# Chaperones and Impersonators: Run-time Support for Contracts on Higher-Order, Stateful Values

T. Stephen Strickland
Sam Tobin-Hochstadt
Northeastern University
{sstrickl,samth}@ccs.neu.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

## Abstract

Racket's *chaperone* and *impersonator* language constructs provide run-time support for implementing higher-order contracts on mutable structures and abstract datatypes. Using chaperones and impersonators, contracts on mutable data can be enforced without changing the API to that data; contracts on large data structures can be checked lazily on only the accessed parts of the structure; contracts on objects and classes can be implemented with lower overhead; and contract wrappers can preserve object equality where appropriate. With this extension, Typed Racket, which relies on contracts for interoperation with untyped code, can now pass mutable values safely between typed and untyped modules.

Although almost any proxy mechanism can support contracts, an unrestricted proxy layer would enable external code to violate the internal invariants of Racket libraries. Chaperones and impersonators are sufficiently expressive to implement contracts—allowing the remainder of the contract system to be implemented outside of the core run-time system—while preserving the abstraction mechanisms and reasoning that both Racket programmers and compiler analyses rely on. The paper includes a model of Racket with mutable data, chaperones, and impersonators, and proofs that the model maintains desired invariants.

## 1. Contracts in an Extensible Language

An extensible progamming language like Racket enables third-party programmers to design and maintain seemingly core aspects of a programming language, such as a class system, a component system, or a type system. At the same time, extensibility creates constant pressure on the design of the core language to support new and different forms of extension. The core must evolve to support new forms of extension without exposing power to compromise abstractions and therefore inhibit composition.

The Racket (Flatt and PLT 2010) contract system is a prime example of this trade-off in extensibility versus composition. The contract system can exist in its rich, state-of-the-art form largely because it can be implemented, modified, and deployed without requiring changes to the core run-time system and compiler. At the same time, since the contract system's job is to help enforce invariants on functions and data, language extensions can accidentally subvert the intent of the contract system if the Racket core becomes too extensible or offers too much reflective capability.

In this paper, we report on an addition to the Racket core that is driven by the need for a better and more extensible contract system, but where there exists an acute possibility of subverting core guarantees of the language. To properly enforce contracts on certain types of data, the contract system needs a mechanism for *interposition* on core operations—or *intercession*, in the terminology of the CLOS Metaobject Protocol (Kiczales et al. 1991)—so that use of those operations can trigger contract checks. For example, if a mutable vector has a contract on its elements, every use or modification of the vector should be guarded by a check. An up-front check does not suffice, because the vector may be modified concurrently or through a callback. Although the contract system needs interposition, the availability of unrestricted interposition could lead to more invariants broken than enforced.

To balance the needs of extensibility and composition, we have developed a two-layer system of *chaperones* and *impersonators*. Chaperones and impersonators are both kinds of proxies, where a wrapper object interposes on operations intended for a target object. Chaperones can only constrain the behaviors of the objects that they wrap; for an interposed operation, a chaperone must either raise an exception, return the same value as the original object would have returned for the operation, or return a chaperone of the original object's result. Impersonators, in contrast, are relatively free to change the apparent behavior of the objects that they wrap; while an impersonator usually behaves the same as the original object and returns the same values for interposed operations, the impersonator may behave differently or return different values. Overall, impersonators can be more expressive, but chaperones are allowed on more kinds of values.

Together, chaperones and impersonators are powerful enough to implement an improved contract system without subverting guarantees that enable composition of language extensions. Thanks to chaperones and impersonators, the Racket contract system now supports higher-order contracts on mutable objects and abstract datatypes. This improvement directly benefits Racket programmers, and it benefits language extensions that are further layered on the contract system—notably Typed Racket (Tobin-Hochstadt and Felleisen 2008), which became more interoperable with untyped Racket as a result of the improvements.

Last but not least, building just enough contract support into the core compiler and run-time system offers the promise of better performance, both for code that uses contracts and code that does not. Core-language support for interposition has already eliminated a factor of three slowdown for some object-oriented operations; this slowdown was present even if the program did not use contracts at all. Contract support for class-based objects now affects performance only for programs that use contracts.

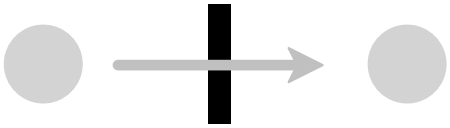
Section 2 motivates the conceptual need in contract checking for chaperones and impersonators. Section 3 describes the implementation of contracts in terms of chaperone and impersonator constructs that (unlike contracts) are built into the run-time system. Section 4 describes a formal model for a subset of Racket with chaperones and impersonators and proves that they preserve key invariants of the base language. Section 5 gives some performance numbers, and section 6 discusses related work.

## 2. Contracts in Racket

We start with a review of predicate and function contracts in Racket, and then we explain how a generalization of the contract system to more kinds of data leads to new categories of contracts.

### 2.1 Predicates and Function Contracts

A contract mediates the dynamic flow of a value across a boundary:



In Racket, contracts most often mediate the boundaries between modules. For example, the left and right bubbles above could correspond to `math.rkt` and `circles.rkt` modules declared as

`math.rkt`

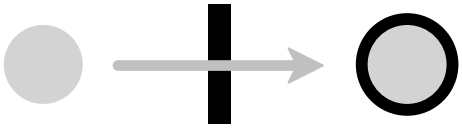
```
(define pi (* (acos 0) 2))
(provide/contract
 [pi real?])
```

`circles.rkt`

```
(require "math.rkt")
pi
```

The circle on the left is the value `3.141592653589793` as bound to `pi` in `math.rkt`. The dividing line in the picture is the contract `real?`, which checks that the value of `pi` is a real number as it crosses to the area on the right. The circle on the right is the successful use of `pi`'s value in `circles.rkt`, since `3.14...` is a real number.

Not all contract checks can be performed immediately when a value crosses a boundary. Some contracts require a delayed check (Findler and Felleisen 2002), which is like a boundary wrapped around a value:



Delayed checks are needed for function contracts, such as when `math.rkt` exports a `sqr` function that is used by `circles.rkt`:

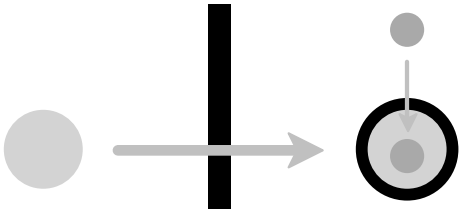
`math.rkt`

```
(define (sqr x) (* x x))
(provide/contract
 [sqr (real? . -> .
       nonnegative-real?)])
```

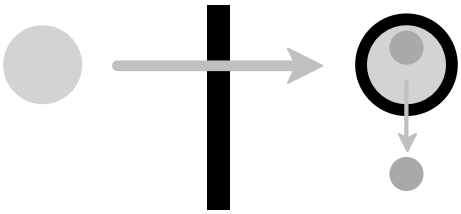
`circles.rkt`

```
(require "math.rkt")
(map sqr ....)
```

In this case, an immediate check on `sqr` cannot guarantee that the function will only be used on real numbers or that it will always return non-negative real numbers. Instead, when `sqr` is applied to a value, the value going into `sqr` crosses the wrapper boundary and is checked to ensure that it is a real number:



Similarly, when a call to `sqr` returns, the value going out of `sqr` crosses the wrapper boundary and is checked to ensure that it is a non-negative real number:



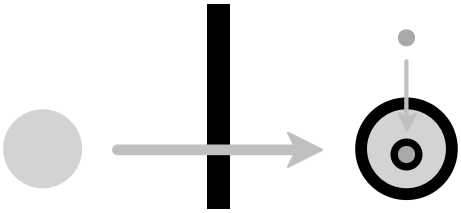
When a function consumes another function as an argument, then a check-delaying wrapper on the consuming function can add delayed checks to the argument function. For example, a numerically approximating `derivative` consumes a function that must accept real numbers and return real numbers:

`math.rkt`

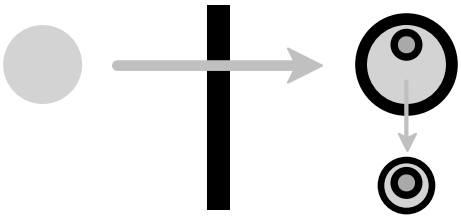
```
(define (derivative f)
  ....)
(provide/contract
 [derivative
  ((real? . -> . real?)
   . -> .
   (real? . -> . real?))])
```

`circles.rkt`

```
(require "math.rkt")
(derivative cos)
```



Along the same lines, a function may capture a wrapped function in its closure. The derivative function produced by `derivative`, for example, closes over the argument with the argument's wrapper intact:



Other kinds of wrappers can implement contracts that guarantee a kind of parametricity for functions. Using `new-∀/c`, for example, the left-hand `poly.rkt` module can promise that its `id` function returns only values that are provided to the function:

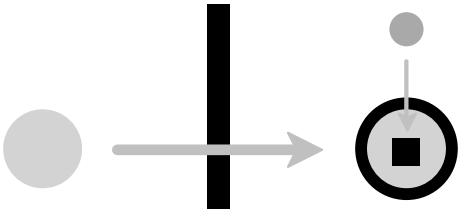
`poly.rkt`

```
(define (id x) x)
(provide/contract
 [id (let ([α (new-∀/c)])
       (α . -> . α))])
```

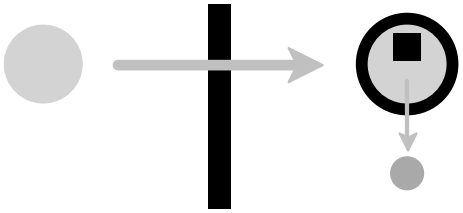
`client.rkt`

```
(require "poly.rkt")
(id 199.99)
```

When the function is called by the right-hand module, the argument to `id` is wrapped to make it completely opaque:



When `id` returns, the result value is checked to have the opaque wrapper, which is removed as the value crosses back over the function's boundary:[1]



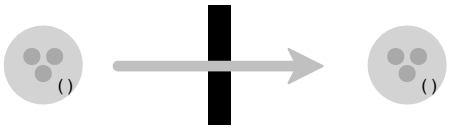
As originally implemented for Racket, simple predicate contracts, function contracts, and even `new-∀/c` require no particular run-time support; function wrappers are easily implemented with $\lambda$ and opaque wrappers via Racket's `struct` form. Run-time support becomes necessary, however, to generalize contracts beyond immediate predicates and function wrappers.

### 2.2 Compound-Data Contracts

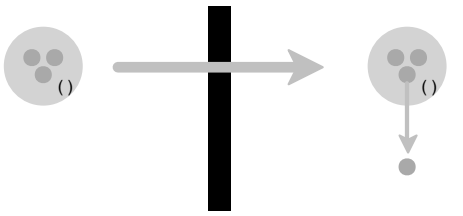
Lists are almost as common in Racket as functions, and list contracts are correspondingly common. In simple cases, the contract on a list can be checked immediately, as in the case of a list of real numbers:

```
math.rkt
(define constants
  (list 8 6.02e+23 6.6e-11))
(provide/contract
 [constants
  (listof nonnegative-real?)])
```

```
circles.rkt
(require "math.rkt")
constants
```




The "`()`" badge on the circle is meant to suggest "list." If the list content is checked completely as it crosses the contract boundary, elements can be extracted from the list with no further checks:



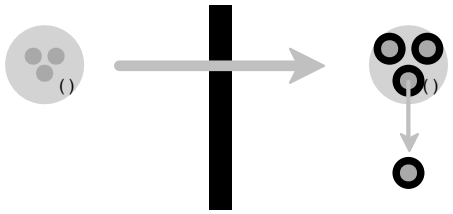

In the case of a list of functions, the list shape of the value can be checked immediately, but the functions themselves may require wrappers. After such a list crosses the contract boundary, the right-hand module sees a list of wrapped functions, and the wrappers remain intact when functions are extracted from the list:

```
math.rkt
(define transforms
  (list identity sqr sqrt))
(provide/contract
 [constants
  (listof
   (nonnegative-real?
    . -> . nonnegative-real?))])
```

```
circles.rkt
(require "math.rkt")

(first transforms)
```

[1] Matthews and Ahmed (2008) show that this wrapper protocol preserves parametricity.

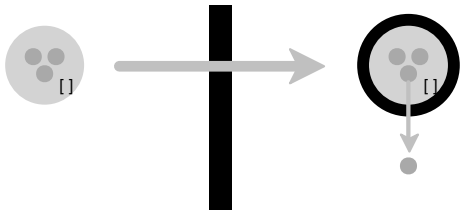



Wrapping the list instead of its elements can be more efficient in some situations (Findler et al. 2007), but the element-wrapping approach is effective to checking the contract. Wrapping the elements of a mutable vector (array), however, does not work:

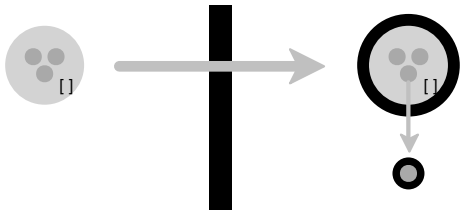
```
math.rkt
(define state
  (vector 0.1 0.4 7.9))
(provide/contract
 [state nonnegative-real?])
```

```
circles.rkt
(require "math.rkt")
state
```

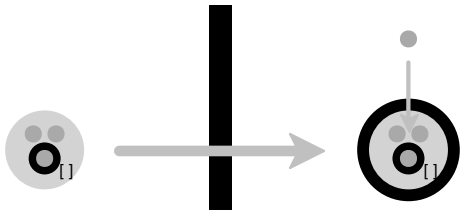
Since the `state` vector is mutable, the intent may be that the left-hand `math.rkt` module can change the values in `state` at any time, with such changes visible to the right-hand module. Consequently, values must be checked at the last minute, when they are extracted from the vector in the right-hand module:




The "`[]`" badge on the circle is meant to suggest "vector." Similarly, any value installed by the right-hand module must be checked as it goes into the vector. If the vector contains functions instead of real numbers, then extracting from the vector may add a wrapper.




Finally, installing a function into the vector may also add a wrapper:




In this last case, since both sides of the module boundary see the same mutable vector, the newly installed function has a wrapper when accessed from the left-hand `math.rkt` module. That wrapper allows the left-hand module to assume that any function it calls from the vector will return a suitable result, or else a contract failure is signalled. Similarly, if the left-hand module abuses the function by calling on an inappropriate argument, a contract failure protects any function that was installed by the right-hand module, as guaranteed by the contract on the vector.

### 2.3 Structure Contracts

Besides functions and built-in datatypes like lists and vectors, Racket allows programmers to define new structure types. Reliable structure-type opacity is crucial in the Racket ecosystem. Not only must ordinary user libraries have their internal invariants protected, but even seemingly core forms, such as $\lambda$ or `class`, are implemented as macros that use opaque structure types to implement the run-time aspect of the construct.

For example, a `widget.rkt` module might define a structure type for GUI widgets:

widget.rkt

```
(define-struct widget
  (parent label callback))
(define (widget-root w)
  (define p (widget-parent w))
  (if p (widget-root p) w))

(provide widget) ; for use in struct/c
(provide/contract
 [make-widget
  (-> (or/c widget? #f) string? (-> event? void?)
      widget?)]
 [widget-root (-> widget? widget?)]
 [widget-parent (-> widget? (or/c widget? #f))]
 [widget? (-> any/c boolean?)])
```

A module that declares a structure type can keep parts of the declaration private to enforce all widgets meet certain invariants. In our example, `widget.rkt` withholds the original `make-widget` function and provides one that restricts the values of the fields. In general, however, the `widget.rkt` module can hide the constructor completely and provide only a function that tracks all of the widgets in the system or fills in certain fields itself. The gain is that simply testing a value with `widget?` ensures the properties that were established by the widget-creation code.

Independent of properties that are checked when a structure instance is created, additional contracts can constrain specific widget instances. For example, the left-hand `scene.rkt` uses the `struct/c` contract to promise that `plot` is an instance of the `widget` struct whose first field is an OpenGL window. The contracts on the other fields perform no checking, while the contract on the constructor from `widget.rkt` is still in force.

scene.rkt

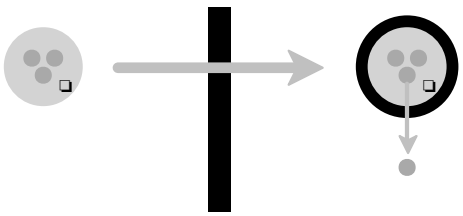
```
(require "widget.rkt")
(define plot (make-widget ....))
(provide/contract
 [plot (struct/c widget
                 gl-widget?
                 any/c any/c)])
```

circles.rkt

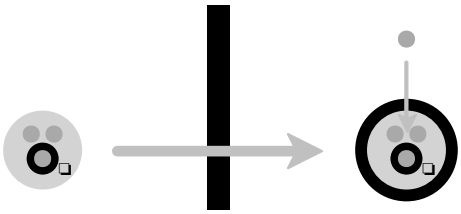
```
(require "widget.rkt"
         "scene.rkt")

(widget-parent plot)
```

The `plot` contract's promise can be checked through a wrapper on the widget when the right-hand module accesses the parent field of the widget:



The left-hand module can similarly constrain any change to the widget's callback function, which may require a wrapper on the function as it is installed into the widget:



As in the case of vectors, the wrapper resides on the function even when it is extracted by the left-hand module, thus ensuring the requirements on the function that the left-hand module imposed through a contract.

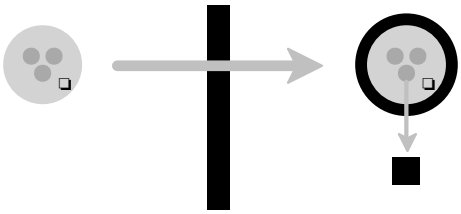
### 2.4 Parameteric Contracts and Generativity Don't Mix

Consider, however, the case where the left-hand module claims that the widget's parent must be treated parametrically:

scene.rkt

```
(require "widget.rkt")
(define plot (make-widget ....))
(provide/contract
 [plot (struct/c widget α any/c any/c)])
```

In this case, extracting the parent from the widget produces an opaque value:



This situation, created by a contract between the `scene.rkt` and `circles.rkt`, is unacceptable to the `widget.rkt` module that created the `widget` structure type, because the `widget-root` function can no longer traverse the hierarchy to find the root widget.

The run-time system must allow contract-enforcing wrappers on structures, but it must constrain wrappers to ensure that guarantees otherwise enforceable through a constructor are not subverted by the wrappers. In particular, the use of $\alpha$ in `scene.rkt` on a field of a `widget` is disallowed. Only contracts that act as *chaperones* may be placed on immutable fields of a structure. A chaperone contract defers all actions to the original values, except that it may sometimes prohibit an action by signalling an error. Since parametric contracts introduce opaque wrappers to values, they are not chaperone contracts.

The constraint on structures to chaperone contracts applies transitively to any wrapper applied at a chaperone-contract boundary. For example, just as `scene.rkt` cannot claim that the parent value is unknown, it cannot claim that the value consumed or produced by a callback procedure is unknown. The `scene.rkt` module can place an additional contract on the callback procedure, as long as the contracts on the consumed and produced values of a callback are themselves chaperone contracts.

### 2.5 Contract Hierarchy

Racket's previous contract system distinguishes two classes of contracts that relate to their implementation requirements:

- *Flat contracts* can be checked completely at boundaries, requiring no additional wrappers.
- *Higher-order contracts* require wrappers to delay checks.

In generalizing contracts to support compound data types and programmer-defined structure, we have refined the second class to two kinds of higher-order contracts:

- *Chaperone contracts* are normal wrapper-applying contracts, where checks at the wrapper boundary are themselves flat or chaperone contracts.
- *Impersonator contracts* are more extreme, like parametricity-enforcing contracts, potentially replacing a value at a boundary crossing with a different or completely opaque value.

This categorization is a hierarchy: a flat contract can be used wherever a chaperone contract is allowed, and any kind of contract can be used where an impersonator contract is allowed. Contract combinators such as `listof` and `->` create chaperone contracts when given chaperone contracts, and they create impersonator contracts when given at least one impersonator contract.

To summarize the impact of this work on the Racket contract library, the following table shows the state of contract support in Racket before our generalizations:

| | flat contracts | higher-order contracts |
|---|---|---|
| procedures | ✓ | ✓ |
| immutable data | ✓ | ✓ |
| mutable data | ~ | ✗ |
| opaque structures | ~ | ✗ |
| objects | ✓* | ✓* |

The tildes indicate points where flat contracts were allowed for mutable data and structures, but the contracts were actually checked only partly, because mutation could subvert the checks necessary to enforce a contract.

The asterisks on the "objects" line indicates that contracts were supported for our Java-like object system, but at a high runtime cost to uses of the object system that did not apply contracts. Many language extensions in Racket are built using macros and programmer-defined structure types, and they would likely suffer in the same way the object system did with the addition of contracts.

The following table shows the current state of Racket support for contracts after our generalizations:

| | flat contracts | chaperone contracts | impersonator contracts |
|---|---|---|---|
| procedures | ✓ | ✓ | ✓ |
| immutable data | ✓ | ✓ | ✗* |
| mutable data | ✓ | ✓ | ✓ |
| opaque structures | ✓ | ✓ | ✗* |
| objects | ✓ | ✓ | ✓ |

The asterisks here indicate where lack of support is a feature; those points in the spectrum do not make sense, as explained in the previous section. Contracts are fully supported and reliably checked in all other points of the space.

## 3. Chaperones and Impersonators

The Racket run-time system is oblivious to the contract system. To support chaperone contracts and impersonator contracts—enforcing requirements on the former and constraining the application of the latter—the run-time system provides *chaperones* and *impersonators* as first-class values. Chaperone contracts are implemented by the contract library in terms of chaperones, and impersonator contracts are implemented with impersonators.

Figure 1 shows part of the Racket chaperone and impersonator API.[2] The API includes a constructor for each primitive datatype that supports interposition on its operations. The `chaperone-of?`

[2] The complete API is about twice as large (Flatt and PLT 2010).

`(chaperone-of? a b)`
Determines whether *a* is the same as or a chaperone of *b*.

`(chaperone-procedure proc interp-proc)`
Chaperone a procedure, interposing on procedure arguments and results via *interp-proc*.

`(chaperone-vector vec interp-ref interp-set)`
Chaperone a vector, interposing on the `vector-ref` and `vector-set!` operations.

`(chaperone-struct struct op interp-op ....)`
Chaperone a structure instance, interposing on the supplied accessor and mutator operations for mutable fields.

...

`(impersonator-of? a b)`
Determines whether *a* is the same as, an impersonator of, or a chaperone of *b*.

`(impersonate-procedure proc interp-proc ....)`
Impersonate a procedure.

`(impersonate-vector vec interp-ref interp-set ....)`
Impersonate a mutable vector.

`(impersonate-struct struct op interp-op ....)`
Impersonate a structure instance.

...

Figure 1: Partial chaperone and impersonator API

predicate checks whether a value is a chaperone of another value—and therefore acceptable, for example, as a replacement result from another chaperone.

### 3.1 Chaperoning and Impersonating Functions

The `chaperone-procedure` function takes a procedure and creates a chaperone that also acts as a procedure. The chaperone accepts the same number of arguments as the original function, it returns the same number of results, and when it is called, the chaperone calls the original function. At the same time, when the chaperone is applied, it can check and possibly replace the arguments to original function or the results from the original function.

To chaperone a function, `chaperone-procedure` needs the function to chaperone and a function to filter arguments and results:

```
(chaperone-procedure orig-proc interpose-proc)
```

For example, to chaperone a function of two arguments, the filtering *interpose-proc* would have the form

```
(λ (a b) .... (values new-a new-b))
```

where `a` and `b` are the arguments originally supplied to the chaperone created by `chaperone-procedure`, and `new-a` and `new-b` are the replacement arguments that are forwarded to the chaperoned *orig-proc*.

After replacements for the original arguments, an *interpose-proc* can return an additional result, which must be a *post-interpose-proc* function to filter the result of the chaperoned function. A *post-interpose-proc* must accept as many values as the chaperoned function returns, and it returns replacements for the chaperoned function's results. Since *post-interpose-proc* is determined after the arguments are available, the replacement result from *post-interpose-proc* can depend on the original arguments provided to *interpose-proc*.

For example, to chaperone a function of two arguments that produces a single result, and to adjust the result as well as the arguments, an *interpose-proc* would have the form

```
(λ (a b) .... (values new-a new-b
                      ; post-interpose-proc:
```

```
                    (λ (result) .... new-result)))
```

where `result` is the result of the chaperoned *`proc`*, and `new-result` is the result that is delivered to the caller of the chaperone.

When the run-time system applies *`interpose-proc`* for a chaperoned function call, it checks that the replacement arguments from *`interpose-proc`* are the same as or chaperones of the original arguments. Similarly, when the run-time system applies a *`post-interpose-proc`* to the chaperoned call's result, it checks that the replacement result is the same as or a chaperone of the original.

Using chaperones to implement contracts is straightfoward. The contract on a procedure like `sqr`,

```
(provide/contract
 [sqr (real? . -> . nonnegative-real?)])
```

is implemented as a chaperone of `sqr`:

```
(chaperone-procedure
 sqr
 (λ (n)
   (unless (real? n) (blame client "real"))
   (values
    n
    (λ (result)
      (unless (nonnegative-real? result)
        (blame provider "nonnegative real"))
      result))))
```

The `->` contract constructor specifically creates a chaperone to implement a function contract when the argument and result contracts are flat contracts, as are `real?` and `nonnegative-real?`. If the `->` contract constructor encounters an argument contract like $\alpha$, then it must instead create an impersonator.

The `impersonate-procedure` constructor works the same way as `chaperone-procedure`. When an impersonator is applied, the run-time system skips the check on argument and result replacements, since they are not required to be the same as or chaperones of the original arguments and results. Naturally, the result of `impersonate-procedure` is not `chaperone-of?` the original procedure, so it cannot be used in situations that require a chaperone. Every chaperone counts as an impersonator, since a chaperone is a restricted form of impersonator.

### 3.2 Chaperoning and Impersonating Vectors

The `chaperone-vector` function takes a vector and creates a chaperone that appears to be like any other vector: the `vector?` predicate returns true when applied to the chaperone, and `equal?` can be used to compare the chaperone to another vector.

To chaperone a vector, `chaperone-vector` needs the vector to chaperone and two functions: one function that interposes on access of vector elements, and another that interposes on assignments to vector slots:

```
(chaperone-vector vec interpose-ref interpose-set)
```

When `vector-ref` is called on the chaperone with an index *`i`*, *`interpose-ref`* is called with three arguments: *`vec`*, *`i`*, and the result of `(vector-ref vec i)`, which is the result that would be returned by the original vector. The result of *`interpose-ref`* is a replacement for `(vector-ref vec i)`, and so it must be the same as or a chaperone of the original. The protocol for *`interpose-set`* is essentially the same.

For example, a use of `partial-sums!`

```
(provide/contract
 [partial-sums! ((vectorof number?) . -> . any)])
```

can be instantiated using `chaperone-vector` to oblige a client to supply a vector that contains only numbers, and to constrain `partial-sums!` itself to mutate the vector only by installing numbers:

```
(chaperone-procedure
 partial-sums!
 (λ (vec)
   (unless (vector? vec) (blame client "vector"))
   (chaperone-vector
    vec
    ; Check accesses:
    (λ (vec i val)
      (unless (number? val) (blame client "number"))
      val)
    ; Check mutations:
    (λ (vec i val)
      (unless (number? val) (blame provider "number"))
      val))))
```

Note how the *`interpose-ref`* procedure blames *`client`*, while *`interpose-set!`* blames *`provider`*; if the vector were a result of `partial-sums!` instead of an argument, the roles would be reversed. This relative swapping of blame is analogous to the swapping that occurs when functions are used as arguments versus produced as results, and it is supported naturally by the chaperone API.

The `impersonate-vector` function works the same way as `chaperone-vector`, but without chaperone checks on replacement values. Less obviously, `impersonate-vector` requires a *mutable* vector, whereas an *immutable* vector might be used to implement a constant lookup table. If a vector is known to be immutable (via Racket's `immutable?` predicate), then `vector-ref` on a particular slot should always return the same result. Chaperones enforce a suitable notion of "same result," so immutable vectors can be chaperoned; impersonators could break the intent of an immutable vector, so immutable vectors cannot be impersonated.

It may seem that an *`interpose-ref`* needs only an *`i`* argument, since the *`interpose-ref`* provided to `chaperone-vector` could capture *`vec`* in its closure and extract the original value from *`vec`*. Passing *`vec`* to *`interpose-ref`*, however, helps avoid the extra overhead of allocating a closure when creating a vector chaperone. More significantly, a vector chaperone can wrap another chaperone, in which case the `vector-ref` interposition functions compose naturally and with linear complexity when *`vec`*, *`i`*, and *`val`* are all provided. Along similar lines, *`interpose-set`* could install its replacement value directly into *`vec`*, but for better composition it instead returns a value to be installed.

### 3.3 Chaperoning and Impersonating Structures

Racket's `define-struct` form creates a new structure type with a fixed number of fields, and it binds constructor, predicate, accessor, and (optionally) mutator functions for the new structure type. For example,

```
(define-struct fish (color [weight #:mutable]))
```

defines a type with the constructor `make-fish` to create instances, the predicate `fish?` to recognize instances, the accessor `fish-color` to extract the first field of an instance, and the accessor `fish-color` to the extract the second field of an instance. Since the second field is annotated `#:mutable`, `define-struct` also binds `set-fish-weight!` as a mutator to change an instance's second field.

The `chaperone-struct` function creates a chaperone on an instance of a structure type. Whereas the chaperone constructors for functions and vectors take a fixed number of interposition functions, `chaperone-struct` deals with arbitrary structure types that can have different numbers of fields and varying visibility of operations. The `chaperone-struct` function thus takes a structure instance with pairs of operations and interposition procedures. For example, a contract on a `fish` instance `dory`—to ensure that `dory` is blue always between 10 and 12 pounds—could be implemented as

```
(chaperone-struct
 dory
 fish-color (validate-color provider)
 fish-weight (validate-weight provider)
 set-fish-weight! (validate-weight client))
```

where `validate-color` and `validate-weight` perform the actual checks:

```
(define ((validate-color whom) self val)
  (unless (equal? val 'blue)
    (blame whom "blue" val))
  val)
(define ((validate-weight whom) self val)
  (unless (and (real? val)
               (<= 10 val 12))
    (blame whom "10 to 12" val))
  val)
```

In principle, every value in Racket is a structure, and chaperone functions like `chaperone-vector` and `chaperone-procedure` internally use `chaperone-struct` to apply chaperones through interposition of private accessors and mutators.[3] By exposing or hiding structure operations, a library implementer can choose to either allow clients to use `chaperone-struct` directly or force clients to use some other chaperone-creation function that is exported by the library.

## 4. Model

Since contracts are implemented as chaperones and impersonators, contract behavior is constrained by the behavior of chaperones, impersonators, and the rest of the core language. We therefore formally model a language with just chaperones and impersonators to prove properties that we want to hold for a language with contracts. We are particularly interested in a *contract erasure* property that pins down when a program with contracts will behave the same with all contracts removed.

Section 4.1 defines VectorRacket, which is a subset of Racket. VectorRacket does not include structures, but it includes both mutable and immutable vectors, and immutable vectors have similar chaperone constraints as opaque structures. Section 4.2 extends VectorRacket to include chaperones and impersonators for vectors. Section 4.3 presents theorems that illustrate the desired properties of chaperones.

### 4.1 VectorRacket

Figure 2 shows the grammar for VectorRacket. The surface language (the left-hand column) includes λ expressions, application, variables ($x$), let expressions, if expressions, errors, booleans ($b$), natural numbers ($n$), a "void" result for side effects, and primitives. The primitives include operations for creating and inspecting vectors, as well as three predicates: equal? to compare two values structurally, egal? to compare two values structurally up to the addresses of mutable vectors, and immutable? to determine whether a value is an immutable vector.

The evaluator for VectorRacket (figure 3) returns the atomic tag proc or vector to indicate that the result of evaluation was some procedure or some vector, respectively. If the result was some other kind of value, the evaluator returns it directly. If evaluation gets stuck at a non-value, the evaluator returns error. The evaluator is a partial function, since it is undefined when evaluation of a program fails to terminate.

The evaluator uses the reduction relation $\rightarrow$, which is shown in figure 4. The relation uses the additional syntactic categories given

[3] In practice, most procedures and vectors have specialized representations that are exploited, for example, by the just-in-time compiler.

$$
\begin{array}{rl}
e ::= & (\lambda\ (x\ ...)\ e) \mid (e\ e\ ...) \mid x \\
 & \mid (\mathsf{let}\ ([x\ e]\ ...)\ e) \mid (\mathsf{if}\ e\ e\ e) \\
 & \mid (\mathsf{error}\ 'variable) \mid v \\
v ::= & b \mid n \mid (\mathsf{void}) \mid prim \\
b ::= & \#\mathsf{t} \mid \#\mathsf{f} \\
prim ::= & \mathsf{vector} \\
 & \mid \mathsf{vector\text{-}immutable} \\
 & \mid \mathsf{vector\text{-}ref} \mid \mathsf{vector\text{-}set!} \\
 & \mid \mathsf{equal?} \mid \mathsf{egal?} \\
 & \mid \mathsf{immutable?}
\end{array}
\qquad
\begin{array}{rl}
p ::= & (s\ b\ e) \\
s ::= & ((x\ sv)\ ...) \\
sv ::= & (\lambda\ (x\ ...)\ e) \mid (\mathsf{vector}\ b\ v\ ...) \\
 & \mid (\mathsf{vector\text{-}immutable}\ v\ ...) \\
v ::= & .... \mid l \\
l, m, o ::= & (\mathsf{loc}\ x) \\
P ::= & (s\ b\ E) \\
E ::= & [] \mid (v\ ...\ E\ e\ ...) \\
 & \mid (\mathsf{if}\ E\ e\ e) \\
 & \mid (\mathsf{let}\ ([x\ v]\ ...\ [x\ E]\ [x\ e]\ ...)\ e)
\end{array}
$$

Figure 2: VectorRacket Syntax

$$
\mathsf{Eval}[\![e]\!] = \begin{cases}
\mathsf{proc} & \text{if } (()\ \#\mathsf{f}\ e) \rightarrow^* (s\ b\ (loc\ x)) \text{ and} \\
 & \quad s(x) = (\lambda\ (x\ ...)\ e') \\
\mathsf{vector} & \text{if } (()\ \#\mathsf{f}\ e) \rightarrow^* (s\ b\ (loc\ x)) \\
v & \text{if } (()\ \#\mathsf{f}\ e) \rightarrow^* (s\ b\ v) \\
\mathsf{error:}\ msg & \text{if } (()\ \#\mathsf{f}\ e) \rightarrow^* (s\ b\ (\mathsf{error}\ msg)) \\
\mathsf{error} & \text{if } (()\ \#\mathsf{f}\ e) \rightarrow^* p' \text{ and} \\
 & \quad p' \not\rightarrow p'' \text{ for any } p''
\end{cases}
$$

Figure 3: VectorRacket Evaluator (function clauses in order)

on the right-hand column of figure 2. The reduction relation operates on programs ($p$), which consist of three parts: a store ($s$) to map locations to procedures and vectors ($sv$), a boolean to track whether evaluation is in the dynamic extent of a chaperone's interposition function (which aids a formulation of the model's results), and an expression. Expressions during evaluation are nearly the same as surface-level expressions, with the exception that the production (loc $x$) is added to stand for a value in the store. Finally, $P$ and $E$ are evaluation contexts for programs and expressions, respectively.

The rules are mostly standard, with a few exceptions. To support a notion of equality on procedures, procedures are allocated in the store via the [procedure] rule, so the [$\beta$v] rule extracts the procedure from the store before substitution. The rules for if treat non-#f values as if they were true (as in Racket).

The [equal?] rule defers to the equal metafunction (not shown here), which returns #t when the (potentially infinite) unfolding of the first argument is equal to the (potentially infinite) unfolding of the second. The [eqal?] rule is similar, where the egal metafunction is given in figure 7. The egal metafunction also compares the structure of its inputs, but instead of traversing mutable data, it equates vectors only when allocated at same store location. The immutable? predicate detects immutable vectors. The remaining rules handle vector allocation, access, and update, where vector allocation records whether it was allocated by an interposition (i.e., the program state's boolean).

### 4.2 VectorRacket with Chaperones

Figure 5 extends the syntax of VectorRacket to support chaperones and impersonators. The extensions include three new primitives, value forms for chaperones and impersonators, and set-marker and clear-marker forms to record whether evaluation is in one of a chaperone's interposition functions.

The chaperone-vector primitive works as in Racket: its first argument is a vector to be chaperoned, its second argument is a procedure to interpose on vector access, and its third argument is a procedure to interpose on vector update:

$$
\begin{array}{lr}
(s\ b\ E[(\lambda\ (y \ldots)\ e)]) & \text{[procedure]} \\
\longrightarrow (s[x{\mapsto}(\lambda\ (y \ldots)\ e)]\ b\ E[(\mathsf{loc}\ x)]) & \\
\quad \text{where } x \text{ fresh} & \\
(s\ b\ E[((\mathsf{loc}\ x_p)\ v \ldots)]) & [\beta v] \\
\longrightarrow (s\ b\ E[\{x{:=}v, \ldots\}\ e]) & \\
\quad \text{where } (\lambda\ (x \ldots)\ e) = s(x_p),\ |(v \ldots)| = |(x \ldots)| & \\
P[(\mathsf{let}\ ([x\ v] \ldots)\ e)] \longrightarrow P[\{x{:=}v, \ldots\}\ e] & \text{[let]} \\
P[(\mathsf{if}\ v\ e_1\ e_2)] \longrightarrow P[e_1] & \text{[ift]} \\
\quad \text{where } v \neq \#\mathrm{f} & \\
P[(\mathsf{if}\ \#\mathrm{f}\ e_1\ e_2)] \longrightarrow P[e_2] & \text{[iff]} \\
(s\ b\ E[(\mathsf{error}\ '\mathit{variable})]) & \text{[error]} \\
\longrightarrow (s\ b\ (\mathsf{error}\ '\mathit{variable})) & \\
\quad \text{where } [] \neq E & \\
(s\ b\ E[(\mathsf{equal?}\ v_1\ v_2)]) & \text{[equal?]} \\
\longrightarrow (s\ b\ E[\mathsf{equal}[[s, v_1, v_2]]]) & \\
(s\ b\ E[(\mathsf{egal?}\ v_1\ v_2)]) & \text{[egal?]} \\
\longrightarrow (s\ b\ E[\mathsf{egal}[[s, v_1, v_2]]]) & \\
(s\ b\ E[(\mathsf{vector}\ v \ldots)]) & \text{[vector]} \\
\longrightarrow (s[x{\mapsto}(\mathsf{vector}\ b\ v \ldots)]\ b\ E[(\mathsf{loc}\ x)]) & \\
\quad \text{where } x \text{ fresh} & \\
(s\ b\ E[(\mathsf{immutable?}\ v)]) & \text{[immut]} \\
\longrightarrow (s\ b\ E[\mathsf{immutable}[[s, v]]]) & \\
(s\ b\ E[(\mathsf{vector\text{-}immutable}\ v \ldots)]) & \text{[immvec]} \\
\longrightarrow (s[x{\mapsto}(\mathsf{vector\text{-}immutable}\ v \ldots)]\ b\ E[(\mathsf{loc}\ x)]) & \\
\quad \text{where } x \text{ fresh} & \\
(s\ b\ E[(\mathsf{vector\text{-}set!}\ (\mathsf{loc}\ x)\ n\ v_{new})]) & \text{[vector-set!]} \\
\longrightarrow (s[x{\mapsto}(\mathsf{vector}\ b\ v_1 \ldots v_{new}\ v_3 \ldots)] & \\
\quad b\ E[(\mathsf{void})]) & \\
\quad \text{where } (\mathsf{vector}\ b\ v_1 \ldots v_2\ v_3 \ldots) = s(x), & \\
\qquad |(v_1 \ldots)| = n & \\
(s\ b_1\ E[(\mathsf{vector\text{-}ref}\ (\mathsf{loc}\ x)\ n)]) & \text{[mvec-ref]} \\
\longrightarrow (s\ b_1\ E[v_2]) & \\
\quad \text{where } (\mathsf{vector}\ b_2\ v_1 \ldots v_2\ v_3 \ldots) = s(x), & \\
\qquad |(v_1 \ldots)| = n & \\
(s\ b_1\ E[(\mathsf{vector\text{-}ref}\ (\mathsf{loc}\ x)\ n)]) & \text{[immvec-ref]} \\
\longrightarrow (s\ b_1\ E[v_2]) & \\
\quad \text{where } (\mathsf{vector\text{-}immutable}\ v_1 \ldots v_2\ v_3 \ldots) = s(x), & \\
\qquad |(v_1 \ldots)| = n &
\end{array}
$$

Figure 4: VectorRacket Reductions

$$
\begin{array}{rl}
\mathit{prim} ::= & \ldots. \mid \mathsf{chaperone\text{-}vector} \mid \mathsf{chaperone\text{-}of?} \mid \mathsf{impersonate\text{-}vector} \\
\mathit{sv} ::= & \ldots. \mid (\mathsf{chaperone\text{-}vector}\ l\ m\ o) \mid (\mathsf{impersonate\text{-}vector}\ l\ m\ o) \\
e ::= & \ldots. \mid (\mathsf{set\text{-}marker}\ e) \mid (\mathsf{clear\text{-}marker}\ e)
\end{array}
$$

Figure 5: VectorRacket Chaperone Syntax Extensions

$$
\mathsf{Eval}\left[\!\left[ \begin{array}{l} (\mathsf{vector\text{-}ref}\ (\mathsf{chaperone\text{-}vector}\ (\mathsf{vector}\ 1\ 2\ 3) \\ \qquad\qquad\qquad\qquad\qquad\qquad (\lambda\ (\mathsf{vec}\ \mathsf{i}\ \mathsf{ov})\ \mathsf{ov}) \\ \qquad\qquad\qquad\qquad\qquad\qquad (\lambda\ (\mathsf{vec}\ \mathsf{i}\ \mathsf{ov})\ \mathsf{ov})) \\ \qquad\qquad 1) \end{array} \right]\!\right] = 2
$$

If the interposition function attempts to return a completely different value, the program aborts, signalling an error that the chaperone misbehaved:

$$
\mathsf{Eval}\left[\!\left[ \begin{array}{l} (\mathsf{vector\text{-}ref}\ (\mathsf{chaperone\text{-}vector}\ (\mathsf{vector}\ 1\ 2\ 3) \\ \qquad\qquad\qquad\qquad\qquad\qquad (\lambda\ (\mathsf{vec}\ \mathsf{i}\ \mathsf{ov})\ 17) \\ \qquad\qquad\qquad\qquad\qquad\qquad (\lambda\ (\mathsf{vec}\ \mathsf{i}\ \mathsf{ov})\ \mathsf{ov})) \\ \qquad\qquad 1) \end{array} \right]\!\right] = \mathsf{error}\text{: bad-cvref}
$$

The [out-cvec-ref] and [in-cvec-ref] rules of figure 6 handle vector-ref of a chaperone. The two rules are essentially the same, but [out-cvec-ref] applies when evaluation first moves into interposition mode, while [in-cvec-ref] applies when evaluation is already in interposition mode (as indicated by the boolean in the program state). In either case, the rules expand a vector-ref application to extract a value from the chaperoned vector, apply the interposition function, and check that the interposition function's result is a chaperone of the original value. The [out-cvec-ref] rule also uses set-marker and clear-marker to move into and out of interposition mode. The [setm] and [clearm] helper rules directly manipulate the boolean in the program state and then reduce to their arguments.

The [out-cvec-ref] and [in-cvec-ref] rules of figure 6 similarly handle vector-set! on vector chaperones. The [ivec-ref] and [ivec-ref] rules handle vector-ref and vector-set! on impersonators, which require no chaperone-of? checks. The [cvec] and [ivec] rules handle chaperone and impersonator construction.

The chaperone-of? primitive is handled by the [cof] rule, which defers to the chaperone-of metafunction defined in figure 7. The result of chaperone-of is #t for syntactically identical values. If both arguments are immutable vectors of the same length, then the elements are checked point-wise. If the first argument is a location in the store that points at a chaperone, then the metafunction recurs using the chaperoned value. Otherwise, chaperone-of returns #f.

### 4.3 Claims

This section describes a number of facts about our model that should generalize to Racket itself. The full proofs are included in this submission's supplementary material.

We first establish some base definitions and lemmas. A store $s_2$ *extends* a store $s$, or $s \leq s_2$, if $s_2$ differs from $s$ only in new allocations or the values stored in mutable vectors. That is, for each location $x$ in the domain of $s$, $x$ is in the domain of $s_2$ and

- $s(x) = (\mathsf{vector}\ v_1 \ldots)$, $s_2(x) = (\mathsf{vector}\ v_2 \ldots)$, and $|(v_1 \ldots)| = |(v_2 \ldots)|$, or
- $s(x)$ is the same term as $s_2(x)$.

Given this definition of store extension, we show that reducing an expression results in a store extension.

**Lemma 1.** *For all $s$, $b$, and $e$, if $(s\ b\ e)$ reduces to $(s_2\ b_2\ e_2)$ for some $s_2$, $b_2$, and $e_2$, then $s \leq s_2$.*

***Proof sketch*** Induct on the sequence of reduction steps. For each step, either the store is unchanged, a fresh location is allocated, or an existing mutable vector is changed.

Furthermore, we show that if we know a value $v_1$ is a chaperone of a value $v_2$ in a given store $s$, then $v_1$ is still a chaperone of $v_2$ in any extension of $s$. □

**Lemma 2.** *For all $v_1$, $v_2$, $s$, and $s_2$, if $s_2 \leq s$, $s$ contains no immutable cycles, and* chaperone-of$[[s, v_1, v_2]]$, *then* chaperone-of$[[s_2, v_1, v_2]]$.

$$
\begin{array}{l}
(s\ \#\mathrm{f}\ E[(\text{vector-ref}\ (\text{loc}\ x)\ n)]) \qquad [\text{out-cvec-ref}] \\
\longrightarrow (s\ \#\mathrm{f}\ E[(\text{let}\ ([\text{old}\ (\text{vector-ref}\ l\ n)]) \\
\qquad (\text{let}\ ([\text{new}\ (\text{set-marker}\ (m\ l\ n\ \text{old}))]) \\
\qquad\ (\text{clear-marker} \\
\qquad\ (\text{if}\ (\text{chaperone-of?}\ \text{new}\ \text{old}) \\
\qquad\quad \text{new} \\
\qquad\quad (\text{error}\ \text{'bad-cvref}))))])) \\
\quad \text{where}\ (\text{chaperone-vector}\ l\ m\ o) = s(x) \\
(s\ \#\mathrm{t}\ E[(\text{vector-ref}\ (\text{loc}\ x)\ n)]) \qquad [\text{in-cvec-ref}] \\
\longrightarrow (s\ \#\mathrm{t}\ E[(\text{let}\ ([\text{old}\ (\text{vector-ref}\ l\ n)]) \\
\qquad (\text{let}\ ([\text{new}\ (m\ l\ n\ \text{old})]) \\
\qquad\ (\text{if}\ (\text{chaperone-of?}\ \text{new}\ \text{old}) \\
\qquad\quad \text{new} \\
\qquad\quad (\text{error}\ \text{'bad-cvref}))))]) \\
\quad \text{where}\ (\text{chaperone-vector}\ l\ m\ o) = s(x) \\
(s\ b\ E[(\text{set-marker}\ e)]) \longrightarrow (s\ \#\mathrm{t}\ E[e]) \qquad [\text{setm}] \\
(s\ b\ E[(\text{clear-marker}\ e)]) \longrightarrow (s\ \#\mathrm{f}\ E[e]) \qquad [\text{clearm}] \\
(s\ \#\mathrm{f}\ E[(\text{vector-set!}\ (\text{loc}\ x)\ n\ v)]) \qquad [\text{out-cvec-set!}] \\
\longrightarrow (s\ \#\mathrm{f}\ E[(\text{let}\ ([\text{new}\ (\text{set-marker}\ (o\ l\ n\ v))]) \\
\qquad (\text{clear-marker} \\
\qquad\ (\text{if}\ (\text{chaperone-of?}\ \text{new}\ v) \\
\qquad\quad (\text{vector-set!}\ l\ n\ \text{new}) \\
\qquad\quad (\text{error}\ \text{'bad-cvset}))))]) \\
\quad \text{where}\ (\text{chaperone-vector}\ l\ m\ o) = s(x) \\
(s\ \#\mathrm{t}\ E[(\text{vector-set!}\ (\text{loc}\ x)\ n\ v)]) \qquad [\text{in-cvec-set!}] \\
\longrightarrow (s\ \#\mathrm{t}\ E[(\text{let}\ ([\text{new}\ (o\ l\ n\ v)]) \\
\qquad (\text{if}\ (\text{chaperone-of?}\ \text{new}\ v) \\
\qquad\quad (\text{vector-set!}\ l\ n\ \text{new}) \\
\qquad\quad (\text{error}\ \text{'bad-cvset})))]) \\
\quad \text{where}\ (\text{chaperone-vector}\ l\ m\ o) = s(x) \\
(s\ b\ E[(\text{vector-ref}\ (\text{loc}\ x)\ n)]) \qquad [\text{ivec-ref}] \\
\longrightarrow (s\ b\ E[(m\ l\ n\ (\text{vector-ref}\ l\ n))]) \\
\quad \text{where}\ (\text{impersonate-vector}\ l\ m\ o) = s(x) \\
(s\ b\ E[(\text{vector-set!}\ (\text{loc}\ x)\ n\ v)]) \qquad [\text{ivec-set!}] \\
\longrightarrow (s\ b\ E[(\text{vector-set!}\ l\ n\ (o\ l\ n\ v))]) \\
\quad \text{where}\ (\text{impersonate-vector}\ l\ m\ o) = s(x) \\
(s\ b\ E[(\text{chaperone-vector}\ l\ m\ o)]) \qquad [\text{cvec}] \\
\longrightarrow (s[x \mapsto (\text{chaperone-vector}\ l\ m\ o)]\ b\ E[(\text{loc}\ x)]) \\
\quad \text{where}\ \text{isvector}[\![s, l]\!],\ x\ \text{fresh} \\
(s\ b\ E[(\text{impersonate-vector}\ l\ m\ o)]) \qquad [\text{ivec}] \\
\longrightarrow (s[x \mapsto (\text{impersonate-vector}\ l\ m\ o)]\ b\ E[(\text{loc}\ x)]) \\
\quad \text{where}\ \text{ismutable}[\![s, l]\!],\ x\ \text{fresh} \\
(s\ b\ E[(\text{chaperone-of?}\ v_1\ v_2)]) \qquad [\text{cof}] \\
\longrightarrow (s\ b\ E[\text{chaperone-of}[\![s, v_1, v_2]\!]])
\end{array}
$$

Figure 6: VectorRacket Chaperone Reductions

***Proof sketch*** All points inspected in the store to determine that $v_1$ is a chaperone of $v_2$ do not change by the definition of store extension, so the calculation in $s_2$ will proceed the same. □

A similar lemma holds for two values that are egal?:

**Lemma 3.** *For all $v_1$, $v_2$, $s$, and $s_2$, if $s_2 \leq s$, $s$ contains no immutable cycles, and* $\text{egal}[\![s, v_1, v_2]\!]$, *then* $\text{egal}[\![s_2, v_1, v_2]\!]$.

For the theorems and claims below, we also need that equal? and egal? are equivalence relations. We also have proven that no reduction steps starting from an empty store can cause an immutable cycle as detailed above, so including that as a hypothesis is not unreasonable.

The first theorem concerns a guarantee that chaperones provide: equalities are preserved for separate retrievals from an immutable data structure. It encapsulates the idea that the following function in Racket either diverges, results in an error, or returns true:

```
(λ (v1 v2 k)
  (if (and (immutable? v1) (immutable? v2)
           (equal? v1 v2))
      (equal? (vector-ref v1 k)
              (vector-ref v2 k))
      #t))
```

$$
\begin{array}{l}
\text{immutable}[\![s, (\text{loc}\ x)]\!] = \#\mathrm{t} \\
\quad \text{where}\ (\text{vector-immutable}\ v\ ...) = s(x) \\
\text{immutable}[\![s, (\text{loc}\ x)]\!] = \text{immutable}[\![s, l]\!] \\
\quad \text{where}\ (\text{chaperone-vector}\ l\ m\ o) = s(x) \\
\text{immutable}[\![s, v]\!] = \#\mathrm{f} \\
\\
\text{egal}[\![s, v, v]\!] = \#\mathrm{t} \\
\text{egal}[\![s, (\text{loc}\ x), v_2]\!] = \text{egal}[\![s, l, v_2]\!] \\
\quad \text{where}\ (\text{chaperone-vector}\ l\ m\ o) = s(x) \\
\text{egal}[\![s, v_1, (\text{loc}\ y)]\!] = \text{egal}[\![s, v_1, l]\!] \\
\quad \text{where}\ (\text{chaperone-vector}\ l\ m\ o) = s(y) \\
\text{egal}[\![s, (\text{loc}\ x), (\text{loc}\ y)]\!] = \wedge[\![\text{egal}[\![s, v_1, v_2]\!], ...]\!] \\
\quad \text{where}\ (\text{vector-immutable}\ v_1\ ...) = s(x), \\
\qquad (\text{vector-immutable}\ v_2\ ...) = s(y), \\
\qquad |(v_1\ ...)| = |(v_2\ ...)| \\
\text{egal}[\![s, v_1, v_2]\!] = \#\mathrm{f} \\
\\
\text{chaperone-of}[\![s, v, v]\!] = \#\mathrm{t} \\
\text{chaperone-of}[\![s, (\text{loc}\ x), v_2]\!] = \text{chaperone-of}[\![s, l, v_2]\!] \\
\quad \text{where}\ (\text{chaperone-vector}\ l\ m\ o) = s(x) \\
\text{chaperone-of}[\![s, (\text{loc}\ x), (\text{loc}\ y)]\!] = \wedge[\![\text{chaperone-of}[\![s, v_1, v_2]\!], ...]\!] \\
\quad \text{where}\ (\text{vector-immutable}\ v_1\ ...) = s(x), \\
\qquad (\text{vector-immutable}\ v_2\ ...) = s(y), |(v_1\ ...)| = |(v_2\ ...)| \\
\text{chaperone-of}[\![s, v_1, v_2]\!] = \#\mathrm{f}
\end{array}
$$

Figure 7: VectorRacket Chaperone Metafunctions

**Theorem 1.** *For all $v_1$, $v_2$, and $s$, if*

- *$s$ contains no immutable cycles,*
- $\text{immutable}[\![s, v_1]\!]$,
- $\text{immutable}[\![s, v_2]\!]$,
- *and* $\text{egal}[\![s, v_1, v_2]\!]$,

*then for all* k *and b, either*

- *there exists no $v_3$ and $s_2$ such that* $(s\ b\ (\text{vector-ref}\ v_1\ \mathsf{k}))$ *reduces to* $(s_2\ b\ v_3)$, *or*
- *there exists a $v_3$ and $s_2$ such that* $(s\ b\ (\text{vector-ref}\ v_1\ \mathsf{k}))$ *reduces to* $(s_2\ b\ v_3)$ *and for all $b_2$ and $s_3 \leq s_2$, either*
  - *there exists no $v_4$ and $s_4$ such that* $(s_3\ b_2\ (\text{vector-ref}\ v_2\ \mathsf{k}))$ *reduces to* $(s_4\ b_2\ v_4)$, *or*
  - *there exists a $v_4$ and $s_4$ such that* $(s_3\ b_2\ (\text{vector-ref}\ v_2\ \mathsf{k}))$ *reduces to* $(s_4\ b_2\ v_4)$, *and for all $s_5 \leq s_4$,* $\text{egal}[\![s_5, v_3, v_4]\!]$.

***Proof sketch*** Induct on the triple ($v_1$, $v_2$, s). If either $v_1$ or $v_2$ is not a location that points to an immutable vector or chaperone, then the corresponding immutable hypothesis is false.

If both $v_1$ and $v_2$ are locations that point to immutable vectors, then there are two cases: either k is out of bounds for both, in which case we do not find values $v_3$ and $v_4$, or k is in bounds, in which case the values $v_3$ and $v_4$ must be egal, since $v_1$ and $v_2$ are egal.

If $s(v_1) = (\text{chaperone-vector}\ v_c\ v_{f1}\ v_{f2})$, then we have an inductive hypothesis relating $v_c$ and $v_2$. If $(s\ b\ (\text{vector-ref}\ v_c\ \mathsf{k}))$ fails to reduce, then we cannot find a $v_3$ and $s_2$ for the reduction of $(s\ b\ (\text{vector-ref}\ v_1\ \mathsf{k}))$. If $(s\ b\ (\text{vector-ref}\ v_c\ \mathsf{k}))$ reduces to $(s_c\ b\ v_{r1})$, then either $(s_c\ b\ (v_{f1}\ v_{r1}))$ fails to reduce, in which case $(s\ b\ (\text{vector-ref}\ v_1\ \mathsf{k}))$ has failed to reduce, or it results in $(s_f\ b\ v_{r2})$. If $v_{r2}$ is not a chaperone of $v_{r1}$, then $(s\ b\ (\text{vector-ref}\ v_1\ \mathsf{k}))$ fails to reduce, and if it is a chaperone, then we have that $s_2 = s_f$ and $v_3 = v_{r2}$. We can then use the induction hypothesis, which relates $v_{r1}$ and $v_4$, the knowledge that $v_3$ is a chaperone of $v_{r1}$, and our lemmas about egal and chaperone-of

in store extensions to get that $v_3$ and $v_4$ are egal in the final store. The chaperone case for $v_2$ follows similarly. □

There are also equivalences that hold through the use of chaperones, even in the presence of arbitrary code. Take the following function:

```
(λ (v1 f k)
  (let ([n (vector-ref v1 k)] [v2 (f v1)])
    (if (and (immutable? v1) (immutable? v2)
             (equal? v1 v2))
        (= (vector-ref v2 k) n)
        #t)))
```

Even though `f` is an arbitrary function that may have side effects, this function should still return `#t` modulo errors or divergence. From this function, we formulate the following more general theorem:

**Theorem 2.** *For all $v_1$ and $s$, if*

- *$s$ contains no immutable cycles,*
- immutable[[$v_1$]], *and*
- *there exists a* k, *$b_r$, $n$, and $s_r$ such that* ($s$ $b_r$ (vector-ref $v_1$ k)) *reduces to* ($s_r$ $b_r$ $n$),

*then for all $s_2 \leq s_r$ and $v_2$, if*

- immutable[[$v_2$]] *and*
- (equal $s_2$ $v_1$ $v_2$),

*then for all $b$, either*

- *there exists no $v$ and $s_3$ such that* ($s_2$ $b$ (vector-ref $v_2$ k)) *reduces to* ($s_3$ $b$ $v$), *or*
- *there exists a $s_3$ such that* ($s_2$ $b$ (vector-ref $v_2$ k)) *reduces to* ($s_3$ $b$ $n$).

***Proof sketch*** First we induct on $v_1$. Like before, unless $v_1$ is a pointer to an immutable vector or a chaperone, then the immutable hypothesis fails. If $s(v_1)$ = (chaperone $v_c$ $v_{f1}$ $v_{f2}$), then if we can show that ($s$ $b$ (vector-ref $v_c$ k)) reduces to ($s_c$ $b$ $n$) for some store $s_2 \leq s_r \leq s_c$, then we get an inductive hypothesis that gives us the consequent we need for $v_2$'s behavior. Since we also have a hypothesis that ($s$ $b$ (vector-ref $v_1$ k)) reduces to ($s_r$ $b$ $n$), we can pull apart that reduction sequence to find the appropriate $s_c$ as a witness.

If $v_1$ is a pointer to an immutable vector, then we fix $v_2$ and $s_2$ and induct on $v_2$. If $v_2$ is not a pointer to an immutable vector or a chaperone, then the immutable hypothesis fails. If $v_2$ is an immutable vector, then from the definition of equal, we get that its kth element must be $n$ as well. If $s_2(v_2)$ = (chaperone-vector $v_c$ $v_{f1}$ $v_{f2}$), then we get an inductive hypothesis about (vector-ref $v_c$ k). If it doesn't reduce in the store $s_2$, then (vector-ref $v_c$ k) doesn't either. If it does, giving us a new store $s_c$, then either applying $s_{f1}$ to $n$ in $s_c$ fails to reduce; we get a value that is not $n$, in which case the chaperone-of? check fails; or we get a value $n$ in a new store which is our store $s_3$. □

Furthermore, we define the idea of *contract erasure* for a subset of programs. If a well-behaved program with chaperones evaluates to a value, then the program with all chaperones removed will evaluate to an equivalent value. In our model, a well-behaved program is a program whose chaperone wrappers do not affect mutable vectors used by the "main" program, that is, the program with chaperones erased. There are two ways that chaperones might do this: through mutating vectors allocated by the main program, or providing the main program with vectors allocated by the chaperone, which can later be used as a channel of communication.

Since chaperone wrappers must return values that are chaperones of the appropriate argument, and chaperones must share the same mutable state, providing the main program with chaperone-allocated vectors is only possible by placing that vector in a vector allocated by the main program. Thus, we need only detect the mutation of main program state within a chaperone wrapper to detect ill-behaved programs. We do this by looking for reductions where the left hand side is marked as being under the dynamic extent of a chaperone wrapper and the redex is a vector-set! on a vector allocated outside of any chaperone wrappers.

In order to prove contract erasure, we prove the following lemma, which states that if a program state without any chaperones reduces a redex to a value, then a similar program which differs only in adding chaperones also reduces to an appropriately similar value. We also disallow set-marker and get-marker, since these are not meant to be written by the user in programs.

**Lemma 4.** *For all $e_2 = E_2[e_6]$ that do not contain uses of set-marker, get-marker, or chaperone-vector, if there exists an $s_2$, $v_2$, and $s_4$ such that* ($s_2$ #f $E_2[e_6]$) *reduces to* ($s_4$ #f $E_2[v_2]$), *then for all $e_1 = E_1[e_5]$ and $s_1$ such that $\langle e_1, s_1\rangle \sim \langle e_2, s_2\rangle$, where the reduction of* ($s_1$ #f $e_1$) *contains no program states of the form* ($s$ t $E$[(vector-set! (loc $x$) $n$ $v$)]) *where* $s(x)$ = (vector #f $v_e$), *either:*

- ($s_1$ #f $e_1$) *diverges,*
- *there exists a $b$ and $s_3$ such that* ($s_1$ #f $e_1$) *reduces to* ($s_3$ $b$ (error 'var)),
- *there exists an $e_3$, $b$, and $s_3$ such that* ($s_1$ #f $e_1$) *reduces to* ($e_3$ $b$ $s_3$) *and $e_3$ is a stuck state,*
- *there exists a $v_1$ and $s_3$ such that* ($s_1$ #f $e_1$) *reduces to* ($s_3$ #f $E_1[v_1]$) *and $\langle E_1[v_1], s_3\rangle \sim \langle E_2[v_2], s_4\rangle$.*

**Theorem 3.** *For all $e$, if* Eval$[\![e]\!]$ = $v$ *and that evaluation involves no reductions involving a state* ($s_b$ #t $E$[(vector-set! (loc $x$) $v_b$ $n$)]) *where* $s_b(x)$ = (vector #f $v_v$ ...), *then* Eval$[\![e_2]\!]$ = $v$, *where $e_2$ is the same as $e$ but where* chaperone-vector *is replaced with* (λ ($v$ $x$ $y$) $v$).

***Proof sketch*** Since Eval$[\![e]\!]$ = $v$, we know that (() #f $e$) reduces to ($s_1$ #f $v_1$) for some $s_1$ and $v_1$. Since every reduction step in the erased program has a corresponding reduction step in the unerased program, (() #f $e_2$) reduces to ($s_2$ #f $v_2$) for some $s_2$ and $v_2$. Using lemma 4, we can show that $\langle v_1, s_1\rangle \sim \langle v_2, s_2\rangle$, and by cases on the possible values $v_1$ and $v_2$, either they are both the same boolean, the same number, both pointers to λ terms in the store, or both pointers to non-λ terms in the store, which means that evaluating the erased program gives the same kind of result as evaluating the unerased program. □

## 5. Performance

Although our motivation for adding chaperones and impersonators to Racket is to support new forms of contracts, performance is also a concern. Our primary concern is that support for chaperones, impersonators, and contracts is "pay as you go"; that is, programs that do not use the features should not pay for them. A secondary concern is the performance of chaperones, impersonators, and contracts themselves, and we expect that building a little support for chaperones into the compiler will improve the overall performance of contracts in the long run.

### 5.1 Performance in Untyped Racket

The Racket implementation uses a just-in-time (JIT) compiler to convert bytecode into machine code for each function when the function is first called. When the JIT compiler encounters certain primitive operations, such as `vector-ref`, it generates inline code to implement the operation's common case. The common case corresponds to a non-chaperone, non-impersonator object. For example, the inlined `vector-ref` code checks whether its first argument has the vector type tag, checks whether its second argument is a fixnum, checks whether the fixnum is in range for the vector, and finally extracts the fixnum-indexed element from the vector; if any of the checks fail, the generated machine code bails out to a slower path, which is responsible for handling chaperones as well

as raising exceptions for bad arguments. The addition of chaperones thus has no effect on the machine code generated by the JIT compiler or its resulting performance when chaperones are not used in dynamically typed Racket code.

Racket's object-oriented class system is implemented as a library, and prior to support for chaperones, the class system implemented object-specific wrappers that operated in a similar manner to chaperones. All object operations required a check on the target object to determine whether it was a wrapped object, and since this test was outside the core run-time system, the JIT compiler was not able to recognize the chaperone pattern and optimize for the common case. In fact, the check interfered with optimizations that the JIT compiler could otherwise perform, and the result was a 3x slowdown on field-intensive microbenchmarks that did not use contracts (Strickland and Felleisen 2010). After switching the implementation to use chaperones, this slowdown was completely eliminated.

As of version 5.1.2.3, Racket's implementation is partly optimized for interposition via chaperones. While a bubble sort is a poor choice for sorting, it is a fine micro-benchmark for vector operations. A bubble sort on a vector of 10,000 fixnums in reverse order, using a plain vector versus a no-op chaperone produces the following times on a 2.53 GHz MacBook Pro, where the "old JIT" was not optimized for chaperones:

| | no JIT | old JIT | JIT |
|---|---|---|---|
| vector | 77,938 msec | 1,598 msec | 1,598 msec |
| chaperoned | 120,047 msec | 66,570 msec | 6,577 msec |

Although chaperone-based contract checking clearly incurs a significant cost for a microbenchmark, the timings above illustrate how support for chaperones and impersonators in the run-time system provides a path to better performance.

### 5.2 Performance in Typed Racket

In contrast to untyped Racket code, chaperones create some overhead for typed code in our current implementation. Typed Racket code compiles to Racket code that uses "unsafe" operations, such as `unsafe-vector-ref`. The Typed Racket compiler ensures that these operations are safe in context due to type safety; it also adds contract checks at the boundary with untyped code, and it explicitly inserts bounds checks. The JIT compiler inlines `unsafe-vector-ref`; prior to support for chaperones, the JIT-generated code could completely skip the check type-tag check on the vector argument. Support for chaperones, however, requires `unsafe-vector-ref` to check its first argument for a chaperone tag and dispatch to the slow path. An additional `unsafe-vector*-ref` operation is available to skip the chaperone check, in case a vector is known never to be a chaperone, but Typed Racket does not use the more specialized operation.

For a bubble sort on a vector of 10,000 fixnums, using unsafe operations for the fixnum comparisons and loop controls, different vector operations and comparable C and Java programs yield the following times on the same 2.53 GHz MacBook Pro:

| | |
|---|---|
| `vector-ref` | 1,444 msec |
| `unsafe-vector-ref` | 1,297 msec |
| `unsafe-vector*-ref` | 1,137 msec |
| `gcc -O0` | 727 msec |
| `java (HotSpot)` | 266 msec |
| `gcc -O2` | 107 msec |

As the numbers illustrate, iteration over numerical vectors is hardly Racket's forte. Indeed, the overhead of checking for chaperones (the difference between `unsafe-vector-ref` and `unsafe-vector*-ref`) in Racket is the same order of magnitude as the total run time for optimized C code. The overhead is high because the current JIT code generates a tag check for every individual access to the vector, even though the same vector is used every time. The route to lowering this overhead is to improve the JIT by building more chaperone support into the core run-time system.

## 6. Related Work

Related work falls into two main categories: other implementations of proxies and other implementations of contracts.

### 6.1 Other Proxy Implementations

The most closely related work to ours is the proxy design (Van Cutsem and Miller 2010) proposed for JavaScript and currently implemented in Firefox (Gal 2010). Building on mirages in AmbientTalk (Dedecker et al. 2005), proxies allow unrestricted intercession of almost any operation performed on JavaScript objects. Like our design, theirs does not support intercession on some operations, including `instanceof` tests, `typeof` tests, and the pointer equality operator `===`. Since JavaScript operations such as vector indexing are represented as message sends, only one proxy API is needed, in contrast to our separate APIs separate Racket values.

The primary distinction between JavaScript proxies and our design is how each avoids breaking existing language invariants. JavaScript provides very few invariants that programmers may assume about the behavior of objects, due to pervasive mutability of both objects and prototypes—even allowing so-called "monkey-patching" where the behavior of *all* objects is affected by a single mutation. Further, there is no analogue in JavaScript of the type tests provided by struct predicates (see section 2.3 for the import of these in Racket) and thus JavaScript programmers do not conditionalize code on such tests. Finally, JavaScript provides no reliable structural equality comparison. Since these invariants do not hold for JavaScript programs, proxies need not respect them, simplifying their design considerably.

In contrast, the existing design of Racket, as in most languages, ensures that programmers can reason using a wide variety of invariants based on information hiding, type and equality testing, and immutable objects. Programmers rely on these invariants to build applications, and compilers and static checkers rely on them to reason about programs. Therefore, our design of an intercession API is constrained to respect them.

The current JavaScript language does provide reflective operations which can prevent future mutations to a single field, or to an entire object. While the interaction of proxies with these operations is still being designed, the current proposal[4] imposes chaperone-like invariants on these fields; in particular, the underlying proxy mechanism remembers the first value produced for such fields and skips the intercession on subsequent accesses. Adapting the chaperone invariant for this use case would make JavaScript proxies more expressive.

The performance results for Firefox's JavaScript proxies are encouraging. The implementation imposes little overhead to support proxies (our main goal), and it produces better relative results when proxies are actually used (our secondary goal).

Austin et al. (2011) present an extension of the JavaScript proxy design to primitive values such as integers. They use this proxy system to design a simple contract system (without blame) for a core JavaScript calculus, including mutable data.

Many other tools that allow unrestricted forms of proxying that help to implement contracts but sacrifice the kind of control over invariants that contracts are intended to promote. Notable examples

[4] `http://bit.ly/jsfixedproxies`

include MOP (Kiczales et al. 1991), AOP (Kiczales et al. 1997), and `java.lang.reflect.Proxy` (Oracle 2000).

More recently, there has been a line of work on limiting the power of aspects to ensure various good properties of the resulting aspects (Clifton and Leavens 2002; Dantas and Walker 2006; Rinard et al. 2004). This work generally relies on static type systems, static analyses, or additional formal specifications (and machine checking) and thus gives different kinds of guarantees than chaperones do. Specifically, a chaperone does not restrict the side-effects that might happen during interposition, only the final result. Nevertheless the work does share the goal of preserving the reasoning power of the programmer.

Although Balzer et al. (2005) point out that aspects alone do not provide all of the tools necessary to implement contracts, aspects would allow us to implement the core functionality of contract checking and, in Racket, macros make up the difference.

### 6.2 Other Contract Implementations

Eiffel (Meyer 1991), the original embodiment of Design by Contract, supports contracts directly in the language run-time system. Contracts in Eiffel are limited so that they can be compiled directly into pre- and post-condition checks on methods; for example, higher-order contracts on individual objects are not supported. Eiffel is also less extensible as a programming language. Other notable examples in the Eiffel category include Euclid (Lampson et al. 1977), Ada (via Anna (Luckham and Henke 1985)), D (Digital Mars 1999), and others have built contract extensions for existing languages including for Java, Python, Perl, and Ruby and Ciao (Mera et al. 2009). In all of these cases, contracts are more easily implemented in the core or through pre-preprocessing since the contracts are more limited, the language is typically less extensible, and the contract system is always less extensible.

Two libraries for Haskell (Chitil and Huch 2006; Chitil et al. 2003; Hinze et al. 2006) support contract and assertion checkers that include both specifications for higher-order functions and combinators for building the specifications. Their implementations could benefit from support for impersonators and chaperones.

Finally, Findler, Guo, and Rogers's earlier work on lazy contracts (Findler et al. 2007) helped us understand how impersonators and chaperones should work, although their work does not handle contracts on mutable data structures.

## 7. Conclusion

A powerful contract system provides a way for programmers to express and then rely on complex invariants about program behaviors. Implementing a full-featured contract system, however, requires the ability to interpose on the programming language's primitive operations. The initial implementation of contracts in Racket supported higher-order contracts for functions by using $\lambda$ as its own interposition operator. As Racket's contract system has grown, however, it must interpose on more operations to properly check contracts.

Ironically, general-purpose interposition on primitive operations makes reasoning about program behavior too difficult, because it forces programmers to cope with the possibility of invoking unknown and potentially untrusted code, even when using seemingly simple operations like vector lookup or field selection. Worse, Racket programmers routinely exploit generativity to ensure that complex invariants on data structures hold; with unrestricted forms of interposition, a simple tag check cannot reliably ensure these invariants.

To address these two problems, we have designed chaperones and impersonators as a controlled form of interposition. The design enables the implementation of a rich contract system without giving away the programmer's ability to reason about the behavior of programs.

## Bibliography


Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual Values for Language Extension. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, 2011.

Stephanie Balzer, Patrick Eugster, and Bertrand Meyer. Can Aspects Implement Contracts? In *Proc. Rapid Implemetation of Software Engineering Techniques*, pp. 145–157, 2005.

Olaf Chitil and Frank Huch. A pattern logic for prompt lazy assertions. In *Proc. Intl. Sym. Functional and Logic Programming*, pp. 126–144, 2006.

Olaf Chitil, Dan McNeill, and Colin Runciman. Lazy Assertions. In *Proc. Intl. Sym. Functional and Logic Programming*, pp. 1–19, 2003.

Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Proc. Foundations of Aspect-Oriented Languages*, 2002.

Daniel S. Dantas and David Walker. Harmless Advice. In *Proc. ACM Sym. Principles of Programming Languages*, 2006.

Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-Oriented Programming. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 31–40, 2005.

Digital Mars. D Programming Language. 1999. `http://www.digitalmars.com/d/`

Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 48–59, 2002.

Robert Bruce Findler, Shu-yu Guo, and Anne Rogers. Lazy Contract Checking for Immutable Data Structures. In *Proc. Implementation and Application of Functional Languages*, pp. 111–128, 2007.

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

Andreas Gal. Proxies in Tracemonkey. 2010. `http://hg.mozilla.org/tracemonkey/`

Ralf Hinze, Johan Jeuring, and Andres Löh. Typed Contracts for Functional Programming. In *Proc. Sym. Functional and Logic Programming*, pp. 208–225, 2006.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. European Conf. Object-Oriented Programming*, pp. 220–242, 1997.

Gregor J. Kiczales, James des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. *ACM SIGPLAN Notices* 12(2), pp. 1–79, 1977.

D. C. Luckham and F. W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software* 2(2), pp. 9–22, 1985.

Jacob Matthews and Amal Ahmed. Parametric Polymorphism Through Run-Time Sealing, or, Theorems for Low, Low Prices! In *Proc. European Sym. on Programming*, 2008.

E. Mera, P. Lopez-Garcia, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *Proc. Intl. Conf. on Logic Programming*, LNCS 5649, 2009.

Bertrand Meyer. *Eiffel : The Language*. Prentice Hall PTR, 1991.

Oracle. java.lang.reflect.Proxy. 2000. `http://download.oracle.com/javase/6/docs/api/java/lang/reflect/Proxy.html`

Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A Classification System and Analysis for Aspect-Oriented Programs. In *Proc. Intl. Sym. on the Foundations of Software Engineering*, 2004.

T. Stephen Strickland and Matthias Felleisen. Contracts for First-Class Classes. In *Proc. Dynamic Languages Symposium*, pp. 97–112, 2010.

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 395–406, 2008.

Tom Van Cutsem and Mark Miller. Proxies: Design Principles for Robust Object-oriented Intercession APIs. In *Proc. Dynamic Languages Symposium*, pp. 59–72, 2010.