Oh Lord, Please Don't Let Contracts Be Misunderstood
(Functional Pearl)

Christos Dimoulas, Max S. New, Robert Bruce Findler, Matthias Felleisen
PLT, USA
{chrdimo,maxsnew,robby,matthias}@racket-lang.org

Abstract

Contracts feel misunderstood, especially those with a higher-order soul. While software engineers appreciate contracts as tools for articulating the interface between components, functional programmers desperately search for their types and meaning, completely forgetting about their pragmatics.

This gem presents a novel analysis of contract systems. Applied to the higher-order kind, this analysis reveals their large and clearly unappreciated software engineering potential. Three sample applications illustrate where this kind of exploration may lead.

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

Keywords Contracts, Specifications, Language design

1. Contracts, "always left out, never fit in"

Design by Contract [44] is a beautiful idea. Programmers articulate behavioral contracts and then the methods design themselves. Implicitly, Meyer claims that this idea scales from small pieces of code to large software systems.

Beugnard et al. [6] confirm this claim with their survey on the lasting impact of contracts in their study of component-based software, software adaptation, service oriented architectures, real-time and high-performance computing. Contracts have also become instrumental in the design of web-based systems to embedded software. In these settings, contracts express and enforce a wide range of component specifications; from basic null-pointer checks to coordination protocols and quality of service statements. Sadly, these achievements get only minimal support from mainstream programming languages; most provide little more than constructs for adding pre-conditions or post-conditions. To compensate for the lack of support from these conventional contract systems, software engineers employ a mixture of ad hoc solutions, invent contract design patterns, and rely on half-baked IDE support.

The introduction of contracts into higher-order functional languages [22] offers a solution to these linguistic problems. In a higher-order world, contracts naturally become first-class citizens, too. As a result, functional programmers may use their programming language to not only express logical assertions about their functions but also construct new forms of contracts with user-defined combinators. Especially in the context of languages that shout "domain specific languages" from every roof top—say Racket—researchers and developers ought to be able to eliminate the problems that plague sophisticated practical applications of conventional contract systems. If they put their mind to it,[1] they could construct linguistic mechanisms that raised the level of expressiveness and allowed programmers to articulate an unprecedented detail of precision for a relatively low cost in terms of code.

This gem demonstrates how this alternative mind set about contracts opens new possibilities. It starts with a novel conceptual analysis of contracts, re-imagining them as an interlocking system of interposition points, linguistic constructs for contract attachment, and DSLs for assembling logical assertions from rather simple building blocks (section 2). This abstract analysis is illustrated with three distinct examples of formulating contracts for libraries and components (section 3). These examples seemingly expand the convex hull of what behavioral software contracts can talk about, though they really just demonstrate untapped potential. Equipped with these examples, the gem finally (section 4) describes the relationship between contracts and type systems, on one hand, and run-time verification on the other. In short, the three of them form a triangle of complementary techniques for helping programmers with the difficult task of specifying interfaces.

Racket [27] is our language of choice to explain these ideas. It advertises itself as a tool for making DSLs, and it has been used extensively to study contracts. Specifically, Racket comes with a triple of superlatives: our father's parentheses, elegant weapons for a more civilized way of delivering gems; the world's most sophisticated higher-order contract system [27, §8], and the galaxy's best mechanism for formulating DSLs [27, §12]. None of the ideas are Racket-specific, though, and with a sufficient amount of extraordinary labor, Haskellians and Camelists can articulate the same ideas in the context of their villages.

2. "You hold the answers deep within your mind"

Programming language research has struggled with the concept of "meaning" for nearly five decades. Back in 1968, James Morris [47, p. 9] re-phrased the philosopher C. W. Morris to define this key concept via a trichotomy as follows:

1. *Syntax* delineates the legal forms, or utterances, of a language.

, .
Copyright © ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/

[1] After the introduction of higher-order contracts, many researchers—including three of the authors—spent valuable time on exploring their semantics and their shallow use in the context of *gradual typing* systems. The title refers to vast number of these efforts.

2. *Semantics* treats with the relation between the forms, or expressions, in the language and the objects they denote.

3. *Pragmatics* treats with the relation between a language and its users (human or mechanical).

We argue that from a strictly operational point of view, semantics are unnecessary. To specify a language, we need only give its syntax and pragmatics.

Strangely enough, Morris then jumps to the conclusion that "[the] pragmatics are most easily specified by how a machine ... will react when presented with a legal form of the language," completely ignoring how a human relates to such legal forms.

Ever since, efforts to give meaning mostly fall into one of two categories: denotational or operational. In the context of contracts, the first draws inspiration from type theory and deals well with type-like contracts [7, 23, 31]. The second category, due to Findler and Felleisen [22], opts for an operational method that specifies how checking contracts involves decomposing a contract into its "atomic" constituents, which can be checked against flat values. Both approaches have significant limitations. The first fails to describe the behavior of contract systems with sufficient precision [13] or to scale to dependent higher-order contracts [20, 23]. The second covers the entire range of contracts but obscures the actual workings of the contract system to such an extent that programmers fail to perceive the full expressive power of contracts.

This section instead presents a conceptual analysis of contracts, pulling together various strands of research on Racket's contract system. From this perspective, a contract system consists of three parts: an interposition layer; a contract attachment mechanism; and a suite of contract combinators. The first controls which aspects of a program's execution a contract system may monitor. The second specifies the means for associating a contract with (the result of) an expression and its context. The value of the expression is dubbed *the carrier of a contract*, while the context defines the *scope* where the contract is active. Finally, the third part constitutes the basis of the DSL with which programmers describe interfaces as contracts.

2.1 The Interposition Layer

An interposition layer intercepts run-time events of interest and relays them to a monitoring system. Information about the event is typically reified as a program value. For instance, Racket's interposition layer may detect calls to first-class functions and send a matching event to the contract system. The event identifies the function, its arguments, and its results.

One way to implement interposition is to modify the compiler of the underlying programming language. For example, Eiffel's compiler [45] injects interposition-related snippets into the program as it is translated. A compiler-based technique is highly inefficient in a higher-order world, however. When contracts themselves, functions, boxes or vectors are first-class values, interposition points become evident only at run time. Hence, using a compiler-based technique would require injecting interposition code for every possible function call, box access, vector mutation, and so on. In particular, it would not suffice to monitor just those pieces of code that are relevant to the contracts actually in the program.

In a higher-order world, the run-time system of a programming language can easily implement interposition via proxy values [57]. Such proxies wrap carriers of contracts and intercept events as needed. For example, a proxy for a mutable box intercepts each attempt to read or modify its content.

The most serious disadvantage of a dynamic interposition system concerns performance. When layers of proxies wrap carriers, the series of interceptions may impact performance in a significant manner. Indeed, even a single wrapping may prevent compilers from realizing certain optimizations. Furthermore, while proxy values can implement interposition for an expressive set of contracts, they are inadequate for others. Consider function contracts that describe temporal protocols. Realizing such a contract relies on detecting calls to captured functions during the extent of a call to a closure. Such contracts require an interposition layer that intercepts extensional events as well as intensional ones. Current approaches to obtain these intensional events fall back on aspect-oriented programming [51] or code re-writing techniques [35].

2.2 The Language of Contracts, the Single

Eiffel-inspired contract systems allow programmers to articulate contracts with the constructs of the underlying programming language. Indeed, in a simplistic contract system, contracts are merely Boolean expressions over function (method) parameters, local variables (fields), and function results. Some contract systems inject $\forall$ and $\exists$ quantifiers, though because their universes are finite, programmers consider them alternative names for definable functions.

At first glance, Racket's contract system extends this vocabulary with notations that mirror its nature as a higher-order functional language. For example,

```
(-> natural-number/c prime?)
```

may specify a contract for a function that maps natural numbers to prime numbers. Restricting the domain even further can be accomplished with the `and/c` combinator:

```
(-> (and/c natural-number/c (between/c 0 10))
    (listof prime?))
```

The range of this function contract says that a list of prime numbers is expected. A programmer can specify a dependency among several parameters and the result with the `->i` notation. Here is an illustrative example:

```
(->i ([s string?]
      [n (s)
         (and/c natural-number/c
                (between/c 0 (length s)))])
     [result (s n) (string-ref s n)])
```

It describes a function of two arguments: a string, named `s`, and an index into `s`, named `n`. The `result` is expected to be equal to the `n`th character of `s`. The `->i` notation demands that programmers explicitly state dependencies between parameter names and contracts, which explains the `(s)` and `(s n)` annotations.

In addition, Racket's contract system comes with notation for its class system, though programs using object-oriented constructs really just expand into the functional core language [26]. A contract from the object-oriented Racket fragment may contain

```
(is-a?/c text%)
```

which represents the property that the expected value is an instance of the built-in text-editor class, called `text%`. Thus,

```
(-> (list/c (is-a?/c text%) (is-a?/c frame%)) any)
```

specifies a function that expects one argument, namely, a two element list. These elements are, respectively, instances of `text%` and `frame%`; the latter is also a class from Racket's GUI framework.

Appearances are deceiving, however. In Racket, contracts are ordinary values, and programmers formulate ordinary expressions to create them. Due to Racket's liberal grammar for names, contract expressions may use a mix of intuitively library functions and notation, say `->` and `->i`. Naturally, these functions and abbreviations employ the interposition layer to implement the functionality.

Because the status of contracts is neither special nor restricted, it is thus perfectly fine to deal with contracts as if they were ordinary elements of Racket. For example, these two definitions

```
(define text/c (is-a?/c text%))
(define frame/c (is-a?/c frame%))
```

name two of the contracts mentioned above. Using these names and relying on ordinary evaluation means that

```
(-> text/c frame/c any)
```

creates a contract, namely the function contract from above. In the same spirit, a Racket programmer may define additional functions over contracts:

```
(define (maybe/c c)
  (or/c #false c))
```

This `maybe/c` contract combinator consumes any contract `c` and produces the contract `(or/c #false c)`, that is, a contract that checks whether a value is `#false` or satisfies the contract `c`. Here is a slightly more complicated contract combinator:

```
(define (table/c dom/c rng/c)
  (listof (list/c dom/c rng/c)))
```

It defines `(table/c d r)` as a contract for a list of lists of two elements, where the first element satisfies the contract `d` and the second one `r`. In particular, `(table/c symbol? string?)` specifies a table mapping Racket's symbols to Racket's strings.

2.3 Contract Attachment Mechanisms

To put contracts to work, programmers need a way to *attach* contracts to program components. An attachment accomplishes two purposes. First, it informs the contract monitoring system which contract to check for which values. Second, the attachment determines the *scope* of the contract, that is, the region of the program text where contracted values interact with the rest of the program.

Eiffel provides a single way for contract attachment; it turns instances of a class with contract annotations into carriers of the contracts. Mechanically, programmers merely annotate a method with Boolean expressions to state pre- and post-conditions. They may also attach such an expression to the field of a class and thus express a so-called class invariant. These annotations turn into dynamic contract checks whenever the instance interacts with some part of the program, including the class itself. In our terminology, the scope of an Eiffel contract is the *entire* program.

In contrast, Racket emphasizes that the attachment of a contract to a program component creates a contract boundary between the inside of the component and the rest of the program [22]. The word "boundary" is too coarse, however, and this section explains not only Racket's idea of contract attachment with two examples but also explains how to apply the notion of scope to attached contracts.

Racket's first construct for contract attachment links contracts with the exports of Racket modules. Here is an illustrative snippet from the implementation of DrRacket [21]:

```
(provide

 (contract-out
  [get-sorted-keybindings
   (-> text/c frame/c (table/c symbol? string?))]

 other-name))
```

The `provide` form of Racket specifies the bindings that a module exports. If such a (set of) bindings is wrapped in `contract-out`, the module attaches a contract to the exported binding. Expressed in our terminology, the contract's scope are all the clients of the module but *not the module itself*.

The particular example associates `get-sorted-keybindings` with a contract in its first `provide` clause. From the contract, a reader can tell that the named value represents a two-argument function. In the scope of this contract, a call to `get-sorted-keybindings` must thus come with two arguments: an instance of `text%` and a `frame%` object. Finally, the contract promises that `get-sorted-keybindings` returns a symbol-to-string table. Outside the scope of the contract for `get-sorted-keybindings`, a call does not have to live up to these constraints. In particular, the exporting module itself may call `get-sorted-keybindings` with different kinds of arguments, and it may not receive a `table` in return.

Contract attachment and contract scopes interact in subtle ways. For example, if the exporting module contains the definition

```
(define other-name
  get-sorted-keybindings)
```

then the `provide` specification implies that a client module can call `get-sorted-keybindings` in two different ways. By using `get-sorted-keybindings` it ensures the enforcement of a contract; by using `other-name`, it can circumvent the contract checks. Put differently, a client in the scope of some contract can refer to the same value in a contracted and in a direct manner, depending on what binding it uses. Thus contract attachment does not really attach a contract to a value but rather to a binding.

The second Racket construct for attaching contracts allows programmers to draw a contract boundary between a definition *within* a module and the rest of the module [56]. Here is such a localized variant of the preceding, module-level example:

```
(define/contract (get-sorted-keybindings txt frm)
   (-> text/c frame/c (table/c symbol? string?))
   #:freevar all-frames (listof frame/c)
   (... txt ... all-frames ... frm ...))
```

The snippet defines `get-sorted-keybindings` along with its contract. At a first approximation, `define/contract` turns the binding introduced by a definition into a carrier of a contract. The scope of the contract is the scope of the function name, *excluding* the body of the definition. Since local definitions may refer to non-local variables, `define/contract` allows programmers to attach contracts to those. The snippet showcases this idea with the `#:freevar` clause, which says that `get-sorted-keybindings` refers to one non-local variable, `all-frames`, and that it carries a `(listof frame%)` contract within the function definition.

Both `contract-out` and `define/contract` harness Racket's static scope to control the scope of a contract. This selectivity, together with the explicit interposition layer and the large pool of basic contract combinators, ensures that Racket's contract system comes with far more expressive power [18] than Eiffel's, in particular, because all Eiffel contracts have global scope. The challenge is to exploit this power effectively, and the key to solving this challenge is to view contract combinators as the basis of a DSL.

2.4 The Language of Contracts, the Complete LP

As many people have observed, a DSL is typically a veneer for an API or, in the terminology of functional programming, a collection of combinators. From the perspective of the preceding analysis, contract combinators come with four key properties, which we need to keep in mind as we construct a DSL on top of them:

1. A contract combinator declares interposition points of interest. For example, the simplest Racket contract combinator is `flat-contract`; it constructs a contract from a Racket predicate.[2] This combinator declares a single event of interest and its corresponding interposition point: the attachment of its result-contract. In contrast, the function contract combinator `->` de-

[2] In fact, Racket automatically coerces every predicate `pred` that shows up in the place of a contract to `(flat-contract pred)`.

clares three events of interest: the contract attachment, the application of the contracted function, and the return event.

2. A contract combinator specifies which information the interposition layer should relay for events of interest. For instance, for `flat-contract` the relevant information is the carrier of the contract itself. For `->` the relevant information consists not only of the carrier but also of the arguments and result of each application of the carrier in the scope of the contract.

3. A contract combinator provides hooks for programmers to specify the desired properties of values, expressed as predicates or contracts. Thus, `flat-contract` expects a single predicate that holds for the contract carrier, while `->` expects one contract per function argument plus its result(s). In addition to the programmer-provided predicates and contracts, contract combinators often check other properties of contract carriers. For instance, `->` implicitly specifies the carrier's arity.

4. A contract combinator establishes the scope of its sub-contracts. For instance, `->` says that while the scope of the argument contracts is the body of the contracted function, the scope of the result contract is that part of the program where the result is bound to an identifier (if any). For `->i`, though, the scope of its argument contracts extends not only the body of the contracted function but also to the code in the entire contract itself.

Indeed, the `->i` contract presents a first interesting example of a programmer-defined contract notation that exploits all elements of our contract analysis. It declares interdependent interposition points; it specifies scopes that include parts of the contract itself; and it thus exceeds the power of `->`. The `->i` combinator merely scratches the surface, however. The work on contract combinators for security [35, 46] and temporal contracts [16, 51] realizes more of the potential of these combinators, and it points the way to a general solution. Linguistic architects need to combine higher-order contract systems with DSLs and DSL-building tools so that programmers can easily express interfaces with the existing contract system. The next section demonstrates how to develop useful contract abstractions in this context so that programmers can write performant and expressive component contracts.

Note on Blame A knowledgeable reader may wonder why our analysis does not mention *blame*. In fact, blame naturally falls out of our analysis. Each contract combinator separates the specified properties for a contract carrier into those that are the responsibility of the contract carrier and those that are the responsibility of its scope. Both have *owners* [13], and those become *parties* to the *contract* via attachment. For the sub-contracts of a contract combinator, these parties switch or retain their role, depending on the scope of each sub-contract of the combinator.

Consider the contract for `get-sorted-keybindings`. Its owner is the module that attaches it to this function, and its scope is the set of client modules. Since the scope of the sub-contracts for the arguments is the body of the carrier, the roles of the parties are reversed for these contracts—just as Findler and Felleisen [22] say. In contrast, the roles remain intact for the result contract as its scope matches the scope of the `->`-defined contract.

In general, the two contract parties point to the pieces of a program that "agreed" to a contract for a value via an attachment mechanism and not the piece of the program that happened to "detect" the failed property. Thus contract scope provides a pragmatic starting point for the debugging process when a contract is violated, as the role of each party matches a static scope (region) of the program. It is exactly this intuition that is behind the formalization of "correct blame" assignment [14]. **End**

Note on Expressiveness Our analysis also provides a method to evaluate the expressive power of contract systems. For example a contract system is a *complete monitor* if its contracts can use all the information from the interposition layer. Consider Racket's `->d` predecessor of `->i`, whose scope does not include code in the contract itself. Thus it cannot intercept calls to the function's functional arguments if they take place within the extent of the contract. If `->d` were the only dependent contract combinator, the full contract system would not be a complete monitor [15]. **End**

3. "Don't be afraid to try again"

When a contract system comes with a DSL-building framework, programmers can create case-specific contract language with the obvious benefits. Such a language provides both a case-specific notation and pragmatics. This section demonstrates this idea with three examples, starting from a simple functional library to replacements for missing features and ending in a protocol language.

3.1 From Contracts to Spot Checkers

PLT Redex [19] is Racket's domain-specific language for specifying reduction semantics. It comes with a suite of tools, including an automated property tester. Like all such tools, the property tester uses a random number generator and a library for constructing enumerations of terms. Critically, these enumerations must be bijections so that the tester chooses small terms over large ones and checks the property against a term at most once.

Racket's enumeration library, `data/enumerate`, is built to provide utilities that make it easy to build elaborate enumerations and to help programmers build bijective enumerations. For the first part, `data/enumerate` supplies higher-order enumeration combinators. For the second part, the library leverages Racket's contract system so that programmers can create monitors for the bijection property.[3]

The basis of `data/enumerate` are enumerations for Racket's basic built-in data-types. For example, `string/e` constructs an infinite enumeration of strings while `char/e` constructs a finite enumeration of characters.[4] A programmer can query an enumeration with `from-nat` to obtain a Racket value from a natural number:

```
> (from-nat string/e 106355035256866)
"Hi!"
```

The `to-nat` function performs the inverse operation of `from-nat` and produces a natural number from a Racket value that belongs in the range of an enumeration:

```
> (to-nat string/e "Hi!")
106355035256866
```

Since contracts are first-class values, `data/enumerate` equips each enumeration with a contract that describes the enumeration's range, dubbed its *enumeration contract*. The `enum-contract` function extracts the enumeration contract from an enumeration:

```
> (enum-contract string/e)
#<procedure:string?>
```

The contract system uses the enumeration contract to monitor whether `from-nat` returns and `to-nat` consumes values that belong in the range of the enumeration:

```
> (to-nat string/e (list 1 2 3))
to-nat: contract violation
  expected: string?
  given: '(1 2 3)
  blaming: top-level
   (assuming the contract is correct)
```

[3] Bijectivity is undecidable for programmer-defined functions.

[4] In Racket, an identifier with suffix `/e` stands for an enumeration.

Programmers construct use-specific enumerations from these basic ones and the library's combinators. Consider the construction of an enumeration of the integers. Using the `map/e` combinator, we start by constructing an enumeration of negative integers, that is, an enumeration that maps each natural number to its inverse. The `map/e` combinator consumes four arguments: an enumeration `e`, a function `f`, its inverse `f-inv` and a contract `c`. With `f`, `map/e` can translate every element of the range of `e` to an element of the range of `f`; with `f-inv`, `map/e` can construct the injection from its result back to the natural numbers. The contract `c` serves as the enumeration contract for the result of `map/e`.[5]

Clearly, `-`[6] is a suitable way to go back and forth between the elements of `natural/e` enumeration and the negative integers. A reasonable contract argument is `(and/c integer? negative?)`, which accepts all negative integers. Surprisingly, a call to `map/e` with these arguments signals a contract violation with a 1% chance:

```
> (map/e - - natural/e
    #:contract (and/c integer? negative?))
map/e: contract violation
  expected: negative?
  given: 0
  blaming: top-level
   (assuming the contract is correct)
```

While the range of the enumeration includes `0`, the enumeration contract excludes it.

To help programmers find such problems, `data/enumerate` uses Racket's contract system to add spot-checkers to its exports. Specifically, the library checks whether `(map/e e f f-inv c)` is a bijection. To this end, the contracts of the library include a way to pick some random naturals below `10000` and check for each such number `x` whether

```
(= (to-nat e (f-inv (f (from-nat e x)))) x)
```

holds. The contract of `map/e` stipulates that

> `f` has to conform to `(-> (enum-contract e) c)`, and
> `f-inv` to `(-> c (enum-contract e))`.

Thus the contract of `map/e` also spot-checks whether `f` and `f-inv` are suitable for translating the range of `e` to the subset of values that satisfy `c`. The latter is exactly the piece of the contract of `map/e` that fails in the example. The contract discovers that applying `f` to `0`, an element of `natural/e`, returns `0`, which is not a negative integer. We thus relax the enumeration contract to `map/e` as follows:

```
(define non-positive-integer/e
  (map/e - - natural/e
    #:contract (and/c integer? (not/c positive?))))
```

The desired enumerations of the integers is just the union of `non-positive-integer/e` and `natural/e`. The `or/e` combinator constructs these unions, alas, its use here also uncovers another contract violation:

```
> (or/e natural/e non-positive-integer/e)
or/e: contract violation;
 new enumeration would not be two-way
 arg 1: #<infinite-enum: 0 1 2 3 4 5 6 7 8 9 10...>
 arg 2: #<infinite-enum: 0 -1 -2 -3 -4 -5 -6 -7...>
  blaming: top-level
   (assuming the contract is correct)
```

This time the `or/e` contract is violated, and the discovery is due to a spot-checker in `data/enumerate` that determines whether the arguments of `or/e` are suitable for jointly creating a bijection. Specifically it spot-checks whether the ranges of the arguments overlap by picking some random naturals below `10000` and subjecting them to the contracts of the two enumerations in an appropriate manner. Here, `non-positive-integer/e` maps `0` to `0`, which also satisfies the enumeration contract of `natural/e`.

Fortunately, `data/enumerate` comes with yet another enumeration combinator that deals with just this issue. The `except/e` combinator consumes an enumeration `e` together with a value `v` and then removes `v` from the range of `e`:

```
> (or/e
   natural/e
   (except/e non-positive-integer/e 0))
#<infinite-enum: 0 -1 1 -2 2 -3 3 -4 4...>
```

The implementation of the spot-checkers is straightforward. Roughly, a spot checker expands into an additional pre-condition for the contracts of basic enumeration combinators. The contracts of higher-order enumeration combinators derive their enumeration contracts and spot checkers from those of their constituents.

In sum, the construction of `data/enumerate` demonstrates the expressive powers of supplying contracts and contract combinators as first-class values. Specifically the construction and the use of the library greatly benefit from the ability to attach contracts to values, extract them, combine them with others, and derive spot-checkers from them as pre-conditions for other contracts. With this combinator-based language of contracts, the enumeration combinators can propagate properties from their arguments to their results, including the contracts on these values. Using this propagated information, the library can derive spot checkers and thus assist its users. Even though these guarantees are partial, they have been extremely useful for the developers and maintainers of Redex who use `data/enumerate` extensively and in sophisticated ways.

3.2 From Contracts to Specialization Interfaces

DrRacket is Racket's IDE [21]. Its implementation exposes many extension points in a rich hierarchy of classes. Developers can hook into these points to customize and extend the IDE's functionality. As for many extensible applications, this increased extensibility comes with a steep price. Due to Racket's class system, the DrRacket framework publicizes method names for *two* distinct purposes: as part of the instance interface and as part of the subclassing interface. Many of the latter must not be called directly from instances because this may break critical invariants concerning the internal state of the IDE. Concisely put, Racket's class system lacks Lamping [38]'s specialization interfaces.

Note on Class Syntax While most of Racket's notation for object-oriented programming with classes is irrelevant here, the reader needs to know that

- by convention, the names of classes end in `%`;
- `(new a%)` creates an instance of class `a%`; and
- `(send t m a)` invokes method `m` on object `t` and argument `a`.

All other notation is explained where needed. **End**

Here we demonstrate how Racket's contract system can overcome this weakness in Racket's class system. To make the idea concrete, we use the simplest extension of DrRacket—the addition or modification of key bindings—as a concrete example. Like most editors, the IDE supports a programmatic interface for associating a sequence of keystrokes with a keybinding handler and registering this combination. A keybinding handler is an arbitrary function that consumes two objects: one that represents the keyboard event and another that is DrRacket's editor, an instance of `text%`.

[5] It is intractable to derive such a contract from `f` automatically.

[6] In Racket, the subtraction function `-` is a variable arity number function. When it consumes a single number, it returns the negation of the argument.

The `text%` class comes with public methods for accessing and modifying the contents of DrRacket's definition window. Two of these methods are `paste` and `do-paste`. Keybinding handlers can invoke the first one to paste the contents of DrRacket's clipboard into the definitions window. Internally, `paste` delegates to `do-paste`, which is only public so that subclasses of `text%` can override it. Put differently, keybinding handlers should not invoke it directly. The documentation of `do-paste` explicitly states

> *[do] not call this method directly; instead, call* `paste`.

As mentioned though, this restriction is not enforced.

One way to enforce the restriction is to introduce a contract that explicitly allows `paste` to enable calls to `do-paste` in its dynamic extent and to prohibit all other calls to the latter function. In other words, contracts are used to articulate simplistic specialization interfaces that hide one method behind another. The general macro-defined form has this shape:

```
(hidden-method/c [method:id hidden:id] ...)
```

The form creates a new class contract where each pair of identifiers specifies that the first one is freely accessible and that the second one may be called only within the dynamic extent of the former.

For our running example, the implementor of DrRacket would use this new form as follows:

```
(define text%/c
  (hidden-method/c [paste do-paste]))
```

The newly defined contract, `text%/c`, can now be used to impose the non-calling restriction on `text%` itself:

```
(define/contract text+c% text%/c text%)
```

With `text%/c` it becomes impossible to invoke `do-paste` directly, while it is still possible to override the method in a subclass of `text+c%`. Consider this specific sample subclass:

```
(define mytext%
  (class text+c%
    (super-new)
    (define/override (do-paste start time)
      (displayln "pasting...")
      (super do-paste start time))))
```

The overriden `do-paste` displays a message and then delegates to the `do-paste` of the superclass `text+c%`. A client can create an instance `t` of `mytext%` and may legally invoke the `paste` method, which in turns invokes the new `do-paste` and thus the original `do-paste` of `text%`:

```
> (define t (new mytext%))

> (send t paste)
pasting...
```

Invoking `do-paste` directly on `t`, however, signals a violation:

```
> (send t do-paste 0 0)
pasting...
do-paste: contract violation;
    `do-paste' should be called only
    in the dynamic extent of `paste'
  blaming: top-level
    (assuming the contract is correct)
```

As the error message says, the invocation of `do-paste` of the superclass `text%` occurred in an inappropriate context.

Let us finally turn to `hidden-method/c` and its implementation. The key is to recognize whether an invocation of the hidden method takes place in the dynamic extent of method proper. To this end, `hidden-method/c` makes use of Racket's so-called parameters.[7]

A parameter is created and initialized with `make-parameter`:

```
(define p (make-parameter 42))
```

In the scope of `p`, `(p)` retrieves the current value of the parameter and `parameterize` temporarily sets it. Here is a toy example:

```
> (p)
42
> (define f
    (parameterize ([p 24])
      (displayln (p))
      (lambda () (displayln (p)))))
24

> (f)
42
```

The first sample interaction shows how `(p)` retrieves `42`, the parameter's current value. The second defines a thunk but uses `parameterize` to temporarily set the value of `p` to `24` during the thunk's creation. In the process, the `displayln` expression prints the modified value of `p`. Finally, the call to `f` is no longer in the dynamic extent of the temporary modification of `p`, meaning its `displayln` expression prints the original value.

Given this context, implementing `hidden-method/c` is relatively straightforward. First, `hidden-method/c` creates a translucent class contract [55].[8] For each pair of identifiers, it introduces a pair of method contracts into the class contract. In addition, it creates a shared but hidden parameter with initial value `#f`. The parameter is set to `#t` for the dynamic extent of an invocation of the method proper. The contract for the hidden method checks the value of the parameter and expects it to be `#t`.

For the running example, `hidden-method/c` creates one such contract for `paste` and another one for `do-paste`. Let us call the hidden parameter `paste-para`. The contract for `paste` temporarily sets the value of `paste-para` to `#t` for the dynamic extent of the body of `paste`. The contract for `do-paste` checks the value of the parameter and signals a contract violation unless its value is `#t`.

In conclusion, this second example shows how first-class contracts allow the addition of a missing linguistic feature. While `hidden-contract/c` is a single construct and rather simple to boot, its implementation subtly combines contract combinators with stateful programming. Since the state is hidden and even thread-safe, the new contract successfully replaces a feature that was missing from Racket's class system. Best of all, its addition to the full-fledged language formally guarantees heretofore informal statements in the documentation.

3.3 From Contract to Protocols

Many card games specify a protocol to coordinate the (inter)actions of players. Consider the "Take 5" game and its protocol:

1. The game is played in rounds, each round consists of turns.
2. At the beginning of each round, the dealer hands each player ten cards and creates four stacks with one card each, face up.
3. At the beginning of each turn, every player picks a card.

[7] Parameters correspond to fluid thread-preserved cells [28, 32, 33].

[8] Translucency means that the class contract checks calls to exactly those methods mentioned in the contract and leaves all other method calls alone.

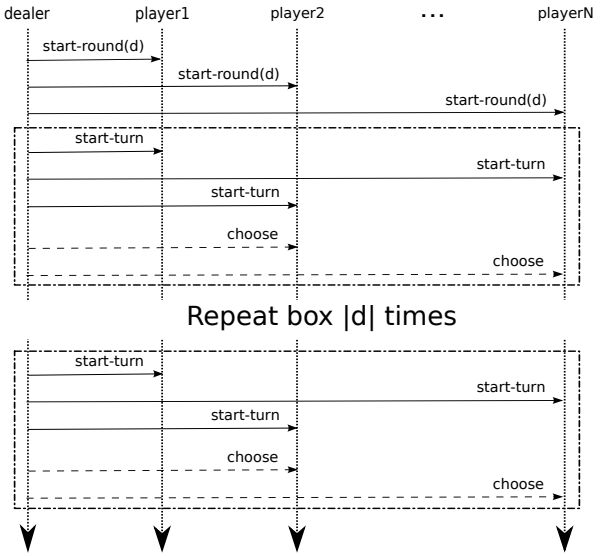

Figure 1. An interaction diagram for the "Take 5" protocol.

```
(define protocol-α
  (protocol
    (init-state start-round)
    (transitions
      ([---> (start-round cards) start-turn])
      ([---> (start-turn deck)
        (∨ start-round start-turn choose)])
      ([---> (choose deck)
        (∨ start-round start-turn)]))))
```

Figure 2. The `protocol-α` implementation.

4. Then the dealer places the chosen cards on the stacks according to some rules. This process may involve one of two kinds of interactions with a player:
   - The dealer may hand a player a complete stack.
   - The dealer may ask a player to pick up a stack.

   The dealer starts a new stack with that player's chosen card.

The number of turns per round per player is the same as the number of cards the dealer hands to each player at the beginning of the round. After a round completes, the dealer calculates a score for each player and, if a player's score exceeds a predetermined threshold, the game is over. Otherwise the players play another complete round of the game. Figure 1 summarizes the verbal description as an interaction diagram.

Now consider an implementation where different teams supply dealers and players. Clearly, such an arrangement calls for a strict enforcement of the protocol. Before a protocol can be enforced, however, someone must articulate it in such a way that everyone can see how it corresponds to the game instructions.

Here we show how programmers can use Racket's contract system to create a DSL for articulating and enforcing protocols. A natural implementation of the game represents dealers and players as objects, which implies that protocols are best implemented via some of form of stateful object contract [55]. It is equally *unnatural*, however, to formulate coordination protocols such as those in figure 1 as plain contracts, which is why we create a DSL.

Figure 1 also suggests that, unlike the `hidden-method/c` contract, a protocol requires a complex vocabulary. Even a simple game such as "Take 5" calls for a vocabulary of current state, state transitions, local and global knowledge, repetitions, and so on. To explain this protocol here, we introduce the protocol DSL in three steps, somewhat mirroring the development process.

The first step focuses on the relative ordering of method calls. All we need to know for this step is that the `player%` class exposes the three methods mentioned in the interaction diagram: `start-round`, `start-turn`, and `choose`. With this in mind, the first formulation of the protocol says that a sequence of method invocations on a `player%` object starts with first a call to `start-round` followed by a call to `start-turn`. After the `start-turn` call, a `player%` can optionally accept a call to `choose`, another call to `start-turn`, or a call to `start-round`, which kicks off a new round.

One natural choice is to express thus protocol as a nondeterministic finite state machine (FSM). The specification of such an FSM needs at least three elements: the states, the transitions among states, and the initial state(s). Since programmers prefer to avoid trivial redundancy, a simple FSM notation typically specifies states implicit, as part of the transition relation.

Modulo some details, this reasoning suggests the following kind of shape:

```
(protocol
  (init-state s0)
  (transitions [--> sf {st ⊕ (∨ st0 st1 ...)}]
               ...))
```

This protocol specification consists of two parts: a description of the possible transitions among the various states of the game and a declaration of what the starting state is. A transition clause has one of two shapes:

- `[--> sf st]` meaning when the machine is in state s_f, it may transition to s_t and no other state; or
- `[--> sf (∨ st0 st1 ...)]` meaning when the machine is in state s_f, it may transition to s_{t0} or to s_{t1} and so on.

The second kind of transition clause introduces the desired nondeterminism.

When Racket expands this specification into an object contract, it maps each state to the name of a method. If such a method mentioned is invoked, it checks whether it is a legitimate successor to the current state and, if so, records its name as the current state. For reasons explained below, the form actually translates into a 0-ary function that returns a new instance of an object contract.

Figure 2 illustrates the use of `protocol` with the specification of the expected method call sequence for the "Take 5" game. For instance, the protocol allows this transition:

```
[---> (start-round cards) start-turn]
```

which states that if the current state of the machine is `start-round`, the player object may accept a call to `start-turn`. Similarly,

```
[---> (start-turn deck)
      (∨ start-round start-turn choose)]
```

says that if the current state is `start-turn`, the player object may accept a call to `start-turn` or `choose`.

The syntax of figure 2 is the actual implemented syntax, which slightly differs from the proposed simple notation above.[9] Since each state corresponds to a method, the transitions also describe the signatures of these method. Here `(start-round cards)` states that method `start-round` consumes a single argument `cards`. Furthermore, each transition is wrapped in another pair of parentheses because transitions may also depend on conditions.

With `protocol-`α in place, a programmer could now instantiate `player%` and attach fresh object contracts to the objects:

```
(define/contract p1 (protocol-α) (new player%))

(define/contract p2 (protocol-α) (new player%))
```

Using these two objects, it is possible to simulate parts of the interaction between players and dealers. In particular, a programmer can determine whether a player recognizes out-of-order method calls. For example, if a dealer calls `choose` twice in a row after launching a round and a turn, the contract detects the protocol violation and signals it like this:

```
> (send p1 choose choice-deck)
choose: contract violation;
  expected the machine to be at state choose,
  however the current state is start-round or start-turn
    blaming: top-level
      (assuming the contract is correct)
```

The error message includes details about the current and the expected states, which point the programmer to an appropriate starting point for the debugging phase.

As the name indicates, the protocol specification of figure 2 is only an alpha version of the protocol from figure 1. The latter prescribes transitions that do not only depend on the possible current states of the protocol's machine but also on other parts of the game's state. For example, a `player%`'s `start-turn` method should be called only as many times as the number of cards handed over with the preceding call to `start-round`.

Hence the goal of the second step is to enable `protocol` to articulate and enforce such conditions. To this end, `protocol` allows the declaration of machine-local registers. Specifically, a `define-local-registers` clause defines and initializes variables whose scope is the transition table of the `protocol` specification. Furthermore, each transition may then check conditions involving these registers or conditionally update them. In particular, `#:if e` evaluates the expression `e` and allows a transition to proceed only if the result is `#t`; `#:update [c r e]` evaluates `c` and, if this produces `#t`, stores the value of expression `e` in register `r`.

Figure 3 shows how to formulate the beta version of the "Take 5" protocol. It declares one local register: `turns-left`. Two of the three transition clauses rely on this register:

- A `start-round` invocation is valid if the value of `turns-left` is `0`. A transition starting in `start-round` also sets the register to the length of `cards`.
- A `start-turn` invocation checks whether `turns-left` is positive. If so, it decreases `turns-left` by `1`, which expresses that a `start-turn` method call is valid only if a `player%` object has received fewer start-turn calls than the length of `cards` for the most recent `start-round` call.

Let's put `protocol-`β to work:

```
(define/contract p1 (protocol-β) (new player%))
```

[9] The source for our paper uses the Scribble documentation language [25], which executes the code snippets when it builds the PDF.

```
(define protocol-β
 (protocol
   (define-local-registers [turns-left 0])
   (init-state start-round)
   (transitions
    ([---> (start-round cards) start-turn]
     #:if (= turns-left 0)
     #:update [#t turns-left (length cards)])
    ([---> (start-turn deck)
     (∨ start-round start-turn choose)]
     #:if (> turns-left 0)
     #:update [#t turns-left (- turns-left 1)])
    ([---> (choose deck)
     (∨ start-round start-turn)]))))
```

Figure 3. The `protocol-`β implementation.

```
(define/contract p2 (protocol-β) (new player%))
```

Assuming that the dealer passes two cards to each player at the start of the round, a simulation of the dealer's interaction with the players fails when the dealer invokes `start-turn` a third time on `p1`. The contract signals the failure of the protocol in this way:

```
> (send p1 start-turn turn-deck)
start-turn: contract violation;
  condition (> turns-left 0) is not true
    blaming: top-level
      (assuming the contract is correct)
```

Note how the message explains that the pre-condition of a transition starting in `start-turn` does not hold.

Turning the beta version of the protocol into the final one, requires another extension of the protocol DSL. Many game (and other kinds of) protocols demand coordination among all of the participants. For instance, the "Take 5" protocol in figure 1 specifies that the player act according to some fixed ordering. To support this kind of coordination, this third step in the development of the protocol DSL introduces registers that are global to a collection of state machines. Technically, `protocol` allows the declaration of *global* registers. As the name implies, these registers are visible to all instances of the protocol contract. Once these instances are attached to `player%` objects, their transitions can inspect the global registers to validate method calls. They can also update these registers to reflect valid transitions.

Using global protocols, a programmer can now articulate the remaining four constraints of the "Take 5" protocol:

1. the dealer asks all players to play their turns;
2. each player can play exactly once per turn;
3. the dealer may invoke `choose` after all players have taken a turn;
4. the dealer may ask as many players to chose as prescribed by the rules of the game.

Figure 4 depicts the complete protocol for "Take 5," called `game-protocol`. It is relatively straightforward to transform the above constraints into Racket code, with the exception of constraint no. 2. The latter uses both a global and a local register:

- `player-id`, which holds a unique token per player per round;
- `played-this-turn`, which holds an initially empty list of tokens of those players that have played so far in a turn.

```
(define game-protocol
  (protocol
   (define-global-registers
     [players 2]
     [plays-so-far 0]
     [max-choices 1]
     [choices-so-far 0]
     [played-this-turn '()])
   (define-local-registers [turns-left 0]
     [player-id 'unknown])
   (init-state start-round)
   (transitions
     ([---> (start-round cards) start-turn]
      #:if (= turns-left 0)
      #:update
        [#t turns-left (length cards)]
        [#t player-id (gensym 'player)])
     ([---> (start-turn deck) (∨ start-round start-turn choose)]
      #:if (and (or (= plays-so-far players) (not (member player-id played-this-turn)))
                (> turns-left 0))
      #:update
        [(= plays-so-far players) played-this-turn '()]
        [(= plays-so-far players) plays-so-far 0]
        [(> choices-so-far 0) choices-so-far 0]
        [#t turns-left (- turns-left 1)]
        [#t plays-so-far (+ plays-so-far 1)]
        [#t played-this-turn (cons player-id played-this-turn)])
     ([---> (choose deck) (∨ start-round start-turn)]
      #:if (and (< choices-so-far max-choices) (= plays-so-far players))
      #:update [#t choices-so-far (+ choices-so-far 1)]))))
```

Figure 4. The final protocol contract for "Take 5".

A call to `start-turn` checks whether the player is the first to be called for a new turn or whether the token of this player is (not) included in the list of players that have already taken their turn this time around.

In addition to checking these conditions, a call to `start-turn` also updates several global registers. If the call concerns a new turn, it sets `played-this-turn` to the empty list. Finally, a valid `start-turn` call always causes the addition of the token of this player to `played-this-turn`, noting that the player has chosen a card for this turn.

To demonstrate the enforcement of `game-protocol`, we again create two `player%`s and attach a fresh instance of the protocol to each new player:

```
(define/contract p1 (game-protocol)
  (new player%))

(define/contract p2 (game-protocol)
  (new player%))
```

Here is the interaction that causes a protocol violation:

```
> (send p1 start-turn turn-deck)
start-turn: contract violation;
 condition (and (or (= plays-so-far players... is not true
  blaming: top-level
   (assuming the contract is correct)
```

As the (abbreviated) error message says, the method call is a request by the dealer to pick a second card for the current turn. Technically, the violation explains that the player's id is already a member of `played-this-turn`.

In perspective, the development of `protocol` shows how contracts can express rather complex coordination constraints for APIs and even families of APIs. The key is to develop linguistic abstractions that hide the complex uses of contracts that use local as well as global state variables. Since Racket's interposition layer is implemented via a run-time construct, the implementation of protocol does *not* need to rewrite third-party code to monitor all possible sequences of method calls. The contract attachment guarantees that even callbacks from such code are monitored. We conjecture that implementing `protocol` in Eiffel is impossible because contracts in that language are not first-class objects in their own right and cannot be attached to objects retro-actively.

Coordination restrictions have also been studied extensively in the context of type-state systems [54] and session type systems [36]. In comparison, contracts significantly boost the expressiveness of enforceable coordination protocols. Contracts can express and enforce protocols that depend on values such as the dependence of the "Take 5" protocol on the argument of the `start-round` method.

3.3.1 Implementing Protocol

The implementation of `protocol` leverages the standard function contract (`->i`) and object contract combinators via Racket's syntax system to synthesize the desired contract-generating function. In

```
#lang racket

(provide protocol ∨)  ; definition of ∨ omitted

(define-syntax (protocol stx)
  (syntax-parse stx
    #:literals (define-global-registers define-local-registers init-state transitions -->)
    [(_ (define-global-registers global:binding ...) ; ... denotes Kleene-star repetition of pattern
        (define-local-registers local:binding ...)
        (init-state init:id)
        (transitions ([---> (calling:id param:id ...)  next:id] side ...) ...))
     ; ---> (expands to)
     #'(let ()
         (define global.name global.val) ...
         (λ ()
            (define current '(init))
            (define local.name local.val) ...
            (object/c [calling (transition current calling next (param ...) side ...)] ...)))]))


(define-syntax (transition stx)
  (syntax-parse stx
    [(_ current calling next (param ...) #:if test:expr #:update u:test+binding ...)
     ; ---> (expands to)
     #'(->i ([this any/c] [param any/c] ...)
         #:pre/name "now-called" (param ...) (member? 'calling current)
         #:pre/name "condition"  (param ...) test
         #:pre/name "next state" (param ...) (set! current (maybe-id->list next))
         #:pre/name "updates"    (param ...) (and (when u.test (set! u.name u.val)) ...)
         any/c)]))
```

Figure 5. The implementation of `protocol`.

a nutshell, the goal is to translate a `protocol` specification of the shape

```
(protocol
  (define-global-registers
    [global-register-1 global-value-1]
    ...
    [global-register-1 global-value-1])
  (define-local-registers
    [local-register-1 local-value-1]
    ...
    [local-register-1 local-value-1])
  (init-state s0)
  (transitions t ...))
```

into an expression that creates a 0-ary function that manages the local and global registers:

```
(let ()
  (define global-register-1 global-value-1)
  ...
  (define global-register-n global-value-n)
  (λ ()
     (define current (list 's0))
     (define local-register-1 local-value-1)
     ...
     (define local-register-k local-value-k)
     (object/c ... (compiled t) ...)))
```

When it is evaluated, this expression creates a closure whose environment contains the global registers and their values. Every time this closure is invoked, it sets up a hidden variable for the *current* state and the local registers; the closure's result is an object contract that enforces the legality of method calls as specified in `t`.

Using Culpepper's `syntax-parse` system [11, 12], it is surprisingly simple to specify this embedded language. The first syntax definition in figure 5 specifies the most complex case of the syntax transformation. Technically, the `define-syntax` form defines a syntax transformer in the current scope, that is, it extends the Racket compiler so that it can handle instances of the `protocol` form. If the current scope is a module, the creator of the module can make this new syntax available to client modules via a provide provision—almost like ordinary bindings, except that the Racket compiler knows to use syntax transformers at compile time.

Here the syntax definition employs `syntax-parse` to specify how the Racket compiler should parse and analyze the form and, if successful, what kind of Racket code it should generate. Not counting the `#:literals` declaration, the `syntax-parse` form consists of a series of clauses, which, in turn, consist of two parts: a parsing *pattern*, which describes what the new syntax looks like, and a *template*, which describes what the generated code looks like. For readability, figure 5 separates the two with a comment.

Essentially `syntax-parse` pattern-matches the given syntax argument (`stx`) of `protocol` against the patterns in its clauses. If a match is found, the Racket compiler creates a substitution environment from the pattern variables, which are all those identifiers in a template that are not declared literals. Unless an analysis is required, the Racket compiler finally uses the substitution environment to fill the template, that is, to replace the pattern variables in the code-quoted template (`#'template`) with their current values.

A context-free syntax analysis is implemented via *syntax class* annotations on pattern variables. For example, the pattern variables following the literal `define-global-registers` have the suffix `:binding`. Roughly speaking, a syntax annotation triggers a context-free analysis of the matched expression. Here is the declaration for `binding`:

```
(define-syntax-class binding
   #:description "binding"
   (pattern [name:id val:expr]))
```

It says a binding must consist of a parenthesized pair: `name` and `val`. Both are annotated with built-in syntax classes; hence `name` must be an identifier and `val` must be an expression. Syntax classes may perform arbitrarily complex analysis computations. More importantly, though, they also deconstruct a pattern variable into its constituents. For `binding`, these are the `name` and the `val` parts.

If an annotated pattern in a template comes with a `.` suffix, it extracts a piece selected by the syntax class. Hence, `global.name` extracts the `name` part of the syntax bound to the `global` pattern variable while `global.val` extracts the `val` part.

At this point, only one aspect of the `protocol` implementation requires an explanation. The last line of the pattern specifies the syntax of the transition function, which consists of a sequence of transition rules. Each rule must have the form

```
([---> (calling:id param:id ...)  next] side ...)
```

where `calling` is the name of a state/method, `(calling param ...)` is a method header for this method, and `next` specifies which methods are legally called next. Finally, `side ...` represents the sequence of side-condition and actions, whose shape is not interesting yet.

The last line in `protocol`'s template synthesizes (the code for) an object contract. For each transition rule, the contract comes with a clause that specifies a method. The name of the method is `calling`. To generate the contracts for these methods, the `protocol` syntax transformer delegates to another syntax transformer, `transition`.

Now consider the second syntax definition in figure 5. It specifies that a transition rule of this shape

```
(transition
 current-states
 calling-state
 next-state
 (para-1 ... para-n)
 side-condition-1 ... side-condition-k)
```

is translated into the following contract:

```
(->i ([this any/c]
      [param-1 any/c]
      ...
      [param-n any/c])
     #:pre/name "description 1" (param ...)
       (compiled side-condition-1 ... states ...)
     ...
     #:pre/name "description k" (param ...)
       (compiled side-condition-k ... states ...)
     any/c)
```

This method contract allows any value for the *n* parameters and also attaches an `any/c` contract to `this`. Following the method header, the contract adds *k* preconditions. We use `#:pre/name` to make the contract readable for our readers; the actual code uses variants that synthesize informative error message.

Note how the pattern of `transition` deconstructs the `side` pattern variables from `protocol`.

Finally, let us sketch how `side`s from transition rules turn into the expected preconditions in the method contract. Consider the `"now-called"` clause. It says that the condition may depend on all of the methods parameters (but actually does not); the actual code merely checks whether the name of the method that is to be called, `'calling`, is a member of the list of methods that can legally be called. Similarly, an `#:update` clause translates into a conditional assignment statement in the `"updates"` precondition. The `u` pattern variable consists of three parts: `u.test`, which is the condition for the update; `u.name`, which is the register to be updated; and `u.val`, which is the expression that computes the new value.

The expressions in `side` may use locally defined names, e.g., `maybe-id->list` in the `"next state"` clause, and they may use names defined in the client modules, e.g., `player-id` in `next`. Racket's syntax system keeps track of these relationships and guarantees proper name resolutions.

In short, protocols look like a generalization of regular object and method contracts. And Racket's syntax system enables programmers to directly implement protocol as extensions of the contract system. At the same time, the DSL provides a linguistic abstraction for clients and a single point of control for implementers.

4. "I am special, so special"

Contracts occupy a unique position when it comes to formulating and enforcing program properties. As such, they naturally relate to types and run-time verification tools. Both of those attract a lot of attention from researchers and practitioners and, in contrast to contract systems, they have seen many different incarnations in practical tools and languages. Using the ontology of section 2, this section argues that the three approaches are not competitors but complements of each other. In other words, contracts truly differ from both type systems and run-time verification tools, and like those they come with their own strengths and weaknesses.

4.1 Contracts and Types

For any particular (combination of) contract(s), a hard-working researcher can design a structural type system for verifying the same logical properties statically. We doubt, however, that there exists a single, easy-to-use type system that can emulate all such uses of contracts. The attempts [24, 31] to create such a type system offers some evidence for our doubts.

One reason for our conjecture is that contract DSLs compose more easily than type systems. Both enforcing and verifying program properties requires the propagation of information across a (running) program, even when the property concerns the behavior of a single interaction. With type systems this means that type information must be passed around in the form of typing and effect tags throughout a typing derivation. In contrast, contracts use the normal flow of values in a program to disseminate such information. The result is that different type system features require at least a Cartesian product of complexity when combined, whereas different combinations of contracts (DSLs) are (mostly) orthogonal.

Furthermore, there is an inherent, wide gap between the reasoning needed to use a particular static type system and the reasoning needed to understand a program's execution. Ideally a programmer thinks logically about code, and a good type system should serve to render the operational behavior accessible. But, while programmers *must* understand the operational behavior code in order to write any programs at all, they do not need to understand *any particular* type system to do so. Thus, because contracts can be understood in a solely operational manner, they present a lower barrier to entry for a programmer than a type system.

Another way to understand the difference is to start from the type-theoretic view of types as propositions. Concretely, type theory views a type as a proposition about the code. The type checker

constructs a proof that the proposition holds. As long as the types of the contextual connections are satisfied, the expression satisfies the property that the type describes—regardless of the concrete nature of the context.

In comparison, the contract specifies a logical assertion that governs an expression and the use context of its value. The question of whether a expression satisfies its contract alone, does not denote; contract satisfaction is a ternary relation of the contract, the expression, and its concrete context [13]. Especially the latter element suggests that contracts can express and enforce more sophisticated program properties than types. Contracts are viable in scenarios where simple types cannot help and where sophisticated type system deteriorate into contract checking anyway. For examples, consider properties that depend on file I/O and network traffic.

To make this idea even more concrete, consider the contract of `map/e` from section 3.1:[10]

```
(->i ([in  (e c) (-> (enum-contract e) c)]
      [out (e c) (-> c (enum-contract e))]
      [e enum?]
      #:contract [c contract?])
      ; appears to be a bijection:
      #:pre/desc (in out e) (bijection?? in out e)
      [result enum?])
```

This higher-order contract intercepts two events for each use of `map/e`: (1) the call to `map/e` and (2) the return of the call. For the call event, the contract system collects the arguments `in`, `out`, `e`, `c` as the event information. For the return event, it collects the result of `map/e`. With `->i`, the programmer uses several hooks to insert predicates that check properties of the collected information. There are the hooks for each of the three arguments and the hook for the result. In addition, the pre-condition relates all three arguments. Similarly, the property for `in` depends on `e` and `c` along with `in`. Furthermore, because `in` and `out` are functions, the programmer can hook in sub-contracts, directing the contract system to collect further events and information. In particular, the hook for `in` specifies that the contract system must monitor `in` as a function with appropriate domain and range, and so on. In sum, the above contract describes a sophisticated desirable property of `map/e` in terms of the familiar operational semantics of our programming language.

Compare this operational understanding to a type-oriented formulation. The effort required would be significant. Types would be propositions in some formal logic that the type-checker tries to prove by analyzing a piece of code, e.g., `map/e`. The success of type-checking is sensitive to the structure of the code. It is thus common that for the benefit of the type-checker, programmers have to organize their code in a particular manner. Also, usually programmers have to embed in their code a significant amount of annotations as hints to help the type-checker. Hence, during type-checking, the programmer must switch from thinking in terms of the operational semantics of the language to thinking in terms of the logical foundations of the type system. The same is true even in non-structural type systems such as those based on liquid types [50] or manifest contracts [31]; what properties of the program the programmer can express, and how, is a function of the logical foundations of the type system. For types that state non-trivial properties, such as the dependent properties and the bijectivity property of the above example, no amount of hints, auxiliary definitions or restructuring of the code is sufficient. Programmers have to use a dependent type system that is parametric to proof objects, such as some version of CoC [10], and escape to a proof assistant, such as Coq [59], to convince the type-checker to admit the type. Beyond that, additional proofs are required for every use of `map/e`.

The comparison also points out the deficits of contract systems. First, contracts enforce much weaker properties of higher-order programs than types. A contract system can prove safety properties only with respect to concrete execution traces. In fact, a contract system, unlike a type-checker, cannot prove any higher-order property of a piece of code. For instance, the fact that `integer/e` in section 3.1 does not raise a contract violation, does not imply that `integer/e` is a bijection or even that `integer/e` is a function whose domain is the naturals. The lack of a contract violation means simply that our example has not resulted in a counter-example concerning the bijection property of `integer/e`. This operational nature of contract systems also explains why tools such as quotient and denotational semantics explain so little about contract systems [7, 23], while they have been so successful in the type-theoretic world. Instead, syntactic techniques have helped build frameworks for reasoning about the correctness of contract systems, including their correct behavior when contract checks fail [14, 15]. Second, while writing and using contracts is cheap for the programmer, dynamic contract checking is often expensive. Unlike type-checking, it inflicts both time and memory cost which may become counter-productive as contracts capture computationally complex properties or as the number of contract checks, even for simple contracts, increases. Recent work on symbolically verifying first-class contracts might combine the best of both worlds [48, 49].

Finally, contract attachment implies one additional distinction between types and contracts. Any module can attach a contract to any binding including re-exports of imported bindings. Such an attachment means "blame" works differently for types and contracts. When a type-checker detects a type error, the type-error points to an inconsistency between the type system and the piece of code that does not type-check; the same connection does not hold for contract violations. Concretely, if a programmer imports `map/e` in a module, attaches to it a contract other than the one the enumerations library picked, and that contract latter fails, the responsibility of the failure can *not* be with the enumerations library. Put differently, the programmer should not look at the enumerations library to figure out what went wrong. Nor is the place at fault that triggered the contract violation. Instead, as the discussion about blame in section 2 says, a contract is an agreement between the server module that attaches a contract to a binding and the contract's scope. When a contract fails, the bug is either in the server or in the contract's scope independently of which module contains the code that triggered the contract violation.

The subtle meaning of blame is often misunderstood when approached from a type-theoretic viewpoint. For example, Wadler and Findler [60]'s paper contains the catchy but simplistic slogan "well-typed programs can't be blamed". This slogan simplifies the situation so much that it is effectively false. Consider the case where a typed module imports a function from an untyped module and asserts that the function has some type. Furthermore, assume that the underlying gradual type system generates the correct contract from the asserted type and attaches it to the imported binding. If the typed module now uses the function and the contract fails, soundness of the gradual type system (correctly) implies that the typed code cannot have *used* the untyped function in a way that violates the generated contract. But, in line with our discussion about blame assignment, the responsibility for the contract failure lies with either the server module that picked which contract to attach to the untyped function or the scope of the contract. In this case, server and the scope *are one and the same*: the typed module. After all, it assumed unwisely that the untyped function lives up to a contract about which the function's author knows nothing.

[10] The actual contract is more involved as `map/e` can consume more than one enumeration.

4.2 Contracts and Run-Time Verification

Run-time verification (RV) is another specification tool that shares several characteristics with contracts. In particular, the analysis of contract systems in section 2 has many similarities with the high-level architecture of run-time verification tools. Like contract systems, run-time verification tools install probes that react to events such as method calls and returns and reify relevant information such as method arguments and results. These probes collect the information about the events and check whether it is consistent with the specification of the code. RV specifications are invariably expressed in some DSL based on a formal logic, usually but not always a variant of temporal logic.

Like type systems research, RV research is in a much more mature state than contract system research. During the last 20 years, a co-ordinated research effort has resulted in a variety of results: theoretical foundations, distinct application domains, specification DSLs with clear expressiveness, synthesis algorithms for monitors with thoroughly evaluated performance characteristics, and various architectures for run-time verification tools. The large number and variety of pragmatically validated tools (e.g. Allan et al. [2], Avgustinov et al. [3], Barringer et al. [4], Bodden [8], Chen and Roşu [9], Drusinsky [17], Gates et al. [29], Goldsmith et al. [30], Havelund and Roşu [34], Kim et al. [37], Marcelo D'Amorim and Havelund [40], Martin et al. [41]) are witness of these outcomes.

At first glance, a run-time verification tool seemingly consists of layers analogous to those of a contract system, and research on RV appears to subsume similar efforts in the world of contracts. Indeed, this statement probably holds for first-order contract systems such as those of Eiffel, Jass [5], J-Contractor [1] and JML [39]. Once contracts move into the higher-order world, however, significant differences emerge between RV tools and contract systems.

One important distinction between RV and contracts is both historical and teleological. While contracts focus on interfaces that describe functional correctness properties of software components, run-time verification turns to temporal properties of execution traces of complete software systems. Linguistic research on contracts that enforce temporal properties of components is pretty recent and limited [16, 51]. In contrast, research on how specification DSLs can capture a wide range of trace properties and how monitors can enforce them performantly, is the heart and soul of RV research. At the same time, though, RV systems cannot cope with properties of advanced language constructs such as anonymous functions, first-class classes, modules, mutable references, (delimited) continuations and/or continuation marks [42, 53] if those are used to specify API interactions. In other words, RV offers no help with specifying rich interfaces for components that facilitate the correct composition and reuse of components in different contexts.

Coping with such powerful language constructs has been the main goal behind the introduction of contracts into modern programming languages [22] and has driven the evolution of contract systems (e.g., Strickland et al. [55], Strickland et al. [57], Takikawa et al. [58]). An interesting outcome of this deviation in goals is that contract systems live inside of programming languages while run-time verification tools live outside. In other words, higher-order contract systems are extensions of programming languages, and run-time verification systems are elements of a language's external tool chain.

A second distinction is technical but closely related to this last point. The exo-linguistic nature of run-time verification tools makes it hard to express properties of programs for a delimited scope. With respect to scope, existing RV specification languages offer two kinds of properties: global properties, e.g., the contents of a particular variable, or properties that are universal for all instances of a particular class, e.g,. the methods of a particular class are called in a specific order during the execution of a program. Recently, significant ingenuity and effort has been invested in refining global class properties so that objects of a class (or groups of objects of different classes) get associated with their own instance of a global class property, e.g., each object receives calls to its methods in a particular order [43]. Still, properties that are "active" for a single object or function and only in a particular program context are beyond the reach of existing RV techniques.

In contrast, programmers can easily specify such properties with contracts as section 2 and section 3 amply demonstrate. Programmers can attach a contract to a class that captures a property that spans all the instances of the class, or a property that all objects of a class should adhere too individually. Programmers can also attach a contract to an object that captures a property of that particular object independently of the contracts of the object's class. And finally programmers can attach a contract for a class or an object only for a particular program context such as the body of a definition or a closure. This power of contract systems is due to their seamless integration with their host languages. Thus programmers have at their disposal means to connect contracts with the scoping mechanisms of the language and the structure of programs.

5. "Do you understand me now?"

A full appreciation of contracts demands a proper analysis of the underlying contract system and all of its pieces. While these pieces are implicitly in numerous publications on higher-order contracts, nobody else has put them into perspective in quite the way this song does, namely, as an interlocking system of an interposition layer, a base of functional combinators, and an attachment mechanism that establishes the scope of contracts and its obligations. With this understanding in place, it becomes clear that programmers can use the rest of the host language's expressive power to build truly sophisticated contracts. The examples of section 3 illustrate this point, but given the power of higher-order functional languages, more is likely to come.

To put this idea in context, all of us must appreciate programming as an optimization problem with many parameters: the desirable reliability of the code, the size and quality of the development team, the available resources, etc. Ideally, programmers ought to have contracts and types and run-time verification at their disposal. Depending on the situation, they could then decide how to best satisfice [52] the optimization problem. Indeed, as each of these tools has its own strengths and weaknesses, programming language researchers ought to figure out how all these tools could coexist so that programmers can smoothly transition from one to the other as the requirements change.

Acknowledgment

The authors gratefully acknowledge the support of several NSF grants: CCF-1421770 and CNS-1524052 (Harvard), CNS-1405756 (Northwestern), and CCF-1518844 (Northeastern).

References

[1] Parker Abercrombie and Murat Karaorman. jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java. *Electronic Notes in Theoretical Computer Science*(70(4)), pp. 55–79, 2002. Presented in RV 2001, Run-time Verification (Satellite Workshop of FLoC '02)

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sacha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 345–364, 2005.

[3] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making Trace Monitors Feasible. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 589–608, 2007.

[4] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule Systems for Run-time Monitoring: From Eagle to Ruler. In *Proc. International Conference on Runtime Verification*, pp. 111–125, 2007.

[5] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass — Java with Assertions. *Electronic Notes in Theoretical Computer Science*(55(2)), pp. 103–117, 2001. Presented in RV 2001, Run-time Verification (Satellite Workshop of CAV '01)

[6] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Contract Aware Components, 10 years after. *Electronic Proceedings in Theoretical Computer Science*(7), pp. 1–11, 2010.

[7] Matthias Blume and David McAllester. Sound and Complete Models of Contracts. *Journal of Functional Programming* 16(4-5), pp. 367–414, 2006.

[8] Eric Bodden. *J-LO, a Tool for Runtime-Checking Temporal Assertions*. RWTH Aachen University, 2005. Master's Thesis.

[9] Feng Chen and Grigore Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. *Electronic Notes in Theoretical Computer Science*(89(2)), pp. 108–127, 2003. Presented in RV 2003, Runtime Verification (Satellite Workshop of CAV '03)

[10] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Compuation*(76:2-3), pp. 95–120, 1988.

[11] Ryan Culpeppeer. Fortifying Macros. *Journal of Functional Programming*(22(4/5)), pp. 439–476, 2012.

[12] Ryan Culpeppeer and Matthias Felleisen. Fortifying Macros. In *Proc. ACM International Conference on Functional Programming*, pp. 235–246, 2010.

[13] Christos Dimoulas and Matthias Felleisen. On Contract Satisfaction in a Higher-Order World. *Transactions on Programming Languages and Systems* 33(5), pp. 16:1–16:29, 2011.

[14] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct Blame for Contracts: No More Scapegoating. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 215–226, 2011.

[15] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete Monitors for Behavioral Contracts. In *Proc. European Symposium on on Programming*, pp. 214–233, 2012.

[16] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal Higher-order Contracts. In *Proc. ACM International Conference on Functional Programming*, pp. 176–188, 2011.

[17] Doron Drusinsky. Temporal Rover. 2010. `http://www.time-rover.com`

[18] Matthias Felleisen. On the Expressive Power of Programming Languages. *Science of Programming* 17, pp. 35–75, 1991.

[19] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[20] Robert Bruce Findler and Matthias Blume. Contracts as Pairs of Projections. In *Proc. International Conference on Functional and Logic Programming*, pp. 226–241, 2006.

[21] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a Programming Environment for Scheme. *Journal of Functional Programming* 12(2), pp. 159–182, 2002.

[22] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM International Conference on Functional Programming*, pp. 48–59, 2002.

[23] Robert Bruce Findler, Matthias Felleisen, and Matthias Blume. An Investigation of Contracts as Projections. University of Chicago, Computer Science Department, TR-2004-02, 2004.

[24] Cormac Flanagan. Hybrid Type Checking. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 245–256, 2006.

[25] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. ACM International Conference on Functional Programming*, pp. 109–120, 2009.

[26] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with Classes, Mixins, and Traits. In *Proc. Asian Symposium on Programming Languages and Systems*, pp. 270–289, 2006.

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

[28] Martin Gasbichler and Michael Sperber. Integrating User-Level Threads with Processes in Scsh. *Higher Order and Symbolic Computation*(18(3-4)), pp. 327–354, 2005.

[29] Ann Q. Gates, Steve Roach, Oscar Mondragon, and Nelly Delgado. DynaMICs: Comprehensive Support for Run-Time Monitoring. In *Proc. International Conference on Runtime Verification*, pp. 164–180, 2001.

[30] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational Queries over Program Traces. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 385–402, 2005.

[31] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts Made Manifest. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 353–364, 2010.

[32] Guy Lewis Steele, Jr. Macaroni is Better Than Spaghetti. In *Proc. Symposium on Artificial Intelligence and Programming Languages*, pp. 60–66, 1977.

[33] Guy Lewis Steele, Jr. The Revised Report on SCHEME: A Dialect of LISP. Massachusetts Institute of Technology Artificial Intelligence Laboratory, AIM-452, 1978.

[34] Klaus Havelund and Grigore Roşu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*(55(2)), pp. 200–217, 2001. Presented in RV 2001, Run-time Verification (Satellite Workshop of CAV '01)

[35] Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. Access Permission Contracts for Scripting Languages. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 112–122, 2012.

[36] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proc. European Symposium on on Programming*, pp. 122–138, 1998.

[37] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods in System Design*(24(2)), pp. 129–155, 2004.

[38] John Lamping. Typing the Specialization Interface. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 201–214, 1993.

[39] Gary T. Leavens. JML's Rich, Inherited Specifications for Behavioral Subtypes. In *Proc. Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods*, pp. 2–34, 2006.

[40] Marcelo D'Amorim and Klaus Havelund. Event-based Runtime Verification of Java Programs. In *Proc. Workshop on Dynamic Analysis*, pp. 1–7, 2005.

[41] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 365–383, 2005.

[42] Jay McCarthy. The Two-state Solution: Native and Serializable Continuations Accord. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 567–582, 2010.

[43] Patrick Meredith and Grigore Roşu. Efficient Parametric Runtime Verification with Deterministic String Rewriting. In *Proc. ACM/IEEE International Conference on Automated Software Engineering*, pp. 70–80, 2013.

[44] Bernard Meyer. Applying Design by Contract. *IEEE Computer* 25(10), pp. 45–51, 1992.

[45] Bernard Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[46] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. Shill: A Secure Shell Scripting Language. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, pp. 183–199, 2014.

[47] James Hiram Morris. Lambda-Calculus Models of Programming Languages. Ph.D. dissertation, Massachusetts Institute of Technology, 1968.

[48] Phúc Nguyễn and David Van Horn. Relatively Complete Counterexamples for Higher-order Programs. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 445–456, 2015.

[49] Phúc Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. Soft Contract Verification. In *Proc. ACM International Conference on Functional Programming*, pp. 139–152, 2014.

[50] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 159–169, 2008.

[51] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Computational Contracts. *Science of Computer Programming*(98:3), pp. 360–375, 2015.

[52] Herbert A. Simon. *Administrative Behavior*. MacMillan, 1947.

[53] Vincent St-Amour, Leif Andersen, and Matthias Felleisen. Feature-specific Profiling. In *Proc. Compiler Construction*, pp. 49–68, 2015.

[54] Robert E. Storm and Shaula A. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*(12(1)), pp. 157–171, 1986.

[55] T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa, and Matthias Felleisen. Contracts for First-Class Classes. *Transactions on Programming Languages and Systems* 35(3), pp. 11:1–1:58, 2013.

[56] T. Stephen Strickland and Matthias Felleisen. Nested and Dynamic Contract Boundaries. In *Proc. International Conference on Functional and Logic Programming*, pp. 141–158, 2009.

[57] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 943–962, 2012.

[58] Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. Constraining Delimited Control with Contracts. In *Proc. European Symposium on on Programming*, pp. 229–248, 2013.

[59] The Coq Cevelopment Team. The Coq Proof Assistant Reference Manual. LogiCal Project, Version 8.0, 2004.

[60] Philip Wadler and Robert Bruce Findler. Well-typed Programs Can't be Blamed. In *Proc. European Symposium on on Programming*, pp. 1–15, 2009.