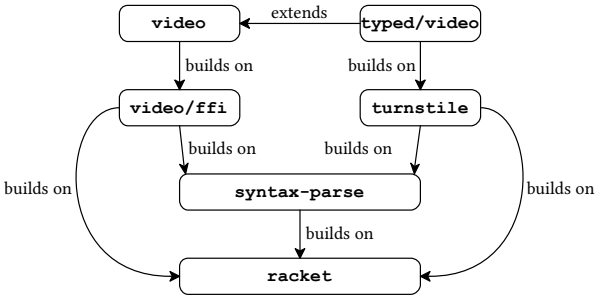


Fig. 1. LOP, a small example

audio—and only a small number of steps demand manual intervention. The task calls for a domain-specific scripting language, and `video` is a declarative eDSL that meets this need.

The `typed/video` language adds a type system to `video`. Clearly, the domain of type systems comes with its own language of expertise; `typed/video`'s implementation therefore uses `turnstile` [6], an eDSL created for expressing type systems. Similarly, the implementation of `video`'s rendering facility calls for bindings to a multimedia framework. Ours separates the binding definitions and the repetitive details of FFI calls into two parts: an eDSL for multimedia FFIs, dubbed `video/ffi`, and a single program in it. Finally, in support of creating all these eDSLs, Racket comes with the `syntax-parse` eDSL [7], which targets the domain of eDSL creation.

The LOP principle implies two subsidiary guidelines:

(1) *Enable creators of a language to enforce its invariants.* A programming language is an abstraction, and abstractions are about integrity. Java, for example, comes with *memory safety* and *type soundness*. When a program consists of pieces in different languages, values flow from one context into another and need protection from operations that might violate their integrity. See section 4.

(2) *Turn extra-linguistic mechanisms into linguistic constructs.* A LOP programmer who resorts to extra-linguistic mechanisms effectively acknowledges that the chosen language lacks expressive power [13].[3] The numerous *external* languages required to deal with Java projects—a configuration language, a project description language, a makefile language—represent symptoms of this problem. We treat such gaps as challenges. See section 5.

These principles have been developed in a feedback loop that includes DrRacket [15] plus typed [36], lazy [4], and pedagogical languages [15].

3 LIBRARIES AND LANGUAGES RECONCILED

Racket is an heir to Lisp and Scheme. Unlike its ancestors, Racket emphasizes functional over imperative programming without enforcing an ideology. Racket is agnostic when it comes to surface syntax, accommodating even conventional variants such as Algol 60.[4] Like many languages, Racket comes with "batteries included."

Most distinctively, Racket eliminates the hard boundary between library and language, overcoming a seemingly intractable conflict. In practice, this means new linguistic constructs are as seamlessly imported as functions and classes from libraries and packages. For example, Racket's class system and `for` loops are imports from plain libraries, yet most programmers use these constructs without ever noticing their nature as user-defined concepts.

Racket's key innovation is a *modular* syntax system [17, 26], an improvement of Scheme's macro system [11, 24, 25], which in turn improves on Lisp's tree-transformation system. A Racket module `provides` services such as functions, classes, and linguistic constructs. To implement these services, a module may `require` the services of other modules. In this world of modules, creating a new language simply means creating a module that provides the services for a language. Such a module may subtract linguistic constructs from a base language, re-interpret some others, and add a few new ones. A language is rarely built from scratch.

Like Unix shell scripts, which specify their dialect on the first line, every Racket module specifies its language on the first line, too. This language specification refers to a file that contains a language-defining module. Creating this file is all it takes to install a language. Practically speaking, a programmer may develop a language in one tab of the IDE, while another tab may be a module written in the language of the first. Without ever leaving the IDE to run compilers, linkers, or any other tools, the developer can modify the language implementation in the first tab and immediately experience this modification in the second. In short, language development is a friction-free process in Racket.

In the world of shell scripts, the first-line convention eventually opened the door to a slew of alternatives to shells: Perl, Python, Ruby, etc. The Racket world is experiencing a similar phenomenon, with language libraries proliferating within its eco-system: `racket/base`, the Racket core language; `racket`, the "batteries included" variant; and `typed/racket`, a typed variant. Some lesser-known examples are `datalog` and a `web-server` language [27, 30]. When precision is needed, we use the lower-case name of the language in `typewriter` font; otherwise we use just "Racket."

demo

```
#lang racket/base
(provide
  ;; type MaxPath = [Listof Edge]
  ;; Natural -> MaxPath
  walk-simplex)

(require "constraints" graph)

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

Fig. 2. A plain Racket module

[3] Like many Programming Language researchers, we subscribe to a weak form of the Sapir-Whorf hypothesis.

[4] See `docs.racket-lang.org/algol60/` [last visited May 24, 2017] as well as `hashcollision.org/brainfudge` [last visited Feb 6, 2017], which shows how Racket copes with obscure syntax.

Figure 2 is an illustrative module. Its first line—pronounced "hash lang racket base"—says that it is written in `racket/base`. The module provides a single function, `walk-simplex`. The accompanying line comments—introduced via semicolons—informally state a type definition and a function signature in terms of this type definition; figure 5 shows how developers can use `typed/racket` to replace such comments with statically checked types. To implement this function, the module imports functionality from the `"constraints"` module in figure 3. The last three lines of figure 2 sketch the definition of the `walk-simplex` function, which refers to the `maximizer` function imported from `"constraints"`.

The `"constraints"` module in figure 3 expresses the implementation of its only service in a domain-specific language because it deals with simplexes, which are naturally expressed with a system of inequalities. The module's `simplex` language inherits the line-comment syntax from `racket/base` but uses infix syntax otherwise. As the comments state, the module exports a single function, `maximizer`, which consumes two optional keyword parameters. When called as `(maximizer #:x n)`—as in figure 2—it produces the maximal `y` value of the system of constraints. As the lower half of figure 3 shows, these constraints are specified with conventional syntax.

```
#lang simplex                                              constraints

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

#:variables x y

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

Fig. 3. A module for describing a simplex shape

In support of this kind of programming, Racket's modular syntax system benefits from several key innovations. A particularly illustrative one is the ability to incrementally *re-define the meaning of existing language constructs* via the module system. This ability allows eDSL creators to ease their users into a new language by reusing familiar syntax, re-interpreted.

Take `lambda` expressions, for example. Suppose a developer wishes to equip a scripting language such as `video` with functions that check whether their arguments satisfy specified predicates. Figure 4 shows the basic idea:

line 01 The module uses the `racket` language.

line 03 It exports a defined compile-time function, `new-lambda`, under the name `lambda`, which is overlined to mark its origin as this module.

line 05 Here the module imports tools from a library for creating robust compile-time functions conveniently [7].

line 07 The comment says a function on syntax trees follows.

line 08 While `(define (f x) . . . )` introduces an ordinary function `f` of `x`, `(define-syntax (c stx) . . . )` creates the compile-time function `c` whose single argument is `stx`.

```
01 #lang racket                                                new-lam
02
03 (provide (rename-out [new-lambda lambda]))
04
05 (require (for-syntax syntax/parse))
06 ...
07 ;; Syntax -> Syntax
08 (define-syntax (new-lambda stx)
09  (syntax-parse stx
10   [(new-lambda (x:id (~literal ::) predicate:id) body:expr)
11     (syntax
12      (lambda (x)
13        (unless (predicate x)
14          (define name (object-name predicate))
15          (error 'lambda "~a expected, given: ~e" name x))
16        body))]))
17 ...
```

Fig. 4. lambda, re-defined

line 09 Like many functional languages, Racket comes with pattern-matching constructs. This one uses `syntax-parse` from the above-mentioned library. Its first piece specifies the to-be-matched tree (`stx`); the remainder specifies a series of pattern-responses clauses.

line 10 This pattern matches any syntax tree whose first token is `new-lambda` followed by a parameter specification and a body. The annotation `:id` demands that the pattern variables `x` and `predicate` match only identifiers in the respective positions. Similarly, `:expr` allows only expressions to match the `body` pattern variable.

line 11 A compile-time function synthesizes new trees with `syntax`.

line 12 The generated syntax tree is a `lambda` expression. Specifically, the function generates an expression that uses `lambda`. The underline marks its origin as the ambient language: here, `racket`.

others Wherever the syntax system encounters the pattern variables `x`, `predicate`, and `body`, it inserts the respective subtrees that match `x`, `predicate`, and `body`.

When another module uses `"new-lam"` as its language, the compiler elaborates the surface syntax into the core language like this:

```
(lambda (x :: integer?) (+ x 1))
–elaborates to⟶ (lambda (x :: integer?) (+ x 1))
–elaborates to⟶ (new-lambda (x :: integer?) (+ x 1))
–elaborates to⟶ (lambda (x)
                  (unless (integer? x)
                    <elided error reporting>)
                  (+ x 1))
```

The first elaboration step resolves `lambda` to its imported meaning [18], which is `lambda`. The second one reverses the "rename on export" instruction. Finally, the `new-lambda` compile-time function translates the given syntax tree into a `racket` function.

In essence, figure 4 implements a simplistic pre-condition system for one-argument functions. Next, the language developer might wish to introduce multi-argument `lambda` expressions, add a position for specifying the post-condition, or make the annotations

optional. Naturally, the compile-time functions could then be modified to check some or all of these annotations statically, eventually resulting in a language that resembles `typed/racket`.

4 SOUND COOPERATION BETWEEN LANGUAGES

A LOP-based software system consists of multiple, cooperating components, each written in domain-specific languages. Cooperation means that the components exchange values, while "multiple languages" implies that these values are created in distinct languages. In this setting things can easily go wrong, as figure 5 demonstrates with a toy scenario. On the left, a module written in `typed/racket` exports a numeric differentiation function. On the right, a module written in `racket` imports this function and applies it in three different ways, all illegal. If such illegal uses of the function were to go undiscovered, developers would not be able to rely on type information for designing functions or for debugging, nor could compilers rely on them for optimizations. In general, *cooperating* multi-lingual components must respect the invariants that each participating language establishes.

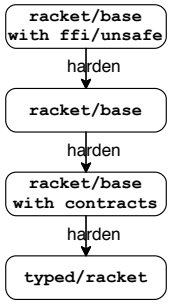
```
#lang typed/racket                    TR

(provide diff)

(: diff
   ((Real -> Real)
    ->
    (Real -> Real)))
(define (diff f)
  (lambda (x)
    (define lo (f (- x eps)))
    (define hi (f (+ x eps)))
    (/ (- hi lo)
       (* 2 eps))))
```

```
#lang racket                          RR

(require "TR.rkt")

;; scenario 1
(diff 1)
;; scenario 2
(define (f-bool x)
  #true)
(diff f-bool)
;; scenario 3
(define (f-char x)
  (string x x))
(diff f-str)
```

Fig. 5. Protecting invariants

In the real world, programming languages satisfy a spectrum of guarantees about invariants. For example, C++ is *unsound*. A running C++ program may apply any operation to any bit pattern and, as long as the hardware does not object, the program execution continues. The program may even terminate "normally," printing all kinds of output after the misinterpretation of the bits. By contrast, Java does not allow the misrepresentation of bits; but it is only somewhat more sound than C++ [1]. ML improves on Java again and is completely sound; no value is ever manipulated by any inappropriate operation.

Racket aims to mirror this spectrum of soundness at two levels: the level of language implementation itself and the level of cooperation between two components written in different embedded languages. Consider the soundness of languages first. As the literature on domain-specific languages suggests [20], these languages normally evolve in a particular manner, and this is true for the Racket world, too. A first implementation is often a thin veneer over an efficient C-level API. Racket developers create this kind of veneer with a foreign interface that allows parenthesized C-level programming [5]. Programmers can refer to a C library, import functions and data structures, and wrap these imports in Racket values. Figure 6 illustrates the idea with the sketch of a module; `video`'s initial implementation consisted of just such a set of bindings to a video-rendering framework. Of course, when a `racket/base` module imports the `ffi/unsafe` library, the language of the module is unsound.



A language developer that starts with the unsound eDSL is likely to make it sound as the second step. To this end, the language is equipped with run-time checks similar to those found in dynamically typed scripting languages to prevent the flow of bad values to unsound primitives. Unfortunately, this form of protection is ad hoc, and unless developers are hyper-sensitive, the error messages may originate from inside the library, blaming some `racket/base` primitive operation. To address this form of problem, Racket comes with higher-order contracts [16]. With those, language developers may uniformly protect the API of a library from bad values. For example, the `video/ffi` language provides language constructs for making the bindings to the video-rendering framework safe. In addition to plain logical assertions, Racket's developers are also experimenting with contracts for checking protocols, especially temporal ones [9]. The built-in blame mechanism of the contract library ensures sound blame assignment [10].

Finally, a language developer may wish to check some logical invariants *before* the programs run. Checking simple types is one particular example, but checking other forms of types is also possible. The `typed/video` language illustrates this point with a type system that checks the input and output types of functions and also numeric constraints on the integer arguments; as a result, no script can possibly render a video of negative length. Similarly, `typed/racket` is a typed variant of (most of) `racket`.

```
#lang racket/base                     ffi

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

(require ffi/unsafe)

(define (simplex M)
  ··· (ffi-simplex-set ···) ···)

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

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

Fig. 6. A Racket module using the foreign-function interface

Now consider the soundness of cooperating languages. As before, it is up to the language developer to anticipate how programs in this language interact with others. For example, the creator of `typed/video` provides no protection for its programs. By contrast, the creators of `typed/racket` intended the language to be used

in a multi-lingual context. Therefore `typed/racket` compiles the types of exported functions into the above-mentioned higher-order contracts. When, for example, an exported function must always be applied to integer values, the generated contract inserts a check that ensures the "integerness" of the argument at every application site for this function; there is no need to insert such a check for the function's return points, because the function is statically type checked. For a function that consumes an integer-valued function, the contract must ensure that the function argument always returns an integer. In general, a contract wraps exported values with a *proxy* [31] that controls access to the value. The idea is due to Matthews and Findler [29], while Tobin-Hochstadt and Felleisen's Blame Theorem [35] shows that if something goes wrong with such a mixed system, the run-time exception points to two faulty components and their boundary as the provable source of the problem [10]. In general, Racket supplies a range of protection mechanisms, and a language creator can use these mechanisms to implement a range of soundness guarantees for cooperating eDSLs.

5 UNIVERSALITY VS EXPRESSIVENESS

Just because a general purpose language can compute all partial-recursive functions, programmers cannot necessarily express all their ideas about programs in this language [13]. This point is best illustrated with an example. Imagine the problem of building an IDE for a new programming language in the very same language. Like any modern IDE, it is supposed to enable users to compile and run their code. If the code goes into an infinite loop, the user must be able to terminate it with a mouse click. To implement this capability in a natural[5] manner, the language must internalize the idea of a controllable process, a thread. If it does not, the implementor of the IDE must step outside the language and somehow re-use processes from the underlying operating system.

For a programming language researcher, "stepping outside the language" signals a failure. Or, as Dan Ingalls [21] phrases it, "[an] operating system is a collection of things that don't fit into a language[; there] shouldn't be one." Hence, we have worked to identify services that Racket borrows from the surrounding operating system and to assimilate such extra-linguistic mechanisms into the language itself [19]. Here are three sample constructs for which programmers used to step outside of Racket but no longer need to:

sandboxes: which restrict access to resources;
inspectors: which control reflective capabilities;
custodians: which manage resources (e.g., threads, sockets).

To understand how the inclusion of such services helps language designers, let us turn to a recent example, the `shill` language [32]. Roughly speaking, `shill` is a secure scripting language in Racket's eco-system. With `shill`, a developer articulates fine-grained security and resource policies—say, which files a function may access or which binaries the script may run—and the language ensures that these constraints are satisfied. To make this concrete, consider a homework server to which students can submit programs. The instructor might wish to run an auto-grade process for all the submissions. Using a `shill` script, the homework server can execute student programs that cannot successfully attack the server, poke around in the file system for solutions, access external connections to steal other students' solutions, and so on. Naturally, `shill`'s implementation makes extensive use of Racket's means for running code in sandboxes and harvesting resources via custodians.

[5] An alternative is to rewrite the entire program before handing it to the given compiler, which is exactly what distinguishes "expressiveness" from "universality."

6 THE STATE OF AFFAIRS

The preceding sections explain how Racket enables programmers

(1) to create languages via linguistic reuse for specific tasks and aspects of a problem;
(2) to equip a language with almost any conventional level of soundness (as found in ordinary language implementations); and
(3) to exploit a variety of internalized operating-system services for the construction of run-time libraries for these embedded languages.

What makes this form of language-oriented programming work is incrementality. If conventional syntax is not a concern, developers can create new languages from old ones, one construct at a time. Similarly, they do not have to deliver a sound and secure product all at once. They can create a new language as a wrapper around an existing C-level library, gradually tease out more of the language from the interface, and make the language as sound or secure as time permits or a growing user based demands.

Furthermore, the entire process takes place within the Racket eco-system. A developer creates a language as a Racket module and installs it by "importing" it into another module. This tight coupling has two implications. First, the development tools of the eco-system can be used for the creation of language modules and their clients. Second, the language becomes available for creating more languages. Large projects often employ a tower involving a few dozen languages—all of which helps manage the huge degree of complexity in modern software systems.

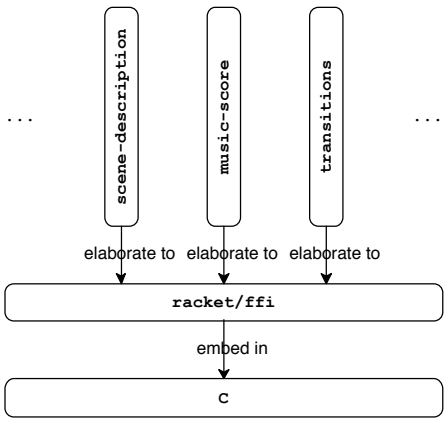


Fig. 7. A sketch of an industrial example

Sony's *Naughty Dog* game studio has created such a large project—actually, a framework for creating projects. Roughly speaking, their Racket-based architecture provides languages for describing scenes, transitions between scenes, the scores for scenes, and so on. Domain specialists use these languages to describe pieces of a game. The

Racket implementation composes these domain-specific programs and then compiles them into dynamically linked libraries for a C-based game engine. Figure 7 sketches the arrangement graphically.

Racket's approach to language-oriented programming is by no means perfect. To start with, recognizing when a library should become a language requires a tasteful judgment call. The next steps require good choices in terms of linguistic constructs, syntax, and run-time primitives.

As for concrete syntax, Racket currently has strong support for the typical, incremental Lisp-style syntax development. It comes with merely traditional support for conventional syntax, i.e., generating lexers and parsers. While traditional parsing introduces the above-mentioned natural separation between surface syntax and meaning, it also means that the development process is no longer incremental. The proper solution would be to inject Racket ideas into a context where conventional syntax is the default.[6]

As for static checking, Racket currently forces language designers to develop such checkers wholesale, not on an incremental basis. The type checker for `typed/racket`, for example, looks like the type checker for any conventionally typed language; it is a complete recursive-descent algorithm that traverses the module's representation and algebraically checks types. What Racket developers really want is a way to attach type checking rules to linguistic constructs, so that such algorithms can be synthesized as needed.

Chang et al.'s recent work is probably a first step toward a solution of this problem [6]. Thus far, they have demonstrated how their approach can equip a DSL with any structural type system in an incremental and modular manner. A fully general solution will also have to cope with sub-structural type systems, such as Rust's, and static program analyses, such as those found in most compilers.

As for dynamic checking, Racket suffers from two problems. On one hand, it provides the building blocks for making language cooperation sound, but developers must create the necessary soundness harnesses on an ad hoc basis. To facilitate the composition of components in different languages, we need both a theoretical framework and abstractions for the partial automation of this task.

On the other hand, the available spectrum of soundness mechanisms lacks power at both ends, and how to integrate these powers seamlessly is unclear. To achieve full control over its context, Racket probably needs access to assembly languages on all possible platforms—from hardware to browsers. To realize the full power of types, `typed/racket` will have to be equipped with dependent types. For example, 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. Tobin-Hochstadt and his Typed Racket group are currently working on first steps in this direction, focusing on numeric constraints [23], similar to Xi and Pfenning's work [37].

As for security, the Racket project calls for a significant breakthrough. While the `shill` team was able to construct the language inside the Racket eco-system, their work exposed serious gaps between Racket's principle of language-oriented programming and its approach to enforcing security policies. The team had to alter many of Racket's security mechanisms and invent new ones. Clearly, Racket must make this step much easier, meaning more research is needed to turn security into an integral part of language creation.

Finally, LOP poses brand new challenges for tool builders. An IDE typically provides tools for a single programming language or a family of related languages, among them debuggers, tracers, and profilers. Good tools communicate with developers on the terms of the source language. By its very nature, LOP calls for customization of such tools to many languages, their abstractions, and their invariants. We have partially succeeded in building a tool for debugging programs in the syntax language [8]; we have the foundations of a debugging framework[28]; and we have started to explore how to infer scoping rule and high-level semantics for newly introduced, language-level abstractions [33, 34]. Customizing such tools automatically to newly created (combinations of) languages remains a wide-open challenge.

In sum, programming language research has stopped short of the ultimate goal, namely, to provide software developers with tools to formulate solutions in the languages of problem domains. Racket is one attempt to continue the search for proper linguistic abstractions. While the project has achieved remarkable success in this direction, it also shows that programming language research has many problems to solve before the vision of language-oriented programming becomes reality.

Acknowledgments. We thank Claire Alvis, Robert Cartwright, Ryan Culpepper, John Clements, Stephen Chang, Richard Cobbe, Greg Cooper, Christos Dimoulas, Bruce Duba, Carl Eastlund, Burke Fetscher, Cormac Flanagan, Kathi Fisler, Dan Friedman, Tony Garnock Jones, Paul Graunke, Dan Grossman, Kathy Gray, Casey Klein, Eugene Kohlbecker, Guillaume Marceau, 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 their contributions.

A preliminary version of this paper appeared in the proceedings of SNAPL 2015 [14]. In addition to the SNAPL reviewers, Sam Caldwell, Eduardo Cavazos, John Clements, Byron Davies, Ben Greenman, Greg Hendershott, Manos Renieris, Marc Smith, Vincent St-Amour, and Asumu Takikawa made several suggestions to improve the presentation of this material. The anonymous CACM reviewers challenged several pieces of our original submission, which greatly improved the exposition.

Over 20 years, this work has been generously supported by our host institutions (Rice University, University of Utah, Brown University, University of Chicago, Northeastern University, Northwestern University, Brigham Young University, University of Massachusetts Lowell, Indiana University) as well as several funding agencies, foundations, and companies: AFOSR, CISCO, CORD, Darpa, the Department of Education's FIPSE program, the ExxonMobil Foundation, Microsoft, the Mozilla Foundation, NSF, and the Texas Advanced Technology Program.

Software. Racket is available at `racket-lang.org`.

[6] Language workbenches, such as Spoofax [22], deal with conventional syntax for DSLs but do not support the incremental modification of existing languages. A recent report [12] suggests, however, that these tool chains are also converging toward the idea of language creation as language modification. We conjecture that, given sufficient time, the development of Racket and language workbenches will arrive at similar designs and the latter will then support DSL integration, too.

REFERENCES

[1] Nada Amin and Ross Tate. Java and Scala's type systems are unsound: the existential crisis of null pointers. In *Object-Oriented Programming Systems, Languages & Applications*, pages 838–848, 2016.

[2] Leif Andersen, Stephen Chang, and Matthias Felleisen. Super 8 languages for making movies. In *International Conference on Functional Programming*, 2017, to appear.

[3] Joe Armstrong. Concurrency oriented programming. In *Frühjahrsfachgespräch der German Unix User Group*, `guug.de/veranstaltungen/ffg2003/papers/`, *last visited Feb 6, 2017*, 2003.

[4] Eli Barzilay and John Clements. Laziness without all the hard work. In *Functional and Declarative Programming in Education*, pages 9–13, 2005.

[5] Eli Barzilay and Dmitry Orlovsky. Foreign interface for PLT Scheme. In *Scheme and Functional Programming*, pages 63–74, 2004.

[6] Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *Principles of Programming Languages*, pages 694–705, 2017.

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

[8] Ryan Culpepper and Matthias Felleisen. Debugging macros. *Science of Computer Programming*, 75(7):496–515, July 2010.

[9] Christos Dimoulas, Max New, Robert Findler, and Matthias Felleisen. Oh Lord, please don't let contracts be misunderstood. In *International Conference on Functional Programming*, pages 117–131, 2016.

[10] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *European Symposium on Programming*, pages 214–233, 2012.

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

[12] Sebastian Erdweg, Tijs van der Storm, Markus Vlter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabril Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems and Structures*, 44, Part A:24 – 47, 2015.

[13] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[14] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *First Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–128, 2015.

[15] 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.

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

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

[18] Matthew Flatt. Bindings as sets of scopes. In *Principles of Programming Languages*, pages 705–717, 2016.

[19] 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.

[20] Martin Fowler and Rebecca Parsons. *Domain-specific Languages*. Addison-Wesley, 2010.

[21] Dan H. Ingalls. Design principles behind smalltalk. *Byte Magazine*, 6:286–298, 1981.

[22] Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench. In *Object-Oriented Programming Systems, Languages & Applications*, pages 444–463, 2010.

[23] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *Programming Language Design and Implementation*, pages 296–309, 2016.

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

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

[26] Shriram Krishnamurthi. *Linguistic Reuse*. PhD thesis, Rice University, 2001.

[27] 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 Computation*, 20(4):431–460, 2007.

[28] Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. The design and implementation of a dataflow language for scriptable debugging. pages 59–86, September 2007.

[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] Jay McCarthy. The two-state solution. In *Object-Oriented Programming Systems, Languages & Applications*, pages 567–582, 2010.

[31] 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.

[32] 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.

[33] Justin Pombrio and Shriram Krishnamurthi. Resugaring: lifting evaluation sequences through syntactic sugar. In *Programming Language Design and Implementation*, pages 361–371, 2014.

[34] Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. Inferring scope through syntactic sugar. In *International Conference on Functional Programming*, 2017, to appear.

[35] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Dynamic Language Symposium*, pages 964–974, 2006.

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

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