Chaperones and Impersonators: Run-time Support for Reasonable Interposition

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

Chaperones and *impersonators* provide run-time support for interposing on primitive operations such as function calls, array access and update, and structure field access and update. Unlike most interposition support, chaperones and impersonators are restricted so that they constrain the behavior of the interposing code to *reasonable interposition*, which in practice preserves the abstraction mechanisms and reasoning that programmers and compiler analyses rely on.

Chaperones and impersonators are particularly useful for implementing contracts, and our implementation in Racket allows us to improve both the expressiveness and the performance of Racket's contract system. Specifically, 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, gradual typing systems, such as Typed Racket, that rely on contracts for interoperation with untyped code can now pass mutable values safely between typed and untyped modules.

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

Keywords Proxies, interposition, intercession, contracts

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

1. Extensibility versus Reasoning

An extensible programming language like Racket (Flatt and PLT 2010) enables the authors of libraries 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, the desire for more extensibility comes at the cost of additional behavior that language primitives may exhibit, making it harder for programmers to reason about their programs and for the implementors of the class, component, or type system to deliver on the promises that such abstractions typically make.

The Racket 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 enables language features to be implemented in a library where the features depend on run-time *interposition*—or *intercession*, in the terminology of the CLOS Metaobject Protocol (Kiczales et al. 1991)—to change the behavior of core constructs. Contract checking is our most prominent example, where interposition is needed to 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: the vector may be modified concurrently or through a callback.

If interposition can change the behavior of core constructs, however, then it entails the acute possibility of subverting core guarantees of the programming language, especially those concerning the composition of components. To balance the needs of extensibility and composition, we have developed a two-layer system of interposition: *chaperones* and *impersonators*. Chaperones and impersonators are both proxies, where a wrapper object interposes on operations in-

tended 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 or return the same value as the original object, possibly wrapped with a chaperone. Impersonators, in contrast, are relatively free to change the behavior of the objects that they wrap but they are not allowed on immutable values. Overall, impersonators are more expressive than chaperones, 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 generative structs. 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), whose interoperability with untyped Racket was improved as a result of the addition of proxies. Furthermore, impersonators can be used to implement traditional proxy patterns, such as transparent access of remotely stored fields using standard interfaces, while chaperones can be used to implement constructs such as revokable membranes (Miller 2006).

Last but not least, building interposition support into the core compiler and run-time system offers the promise of better performance, both for code that uses libraries such as contracts and code that does not. For example, the addition of contract support for Racket's class system introduced a factor of three slowdown for some object-oriented operations, even in programs that did not use contracts at all; the support for interposition we present here has eliminated this overhead. Our current implementation of chaperones and impersonators compares favorably to current implementations of Javascript proxies (Van Cutsem and Miller 2010), even though chaperones require additional run-time checks to enforce their constraints.

In short, chaperones and impersonators are a new point in the design space for interposition that enable important, higher-level operations (e.g., higher-order contracts on mutable containers) to be implemented efficiently without subverting any of the existing guarantees of a language that already has both threads and state.

Section 2 uses contract checking in Racket to explore issues of expressiveness and invariants related to interposition. Section 3 describes Racket's chaperone and impersonator API and relates it to the implementation of contracts. Section 4 reports on additional uses of interposition in Racket: remote objects and membranes. Section 5 presents a formal model for a subset of Racket with chaperones and impersonators. Section 6 reports performance numbers. Section 7 discusses related work.

2. Interposition via Contracts

Contract checking is easily the most prominent use of interposition in Racket, and a look at contract checking by itself exposes many of the expressiveness and invariant-preservation concerns that affect a more general interposition mechanism. We therefore start with a careful exploration of Racket contracts as a way of motivating the design of chaperones and impersonators. Readers familiar with the issues surrounding the interaction between contracts, mutable container types, and parametricity may wish to skip to section 2.5 to understand what chaperones and impersonators enable in the contract system, and readers not interested in the motivation behind the design but instead wanting to get right to the details of programming with chaperones and impersonators may wish to skip to section 3.

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 match the boundary between the `math.rkt` and `circles.rkt` modules declared as

`math.rkt`

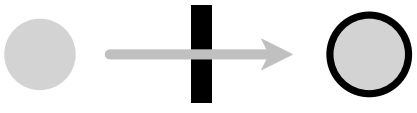
```
(define pi (* (acos 0) 2))
(provide/contract
 [pi real?])
```

`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 to `circles.rkt`.

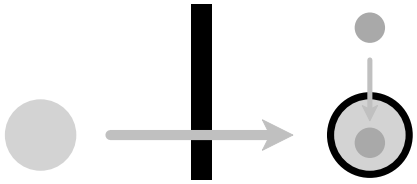
`math.rkt`

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

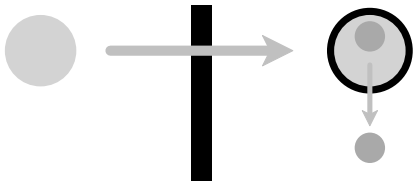
`circles.rkt`

```
(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 inside `circles.rkt`, the argument crosses the wrapper boundary and is checked to ensure that it is a real number:



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

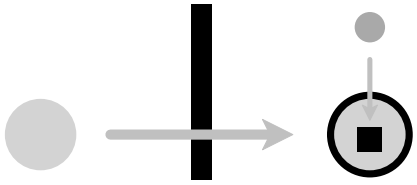


Other kinds of wrappers can implement contracts that guarantee a kind of parametricity for functions. Using `new-∀/c`, for example, the left-hand `poly.rkt` module can promise that its `id` function returns only values that are provided to the function:

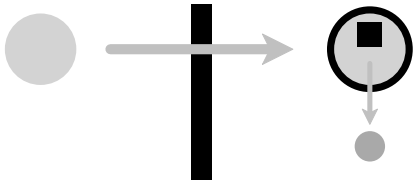
```
poly.rkt
(define (id x) x)
(provide/contract
 [id (let ([α (new-∀/c)])
       (α . -> . α))])
```

```
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 λ and opaque wrappers via Racket's `struct` form. Run-time support becomes necessary, however, to generalize contracts beyond immediate predicates and function wrappers.

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

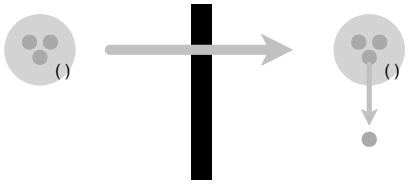
2.2 Compound-Data Contracts

Lists are 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:

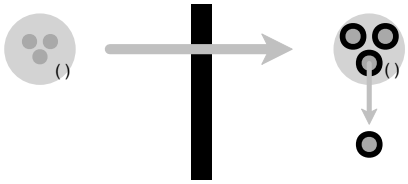


In the case of a list of functions, the list shape of the value can be checked immediately, but the functions themselves may require wrappers. After such a list crosses the contract boundary, the right-hand module sees a list of wrapped functions, and the wrappers remain intact when functions are extracted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



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 for checking the contract. Wrapping the elements of a mutable vector (array), however, does not work:

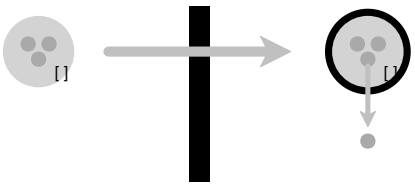
math.rkt

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

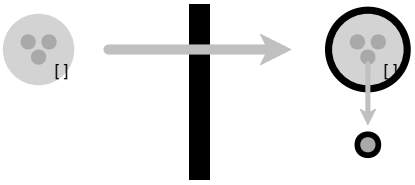
circles.rkt

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

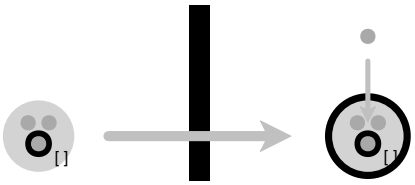
Since the `state` vector is mutable, the intent may be that the left-hand `math.rkt` module can change the values in `state` at any time, with such changes visible to the right-hand module. Consequently, values must be checked at the last minute, when they are extracted from the vector in the right-hand 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 opacity is crucial in the Racket ecosystem. Not only must ordinary user libraries have their internal invariants protected, but systems libraries themselves assume structure opacity because seemingly core forms, such as λ or `class`, are implemented as macros that use structures.

Racket's `struct` form creates a new structure type:

```
(struct widget (parent label callback))
```

This declaration binds `widget` to a constructor that takes three arguments to create a widget instance, and it binds `widget?` to a predicate that produces true for values produced by `widget` and false for any other Racket value. The declaration also binds `widget-parent` to a selector procedure that extracts the `parent` field from a widget, and so on.

The following `widget.rkt` module declares the `widget` structure type, but it also uses the `#:guard` option to add a contract-like guard on the `widget` constructor. It demands that the first constructor argument must be either `#f` or itself a widget, otherwise the construction is rejected.[2]

widget.rkt

```
(struct widget (parent label [callback #:mutable])
  #:guard (λ (p l c info)
            (if (or (false? p) (widget? p))
                (values p l c)
                (error "bad widget"))))

(define (widget-root w)
  (let ([p (widget-parent w)])
    (if p (widget-root p) w)))

(provide widget)
(provide/contract
 [widget-root (widget? . -> . widget?)])
```

The guard on `widget` enforces the invariant that a `widget`'s parent is either `#f` or itself a widget. Consequently, the implementation of `widget-root` can safely recur on a non-`#f` parent without double-checking that the parent is itself a `widget`, because the `widget` constructor guarantees this property.

While the guard on `widget` enforces an invariant for all widgets, a *structure contract* written with `struct/c` can constrain a specific widget instance. For example, the contract

```
(struct/c widget widget? any/c any/c)
```

describes a `widget` instance whose first field is a widget (i.e., it cannot be `#f`), while no new promises are offered the second and third fields. Along the same lines, the following `scene.rkt` module below promises that `plot` is an instance

[2] The `info` argument to the guard procedure contains information indicating if the guard is being invoked on a `widget` instance or a sub-`struct` of `widget`.

of the `widget` structure whose first field is an OpenGL window.

scene.rkt

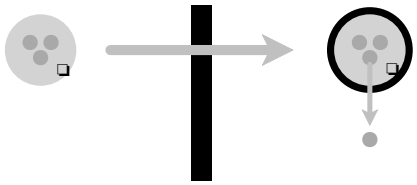
```
(require "widget.rkt")
(define plot (widget ....))
(provide/contract
 [plot (struct/c widget
                 gl-widget?
                 any/c any/c)])
```

circles.rkt

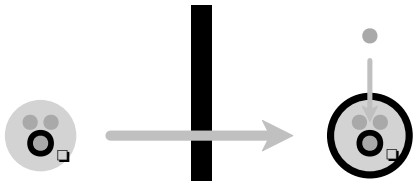
```
(require "widget.rkt"
         "scene.rkt")

(widget-parent plot)
```

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



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.

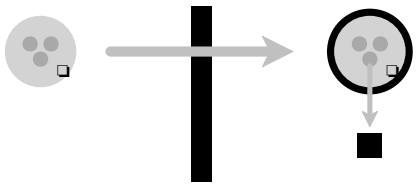
2.4 Parametric Contracts and Generativity Don't Mix

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

scene.rkt

```
(require "widget.rkt")
(define plot (widget ....))
(provide/contract
 [plot (struct/c widget (new-∀/c) 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. If `circles.rkt` applies `widget-root` to `plot`, then `widget-root` fails with an internal error: it gets a value for `plot`'s parent that is neither `#f` or a `widget`, despite the `widget #:guard` and the `widget?` argument contract.

Accordingly, allowing `(struct/c widget (new-∀/c) ...)` would be a mistake. Furthermore, the problem is not confined to an immediate use of `new-∀/c` in `struct/c`. Just as `scene.rkt` must not claim that the `parent` value of `plot` is parametric, it must not claim that the value in the `callback` field of `plot` is a procedure that consumes or produces a parametric value. After all, the guard on `widget` may have wrapped the procedure to ensure properties of the function.

The general principle is that `struct` is *generative*: each time a `struct` definition is evaluated, it creates a unique structure type. If access to a `struct`-generated constructor is limited (e.g., to a particular module), then properties of field values can be ensured by construction. This approach to information hiding breaks down if the contract system allows parametric contracts on immutable structures.

The contract system must therefore distinguish contracts that can break invariants and must be disallowed in certain contexts from contracts that do not break invariants and are safe in all contexts. Instead of imposing constraints specific to combinations of `new-∀/c`, `struct/c`, and `->`, we seek a more general categorization of contracts and contract composition.

2.5 The Contract Hierarchy

Racket's original contract system (Findler and Felleisen 2002) distinguishes two classes of contracts:

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

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

- *Chaperone contracts* can perform immediate checks and add wrappers to delay checks, but immediate or delayed checks can only raise errors. That is, the wrapped values must behave the same after crossing a contract boundary as before, up to contract failures.
- *Impersonator contracts* may replace a value at a contract-boundary crossing with a different or completely opaque value (as with parametric contracts).

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. Only chaperone contracts can be placed on immutable fields in structures like `widget`, be-

cause more general contracts could produce a different result for different uses of the value, making it appear mutable.

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

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

The tildes indicate points where flat contracts were allowed for mutable data and structures. In these cases, the contracts were checked only partly, because mutation could subvert the checks later. The asterisks on the "objects" line indicates that contracts were supported for our Java-like object system, but at a high runtime cost to objects that did not use 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
functions	✓	✓	✓
immutable data	✓	✓	✗
mutable data	✓	✓	✓
opaque structures	✓	✓	✗
objects	✓	✓	✓

The unsupported cases in this table are unsupported by design; those points in the spectrum do not make sense, as explained above. 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. Instead, the run-time system provides *chaperones* and *impersonators* as first-class values, with which it is then possible to implement chaperone and impersonator contracts as well as additional proxy patterns.

Figure 1 shows part of the Racket chaperone and impersonator API.[3] The API includes a constructor for each primitive datatype that supports interposition on its operations. The `chaperone-of?` 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 and satisfies the `procedure?` predicate. 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 chaperone the arguments to original function or the results from the original function.

[3] The complete API is about twice as large (Flatt and PLT 2010, §13.5).

`(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

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`.

An `interpose-proc` can return an extra value to interpose on the result of the procedure. The extra value 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 the `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))))
```

Here, the `blame` function takes an identifier which names the blamed party and a string that describes the reason that party broke the contract. The parties `provider` and `client` are the modules that export and import the `sqr` function, respectively.

The `->` contract constructor creates a chaperone to implement a function contract when the argument and result contracts are flat contracts, like `real?` and `nonnegative-real?`, or chaperone contracts. If the `->` contract constructor encounters an impersonator contract like α, 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.

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 this value or a chaperone thereof. The protocol for `interpose-set` is essentially the same.

For example, the contract for `partial-sums!`,

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

is implemented using `chaperone-vector`. The installed chaperone ensures the client supplied a vector that contains only numbers and constrains `partial-sums!` from changing the vector to include non-numbers:

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

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

It may seem that an `interpose-ref` needs only an index, since the `interpose-ref` provided to `chaperone-vector` could capture `vec` in its closure and extract the original value from `vec`. Passing `vec`, 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 to facilitate composition it instead returns a value to be installed.

The `impersonate-vector` function works the same way as `chaperone-vector`, but without chaperone checks on replacement values. In addition, `impersonate-vector` is limited to *mutable* vectors. 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.

3.3 Chaperoning and Impersonating Structures

As noted in section 2.3, Racket's `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,

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

defines the constructor `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-weight` to the extract the second field of an instance. Since the second field is annotated `#:mutable`, `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 a sequence of pairs of operations and interposition procedures. For example, a contract on a `fish` instance `dory`—to ensure that `dory` is blue and 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.

In principle, every value in Racket is a structure, and `chaperone-vector` and `chaperone-procedure` internally use `chaperone-struct` to apply chaperones through interposition of private accessors and mutators.[4] 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.

The `impersonate-struct` function works the same way as `chaperone-struct`, but without chaperone checks on replacement values. Just like `impersonate-vector`, `impersonate-struct` only allows interposition on mutable fields of a structure.

[4] In practice, most (but not all) procedures and vectors have specialized representations that are exploited by the just-in-time compiler.

4. Interposition beyond contracts

Chaperones and impersonators are not limited in use to implementing contracts. To demonstrate additional uses, we discuss the implementation of remote objects that provide a local view on a remote service with impersonators, and we discuss the implementation of revocable membranes (Miller 2006) with chaperones.

4.1 Remote Objects

To illustrate remote objects, we use impersonators to implement a view on IMAP accounts. The resulting view is manipulated by the programmer like normal, local data, but the view retrieves information lazily from the IMAP server using the interposition capability of impersonators. Since the data we retrieve from the server is not the same as the data stored in the wrapped local data structures, we must use impersonators, not chaperones.

First, we require an existing Racket library that provides IMAP functionality:

```
(require net/imp)
```

Building on this library, an IMAP session is represented as a hash table that maps mailbox names to mailboxes. A hash-table impersonator interposes on the lookup, assign, remove, and key-enumeration operations of a hash table, where the lookup interposition function first filters the key and returns another procedure to adjust the lookup result. To simplify the presentation, the impersonator for the mailbox hash allows only lookup and key-enumeration (replacing or removing mailboxes is not allowed), effectively making the table read-only.

```
(define (imap-hash server user pass)
  (define-values (conn c r)
    (imap-connect server user pass "INBOX"))
  (define mailboxes
    (map second (imap-list-child-mailboxes conn #f)))
  (imap-disconnect conn)
  (impersonate-hash
   (make-hash (map (λ (m) (cons m #f)) mailboxes))
   (λ (h k)
     (values
      k
      (λ (h k v)
        (imap-mailbox-vector server user pass k))))
   (λ (h k v) (error "assign not allowed"))
   (λ (h k) (error "remove not allowed"))
   (λ (h k) k)))
```

To client code, the result of `imap-hash` looks like a hash table `ht` indexed by byte-string mailbox names, so that a client who wants to read messages from the "Ultra Mega" mailbox would access messages using `(hash-ref ht #"Ultra Mega")`. Instead of returning the placeholder `#f` that is stored in the hash table, the table access triggers a download of the "Ultra Mega" messages via `imap-mailbox-vector`.

Ultimately, each message contains four parts: (1) the message position within the mailbox, (2) the headers of the message, (3) the body of the message, and (4) the IMAP flags associated with the message.

```
(struct message (pos
                 [headers #:mutable]
                 [body #:mutable]
                 [flags #:mutable]))
```

Fields other than `pos` in `message` are marked as mutable so that the impersonator can retrieve them lazily. That is, the impersonator for a `message` connects to the IMAP server when any field other than `pos` is accessed:

```
(define (imap-message server user pass mbox pos)
  (define (fetch-field field)
    (define-values (conn c r)
      (imap-connect server user pass mbox))
    (define val
      (first (first
              (imap-get-messages conn
                                 (list pos)
                                 (list field)))))
    (imap-disconnect conn)
    val)
  (impersonate-struct
   (message pos #f #f #f)
   message-headers
   (λ (s f) (fetch-field 'header))
   message-body
   (λ (s f) (fetch-field 'body))
   message-flags
   (λ (s f) (fetch-field 'flags))))
```

4.2 Revocable Membranes

Revocable membranes (Donnelley 1976; Miller 2006; Rajunas 1989) allow one software component to share values that contain behavior or state with another component, but to later revoke access to those values when needed. Such membranes are useful for providing untrusted parties time-limited access to sensitive data, and they have been verified to enforce the appropriate security properties (Murray 2010).

Because the wrapped values should react identically to the original values except for errors due to the membrane being revoked, chaperones are sufficient to implement membranes, despite the invariants that Racket enforces. In our implementation, a component provides chaperoned versions of values that may contain behavior or state, and the chaperone checks for revocation before applying functions and accessing or mutating state. In addition to demonstrating the expressiveness of chaperones, previous membrane designs rely on a universal message-passing interface, whereas chaperones support membranes for data such as vectors and structures.

To represent the membrane, we use a structure type that contains one field, `revoked?`, that initially contains `#f`. We provide two utility functions on membranes:

- `check-membrane`, which errors if a membrane has been revoked, and
- `revoke-membrane`, which revokes a membrane.

```
(struct membrane ([revoked? #:mutable #:auto])
        #:auto-value #f)


(define (check-membrane m)
  (when (membrane-revoked? m)
    (error "membrane revoked!")))


(define (revoke-membrane m)
  (set-membrane-revoked?! m #t))
```

Van Cutsem and Miller (2010) and Austin et al. (2011) implement revocable membranes via a generic proxy mechanism that is uniform for all types of value. Thus, they need to implement only one proxy wrapper that appropriately recurs on any values returned from operations. In our system, however, different types of values have different chaperone interfaces, so we have a central dispatching function, `membrane-value`, that consumes a value and adds a membrane to it.

For basic values, like strings or numbers, no wrapping is necessary. For values like pairs, the pair need not be wrapped, but the values contained in the pair may require wrapping. For procedures and vectors, we simply use the appropriate type of chaperone. The interposing functions check the membrane, and if it is not yet revoked, allow the operation to proceed. Attempting to pass a value through a membrane causes an error if the value cannot be wrapped and may contain behavior or state.

```
(define (membrane-value v m)
  (cond
    [(or (null? v) (string? v) (number? v)) v]
    [(pair? v)
     (cons (membrane-value (car v) m)
           (membrane-value (cdr v) m))]
    [(vector? v)
     (membrane-vector v m)]
    [(procedure? v)
     (membrane-procedure v m)]
    [(struct? v)
     (membrane-struct v m)]
    [else
     (error "Value cannot flow through membrane" v)]))
```

To build a membrane, we simply create a chaperone that calls `check-membrane` before allowing any operation. Here's the code to add the chaperone to a vector.

```
(define (membrane-vector v m)
  (chaperone-vector
   v
   (λ (v i r)
     (check-membrane m)
     (membrane-value r m))
   (λ (v i r)
     (check-membrane m)
     (membrane-value r m))))
```

The other operations are similar (although the structure membrane is more complex, since structures have many operations in Racket). The full implementation is given in the technical appendicies that accompany this paper.

5. Reasoning about Reasonable Interposition

This section presents a formal model of our chaperones and impersonators. Using the model, we can formally establish limits on allowed interposition, and we can state a precise theorem that illustrates the desired properties of chaperones. To begin, we define VectorRacket, a subset of Racket with both mutable and immutable vectors. We then extend VectorRacket to include chaperones and impersonators for vectors and present our theorem for chaperone erasure.

5.1 Constraining Interposition

By requiring that a chaperone or impersonator is attached to a value before it flows into otherwise oblivious code, the design of chaperones and impersonators implicitly constrains the interposition to that specific value. After a value is chaperoned, however, the dynamic behavior of the chaperone is hardly constrained; it is certainly not constrained to purely functional behavior. An interposition function associated with a chaperone can use the full power of Racket, which means that it can print output, modify global variables, or even change mutable arguments as they flow through the interposition layer.

At first glance, a lack of constraints on side effects may seem like an open invitation to breaking existing invariants of the programming language. An externally visible side effect that is performed through a chaperone, however, is no different from a side effect that is concurrently performed by another thread. A chaperone may gain access to local values that might not otherwise be exposed to other threads, but in a mostly functional language like Racket, those arguments tend to be immutable, which means that extra side effects through chaperones are constrained already by the immutability of the data.

In contrast, impersonators are prohibited from acting on immutable values, precisely to ensure that the invariants of immutability are preserved. For example, extracting the value of a field from an immutable structure should always return the same result; chaperoned structures still preserve this behavior, thanks to the `chaperone-of?` check that impersonators skip.

Since chaperones and impersonators offer little additional possibilities for side effects compared to threads, and since Racket libraries must already account for the possibility of concurrent threads when checking and enforcing invariants, chaperones and impersonators create few new complications on that front. We are therefore concerned with the ability of a chaperone or impersonator to change the result that is produced by an operation, and hence our investigation concentrates on that problem. To further simplify the model, we restrict attention to procedures and mutable and immutable vectors, since the structure-type generativity can be simulated through vectors, procedures, and hidden type tags.

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

Figure 2: VectorRacket Syntax

5.2 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 two predicates: `equal?` to compare two values structurally 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 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 with the formulation of our formal results), and an expression. The langauge for expressions is nearly the same as the set of surface-level expressions, with the exception that the production $(\texttt{loc}\ x)$ 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 [βv] 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 `immutable?` predicate detects immutable vectors. The remaining rules handle vec-

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

Figure 3: VectorRacket Evaluator (function clauses in order)

tor allocation, access, and update, where vector allocation records whether it was allocated by an interposition (i.e., the program state's boolean).

5.3 VectorRacket with Chaperones

Figure 5 extends the syntax of VectorRacket with 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 has entered 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:

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

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

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

The [out-cvec-ref] and [in-cvec-ref] rules of figure 6 handle `vector-ref` for chaperones. 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.

$(s\ b\ E[(\lambda\ (y\ ...)\ e)])$ [procedure]
$\longrightarrow (s[x \mapsto (\lambda\ (y\ ...)\ e)]\ b\ E[(\mathtt{loc}\ x)])$
where x fresh

$(s\ b\ E[((\mathtt{loc}\ x_p)\ v\ ...)])$ [βv]
$\longrightarrow (s\ b\ E[\{x{:=}v, ...\}\ e])$
where $(\lambda\ (x\ ...)\ e) = s(x_p)$, $|(v\ ...)| = |(x\ ...)|$

$P[(\mathtt{let}\ ([x\ v]\ ...)\ e)] \longrightarrow P[\{x{:=}v, ...\}\ e]$ [let]

$P[(\mathtt{if}\ v\ e_1\ e_2)] \longrightarrow P[e_1]$ [ift]
where $v \neq \#\mathrm{f}$

$P[(\mathtt{if}\ \#\mathrm{f}\ e_1\ e_2)] \longrightarrow P[e_2]$ [iff]

$(s\ b\ E[(\mathtt{error}\ '\mathit{variable})])$ [error]
$\longrightarrow (s\ b\ (\mathtt{error}\ '\mathit{variable}))$
where $[\,] \neq E$

$(s\ b\ E[(\mathtt{equal?}\ v_1\ v_2)])$ [equal?]
$\longrightarrow (s\ b\ E[\mathsf{equal}[\![s, v_1, v_2]\!]])$

$(s\ b\ E[(\mathtt{vector}\ v\ ...)])$ [vector]
$\longrightarrow (s[x \mapsto (\mathtt{vector}\ b\ v\ ...)]\ b\ E[(\mathtt{loc}\ x)])$
where x fresh

$(s\ b\ E[(\mathtt{immutable?}\ v)])$ [immut]
$\longrightarrow (s\ b\ E[\mathsf{immutable}[\![s, v]\!]])$

$(s\ b\ E[(\mathtt{vector\text{-}immutable}\ v\ ...)])$ [immvec]
$\longrightarrow (s[x \mapsto (\mathtt{vector\text{-}immutable}\ v\ ...)]\ b\ E[(\mathtt{loc}\ x)])$
where x fresh

$(s\ b\ E[(\mathtt{vector\text{-}set!}\ (\mathtt{loc}\ x)\ n\ v_{new})])$ [vector-set!]
$\longrightarrow (s[x \mapsto (\mathtt{vector}\ b\ v_1\ ...\ v_{new}\ v_3\ ...)]$
$\quad b\ E[(\mathtt{void})])$
where $(\mathtt{vector}\ b\ v_1\ ...\ v_2\ v_3\ ...) = s(x)$,
$\quad |(v_1\ ...)| = n$

$(s\ b_1\ E[(\mathtt{vector\text{-}ref}\ (\mathtt{loc}\ x)\ n)])$ [mvec-ref]
$\longrightarrow (s\ b_1\ E[v_2])$
where $(\mathtt{vector}\ b_2\ v_1\ ...\ v_2\ v_3\ ...) = s(x)$,
$\quad |(v_1\ ...)| = n$

$(s\ b_1\ E[(\mathtt{vector\text{-}ref}\ (\mathtt{loc}\ x)\ n)])$ [immvec-ref]
$\longrightarrow (s\ b_1\ E[v_2])$
where $(\mathtt{vector\text{-}immutable}\ v_1\ ...\ v_2\ v_3\ ...) = s(x)$,
$\quad |(v_1\ ...)| = n$

Figure 4: VectorRacket Reductions

The [out-cvec-ref] and [in-cvec-ref] rules of figure 6 handle `vector-set!` on vector chaperones in a similar manner. The [ivec-ref] and [ivec-ref] rules handle `vector-ref` and `vector-set!` on impersonators, which do not require `chaperone-of?` checks. The [cvec] and [ivec] rules handle chaperone and impersonator construction.

The `chaperone-of?` primitive defers to the metafunction `chaperone-of` of figure 6. The result of `chaperone-of` is #t for syntactically identical values. If both arguments are immutable vectors of the same length, the elements are checked point-wise. If the first argument is a location in the store that points at a chaperone, the metafunction recurs using the chaperoned value. Otherwise, `chaperone-of` returns #f.

5.4 Chaperone Erasure

To state our central theorem on chaperone proxies, we need the notion of *chaperone erasure* for a subset of programs. If a

$prim$ ::= .... | `chaperone-vector` | `chaperone-of?` | `impersonate-vector`
sv ::= .... | (`chaperone-vector` $l\ m\ o$) | (`impersonate-vector` $l\ m\ o$)
e ::= .... | (`set-marker` e) | (`clear-marker` e)

Figure 5: VectorRacket Chaperone Syntax Extensions

$(s\ \#\text{f}\ E[(\texttt{vector-ref}\ (\texttt{loc}\ x)\ n)])$ [out-cvec-ref]
$\longrightarrow (s\ \#\text{f}\ E[(\texttt{let}\ ([\texttt{old}\ (\texttt{vector-ref}\ l\ n)])$
$\quad(\texttt{let}\ ([\texttt{new}\ (\texttt{set-marker}\ (m\ l\ n\ \texttt{old}))])$
$\quad\quad(\texttt{clear-marker}$
$\quad\quad\ (\texttt{if}\ (\texttt{chaperone-of?}\ \texttt{new}\ \texttt{old})$
$\quad\quad\quad\ \texttt{new}$
$\quad\quad\quad\ (\texttt{error}\ \texttt{'bad-cvref})))))])$
where $(\texttt{chaperone-vector}\ l\ m\ o) = s(x)$

$(s\ \#\text{t}\ E[(\texttt{vector-ref}\ (\texttt{loc}\ x)\ n)])$ [in-cvec-ref]
$\longrightarrow (s\ \#\text{t}\ E[(\texttt{let}\ ([\texttt{old}\ (\texttt{vector-ref}\ l\ n)])$
$\quad(\texttt{let}\ ([\texttt{new}\ (m\ l\ n\ \texttt{old})])$
$\quad\quad(\texttt{if}\ (\texttt{chaperone-of?}\ \texttt{new}\ \texttt{old})$
$\quad\quad\quad\ \texttt{new}$
$\quad\quad\quad\ (\texttt{error}\ \texttt{'bad-cvref}))))])$
where $(\texttt{chaperone-vector}\ l\ m\ o) = s(x)$

$(s\ b\ E[(\texttt{set-marker}\ e)]) \longrightarrow (s\ \#\text{t}\ E[e])$ [setm]

$(s\ b\ E[(\texttt{clear-marker}\ e)]) \longrightarrow (s\ \#\text{f}\ E[e])$ [clearm]

$(s\ \#\text{f}\ E[(\texttt{vector-set!}\ (\texttt{loc}\ x)\ n\ v)])$ [out-cvec-set!]
$\longrightarrow (s\ \#\text{f}\ E[(\texttt{let}\ ([\texttt{new}\ (\texttt{set-marker}\ (o\ l\ n\ v))])$
$\quad(\texttt{clear-marker}$
$\quad\ (\texttt{if}\ (\texttt{chaperone-of?}\ \texttt{new}\ v)$
$\quad\quad\ (\texttt{vector-set!}\ l\ n\ \texttt{new})$
$\quad\quad\ (\texttt{error}\ \texttt{'bad-cvset}))))])$
where $(\texttt{chaperone-vector}\ l\ m\ o) = s(x)$

$(s\ \#\text{t}\ E[(\texttt{vector-set!}\ (\texttt{loc}\ x)\ n\ v)])$ [in-cvec-set!]
$\longrightarrow (s\ \#\text{t}\ E[(\texttt{let}\ ([\texttt{new}\ (o\ l\ n\ v)])$
$\quad(\texttt{if}\ (\texttt{chaperone-of?}\ \texttt{new}\ v)$
$\quad\quad\ (\texttt{vector-set!}\ l\ n\ \texttt{new})$
$\quad\quad\ (\texttt{error}\ \texttt{'bad-cvset})))])$
where $(\texttt{chaperone-vector}\ l\ m\ o) = s(x)$

$(s\ b\ E[(\texttt{vector-ref}\ (\texttt{loc}\ x)\ n)])$ [ivec-ref]
$\longrightarrow (s\ b\ E[(m\ l\ n\ (\texttt{vector-ref}\ l\ n))])$
where $(\texttt{impersonate-vector}\ l\ m\ o) = s(x)$

$(s\ b\ E[(\texttt{vector-set!}\ (\texttt{loc}\ x)\ n\ v)])$ [ivec-set!]
$\longrightarrow (s\ b\ E[(\texttt{vector-set!}\ l\ n\ (o\ l\ n\ v))])$
where $(\texttt{impersonate-vector}\ l\ m\ o) = s(x)$

$(s\ b\ E[(\texttt{chaperone-vector}\ l\ m\ o)])$ [cvec]
$\longrightarrow (s[x \mapsto (\texttt{chaperone-vector}\ l\ m\ o)]\ b\ E[(\texttt{loc}\ x)])$
where $\mathsf{isvector}[\![s, l]\!], x$ fresh

$(s\ b\ E[(\texttt{impersonate-vector}\ l\ m\ o)])$ [ivec]
$\longrightarrow (s[x \mapsto (\texttt{impersonate-vector}\ l\ m\ o)]\ b\ E[(\texttt{loc}\ x)])$
where $\mathsf{ismutable}[\![s, l]\!], x$ fresh

$(s\ b\ E[(\texttt{chaperone-of?}\ v_1\ v_2)])$ [cof]
$\longrightarrow (s\ b\ E[\mathsf{chaperone\text{-}of}[\![s, v_1, v_2]\!]])$

$\mathsf{chaperone\text{-}of}[\![s, v, v]\!] = \#\text{t}$
$\mathsf{chaperone\text{-}of}[\![s, (\texttt{loc}\ x), v_2]\!] = \mathsf{chaperone\text{-}of}[\![s, l, v_2]\!]$
where $(\texttt{chaperone-vector}\ l\ m\ o) = s(x)$
$\mathsf{chaperone\text{-}of}[\![s, (\texttt{loc}\ x), (\texttt{loc}\ y)]\!] = \wedge[\![\mathsf{chaperone\text{-}of}[\![s, v_1, v_2]\!], ...]\!]$
where $(\texttt{vector-immutable}\ v_1\ ...) = s(x)$,
$(\texttt{vector-immutable}\ v_2\ ...) = s(y), |(v_1\ ...)| = |(v_2\ ...)|$
$\mathsf{chaperone\text{-}of}[\![s, v_1, v_2]\!] = \#\text{f}$

Figure 6: VectorRacket Chaperone Reductions and Metafunctions

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.

Theorem 1. *For all e, if* $\mathsf{Eval}[\![e]\!] = v$ *and that evaluation contains no reductions whose left hand side is of the form* $(s_b\ \#\text{t}\ E[(\texttt{vector-set!}\ (\texttt{loc}\ x)\ v_b\ n)])$ *where* $s_b(x) =$ $(\texttt{vector}\ \#\text{f}\ v_v\ ...)$*, then* $\mathsf{Eval}[\![e_2]\!] = v$*, where e_2 is the same as e but where any uses of* `chaperone-vector` *are replaced with* $(\lambda\ (v\ x\ y)\ v)$.

Proof sketch To prove this theorem, we look at the trace of reductions for both the unerased and erased programs. First, we set up an approximation relation that relates program traces in the unerased reduction trace to program states in the erased reduction trace. Erased program states are approximately equal to unerased program states when they contain the same expression, modulo the replacement of `chaperone-vector` with $(\lambda\ (v\ x\ y)\ v)$, and the graph of memory allocated by the main program is the same in both, modulo any chaperones allocated by the unerased program. We then show that VectorRacket reduction respects this approximation, and that values from approximated states are equal under our evaluation function.

The full proof is given in the technical appendicies.

6. Performance

Although our motivation for adding chaperones and impersonators to Racket is to increase the expressiveness of the contract system, performance is a major concern. Our primary concern is that support for chaperones, impersonators, and contracts is "pay as you go" as much as possible; 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, which should not impose excessive overheads on programs that use them.

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. We therefore concentrate our performance analysis on the overhead of using chaperones and impersonators, both by comparing this overhead to programs without interposition as well as comparing the performance of chaperones and impersonators to other proxy systems.

The source code for all of the benchmarks presented in this section are in the technical appendices.

6.1 Procedure Performance

To measure the performance overhead of chaperones and impersonators for procedures, we start with microbenchmarks comparing Racket to two variants of Scheme—Chicken and Larceny—plus two variants of Javascript—V8 and SpiderMonkey, with JägerMonkey and type inference (Hackett and Guo 2012) enabled for the latter.

The first set of benchmarks involve 10 million calls to the identity function in increasingly expensive configurations, with results shown in figure 7:

- **direct** — Each call is a direct call, which is inlined by the Racket compiler and most others.
- **indirect** — Each call is through a variable that is assigned to the identity function. Since Racket is designed for functional programming, its compiler makes no attempt to see through the assignment, so the assignment disables inlining of the function. For Javascript, the indirection is similarly just an assignment, but Javascript implementations tend to see through such assignments; and we make no attempt to obfuscate the program further from Javascript JITs.
- **wrapped** — Each call is through a function that calls the identity function. Like **indirect**, both the identity function and its wrapper are hidden from the Racket compiler via assignments to prevent inlining—and therefore to simulate at the source level the kind of indirections that a chaperone or impersonator create.

Run times in milliseconds

	Racket	Chicken	Larceny	V8	SM
direct	29	115	66	42	37
indirect	123	226	63	40±	46
wrapped	176	218	85±	36	95
wrapped+check	358	446	163	195	139
wrapped+return	562±	525	197	401	1,211
proxy	–	–	–	1,903±	1,931
impersonate	922	–	–	–	–
chaperone	920	–	–	–	–
impersonate+return	1,642	–	–	–	–
chaperone+return	1,676	–	–	–	–
church	1,258	1,113	706	2,282	9,368
church-wrap	4,067	8,014	3,458	8,071	26,785
church-proxy	–	–	–	41,717	79,214
church-chaperone	43,805	–	–	–	–
church-chaperone/a	4,653	–	–	–	–
church-contract	7,607	–	–	293,149†	226,135†

Average of three runs; ± inidicates a standard deviation between 5% and 10% of the average, while all others are within 5% of the average. Benchmark machine: MacBook Air, 1.8 GHz Intel Core i7, 4GB running OS X 10.7.4. Implementations: Racket v5.3.0.16 (git b0f81b5365) 64-bit, Chicken v4.7.0 64-bit using `-O3 -no-trace`, Larceny v0.98b1 32-bit using `-r6rs -program`, V8 shell v3.12.19 (git cb989e6db8) 64-bit using `--harmony`, SpiderMonkey v1.8.5+ (hg 23a7ba542bb5) 64-bit using `-m -n`, where † uses contracts generated by Contracts.coffee 0.2.0 (Disney 2012), and the V8 run further uses `--noincremental-marking` to avoid a problem with weak maps.

Figure 7: Procedure-call microbenchmark results

- **wrapped+check** — Each call to a function like **wrapped** is generated by a higher-order function that accepts a function to convert to the original function's argument and another to convert the result; the identity conversion is provided. This variant simulates the old implementation of contracts in Racket by using `lambda` as the interposition mechanism instead of `impersonate-procedure` or `chaperone-procedure`.
- **wrapped+return** — Like **wrapped**, but in addition to returning the result of the identity function, the wrapper returns another function (also the identity function) that the caller should apply to the result. This variant simulates interposition on both the arguments and results of a wrapped function as performed by chaperones and impersonators, but staying within normal function calls.
- **proxy** — For Javascript, calls the identity function through a proxy created by `Proxy.createFunction` (Van Cutsem and Miller 2010), which is roughly analogous to calling a function through an impersonator.
- **impersonate** and **chaperone** — For Racket, calls the identity function through an impersonator and chaperone, respectively, interposing only on the arguments of the function.

- **impersonate+return** and **chaperone+return** — For Racket, calls the identity function through an impersonator and chaperone, respectively, interposing on both the arguments and results of the function with the identity conversion.

Although it is difficult to compare performance across languages with different semantics, these results suggest that Racket's performance is not out of line with other dynamic-language implementations, both in terms of its baseline performance and in the performance of chaperones and impersonators compared to Javascript proxy implementations.

In addition to calling the identity function 10 million times, we also run a λ-calculus computation of factorial using Church numerals, which stresses higher-order functions. Again, we try several variants with results shown in figure 7:

- **church** — Computes the factorial of 9 using Church numerals.
- **church-wrap** — Like **church**, but with wrapping functions to simulate contract checks. The simulated contracts are higher order, involving about 360 wrappers and just short of 10 million applications of wrapped functions.
- **church-proxy** — Like **church-wrap**, but implementing wrappers with Javascript proxies.
- **church-chaperone** — Like **church-wrap**, but implementing wrappers with chaperones.
- **church-chaperone/a** — Like **church-chaperone**, but recognizing `(any/c . -> . any)` contracts to avoid unnecessary chaperoning in that case, which is the kind of shortcut that Racket's contract system detects.
- **church-contract** — Like **church-wrap**, but using either Racket contracts, which are in turn implemented with chaperones, or Contracts.coffee (Disney 2012), which compiles to JavaScript proxies.

The initial **church** variants corroborate the other microbenchmark results. The **church-contract** result shows Racket's performance to be significantly better than the JavaScript versions, but this is likely because of optimizations that the Racket contract library performs, and not due to differences at the proxy/chaperone layer.

To check the effect of chaperones in realistic applications, we use a few existing Racket programs and tests that make heavy use of functions with contracts:

- **make guide** — Builds a representation of the Racket Guide, which involves many constructors such as `section` and `element`, as well as the decoding of some string literals into typesetting elements. Most contract checking involves the constructors.
- **render guide** — Renders the documentation from **make guide** to HTML. The relevant contracts are on structure accessors (but not individual structure instances) and on functions to resolve cross references.

	msecs	makes	calls	
make guide	10,792	14,051	90,924	chaperone
	10,818	14,049	90,922	impersonate
	10,606	13,602	89,914	no interpose
	10,467	13,602	0	no chaperone
proxy overhead: 1%				
additional chaperone overhead: 0%				
contract checking overhead: 2%				
render guide	3,727	2,188	1,846,966	chaperone
	3,741	2,188	1,846,966	impersonate
	2,044	172	1,730,158	no interpose
	1,889	172	0	no chaperone
proxy overhead: 8%				
additional chaperone overhead: 0%				
contract checking overhead: 83%				
keyboard	7,258	160	1,244,705	chaperone
	7,253	160	1,244,705	impersonate
	5,321	0	1,244,545	no interpose
	5,182	0	0	no chaperone
proxy overhead: 3%				
additional chaperone overhead: 0%				
contract checking overhead: 36%				
slideshow	5,180	6,467	208,263	chaperone
	5,168	6,467	208,263	impersonate
	4,776	4,605	206,401	no interpose
	4,663	4,605	0	no chaperone
proxy overhead: 2%				
additional chaperone overhead: 0%				
contract checking overhead: 8%				
plot	2,394	1,755	274,768	chaperone
	2,362	1,755	274,768	impersonate
	1,886	1,750	136,923	no interpose
	1,854	1,750	0	no chaperone
proxy overhead: 2%				
additional chaperone overhead: 1%				
contract checking overhead: 25%				
typecheck	47,816	1,975,414	7,440,497	chaperone
	47,302	1,975,414	7,440,497	impersonate
	24,144	918,343	3,482,644	no interpose
	22,610	918,491	0	no chaperone
proxy overhead: 7%				
additional chaperone overhead: 1%				
contract checking overhead: 96%				

Figure 8: Realistic procedure benchmark results

Run times in milliseconds

	Racket	∼Racket	Chicken	Larceny	V8	SM
bubble	1,317	1,315	4,341	680	463	552
proxy	–	–	–	–	124,215	97,226
chaperone	6,358	29,171	–	–	–	–
unsafe	885	–	3,020±	815	–	–
unsafe*	692	–	–	–	–	–

∼Racket corresponds to Racket without specialized JIT handling of chaperoned vectors. Chicken in unsafe mode corresponds to adding `-unsafe`. Larceny in unsafe mode corresponds to setting `(compiler-switches 'fast-unsafe)` before using `compile-file`.

Figure 9: Vector microbenchmark results

	msecs	makes	refs	
ode-apply	10,632	1,456,221	40,817,268	chaperone
	10,236	1,456,221	40,817,268	impersonate
	9,265	1,456,221	40,817,268	no interpose
	7,794	1,456,221	0	no chaperone
	5,532	0	0	no procedure

vector proxy overhead: 19%
additional vector chaperone overhead: 4%
vector contract checking overhead: 10%
procedure and vector contract use overhead: 92%

Figure 10: Realistic vector benchmark results

- **keyboard** — A test of DrRacket's responsiveness to keyboard events, which simulates a user typing "(abc)" 400 times. DrRacket reacts by adding the characters to an editor buffer, matching parentheses and syntax-coloring through an associated coroutine (that is covered in the timing result). Contracts from many different Racket libraries are involved.
- **slideshow** — Construction of a Slideshow talk that includes many animations, so that the slide set contains over 1000 frames. The relevant contracts are mainly on the construction of "pict" values that are composed to form the animation frames.
- **plot** — Renders a 3-D plot to a PNG file. The relevant contracts are mainly on the drawing library.
- **typecheck** — Runs the Typed Racket compiler on test input. The Typed Racket compiler uses many higher-order contracts on its internal modules.

Figure 8 shows timing results. For each program, we show the run time, number of created procedure chaperones ("makes"), and number of calls to chaperoned procedures ("calls"). We then show how the timing changes when chaperones are replaced internally with impersonators (skipping the `chaperone-of?` check), when application of a chaperone procedure redirects internally to the chaperoned procedure (avoiding the overhead of the interposition procedures that actually check contracts), and when `chaperone-procedure` is internally short-circuited to just return the procedure (effectively disabling the contracts in the original code). The results show that the cost of checking contracts is sometimes quite significant—as exposed by the time difference when interposition procedures are skipped—while the overhead of the core chaperone and impersonator mechanism is negligible or small for these programs.

6.2 Vector Performance

Our microbenchmark for vector performance is bubble sort on a vector of 10,000 integers in reverse order. Figure 9 shows timing results, where the "∼Racket" column corresponds to Racket with specific JIT support for chaperoned vectors removed; we include the column to demonstrate how building chaperone and impersonator support into the run-time system allows the JIT to substantially reduce the overhead of proxies on vectors. The **proxy** variant for Javascript uses `makeForwardingHandler` from Van Cutsem and Miller (2010), while **chaperone** uses a Racket vector chaperone.

Besides the overhead of proxies when used, and in contrast to procedure chaperones and inspectors, chaperones and inspectors for vectors are not completely "pay as you go" in Racket. The table in Figure 9 includes an **unsafe** row to show the performance of bubble sort when `vector-ref` operations are replaced by `unsafe-vector-ref`. While the `unsafe-vector-ref` operation assumes that its arguments are a vector and an in-range index, the vector may be a chaperoned or impersonated vector. The **unsafe*** row shows performance using `unsafe-vector*-ref`, which assumes a non-chaperoned, non-impersonated vector. These unsafe operations are suitable for use in macro expansions or typed contexts where the operations are statically known to be safe, and in most such contexts, `unsafe-vector-ref` must be used. The difference in performance between **unsafe** and **unsafe*** thus reflects a price imposed on `unsafe-vector-ref` by the existence of chaperones and impersonators. The cost is small, though not negligible for the microbenchmark.

We use the `williams/science.plt` PLaneT package to illustrate the impact on realistic programs with extensive use of vector contracts. Many functions from this package expect vectors of real numbers as inputs. We adjusted a test case for `ode-evolve-apply` so that it performs 1,456,221 iterations; the argument vector in the test is short, but the vector is accessed frequently, so that 40 million accesses are chaperoned. Figure 10 shows the benchmark results; as for the benchmark suite for procedures, we show how the timing changes when chaperones are replaced internally with impersonators, when access of a chaperoned vector directly accesses the vector content (skipping the interposition that checks for a real number), and when `chaperone-vector` is internally short-circuited to re-

Run times in milliseconds

	Racket	Chicken	Larceny	V8	SM
direct	902	2,324±	756±	293	349
proxy	–	–	–	39,345	21,768
chaperone	5,835	–	–	–	–
unsafe	289	1,250±	800	–	–
unsafe*	285	–	–	–	–

Figure 11: Structure microbenchmark results

	msecs	makes	refs	Mbytes	
lazy	18,616	6,026,413	235,002,269	446	chaperone
	22,122	0	0	227	original
	2,824	0	0	153	no contract

structure contract checking overhead: 559%
chaperones versus original: 84%

Figure 12: Realistic structure benchmark results

turn its first argument (as if no contracts were present in the original code). Finally, we show the time when both `chaperone-vector` and `chaperone-procedure` are short-circuited, which completely removes the contract from `ode-evolve-apply`. The cost of checking the contract on vector elements is small, while the use of a contract overall is a substantial cost. The overhead imposed specifically by the chaperone and impersonator mechanism is more substantial than in the case of procedures, but it is in line with the overall cost of using contracts.

6.3 Structure Performance

Our microbenchmark for structure performance is to access the first field of a two-element structure 100 million times. Figure 11 shows timing results that are analogous to the vector benchmarks. Although an unsafe structure reference via `unsafe-struct-ref` must pay for the existence of chaperones and impersonators, the extra test makes little difference in our benchmark, as reflected in the close results for **unsafe** and **unsafe***; that the difference appears so small is probably due to accessing the same structure repeatedly in the microbenchmark, so that the type tag is always in cache.

For a more realistic benchmark, we re-use a benchmark from Findler et al. (2007)'s work on lazy contracts that consists of replaying a trace of heap operations from Felzenszwalb and McAllester (2006)'s vision algorithm. We replay the trace in a binomial heap. Contracts on the heap operations ensure that the heap is well-formed, but checking them at every step is prohibitively expensive, so the contracts are checked lazily. These contracts ensure that the structure of the heap is well-formed as far as it is explored, but unexplored parts of the heap are not checked. The original implementation of lazy structure contracts required that the program is changed to use a special structure-declaration form, while the new chaperone-based version works with the original structure declaration. The chaperone-based version is also lazier, in that structure checks are triggered per-field; with the old implementation, an access of any of a structure's fields would trigger checking on all of the fields. An advantage of the old implementation, however, was that it could drop unchecked field values after checking them, while the chaperone-based implementation merely prevents the unchecked variants from being accessed.

Figure 12 shows the result of running the benchmark on a picture of a koala's face. The figure shows the running times, number of created structure chaperones, number of chaperoned structure references, and peak memory use of the benchmark in three configurations: using the chaperone-based implementation of lazy contracts, using the original implementation of lazy contracts, and using no contracts. The chaperone-based implementation is faster than the original implementation, mainly due to its finer granularity of contract checking, but it also uses more memory, since it retains unchecked versions after checked versions are available. Overall, the benchmark results show that chaperones perform well for lazy structure contracts.

Aside from these benchmarks, structure chaperones in Racket directly improved the performance of Racket's class system. Prior to support for chaperones, the class system implemented object-specific wrappers that operated like impersonators. 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 impersonator 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 impersonators, this slowdown was completely eliminated.

6.4 Discussion

Chaperones and impersonators, as implemented in Racket, are expressive and expensive constructs; microbenchmarks indicate a factor of 5 to 10 in run time over normal, uninlined function call or a vector or structure access (see Figure 7). Despite this cost, real applications experience a much lower slowdown when using them, up to a factor of 2 and usually less (see the "contract checking overhead" results in Figures 8 and 10), although lazy contracts can be expensive (see Figure 12). Considering our microbenchmarks and our measurements of larger applications as a whole, we draw the following conclusions:

- Racket's baseline performance and its proxy performance are on par with other production systems offering

similar functionality (see Figure 7). Slower performance from Racket's impersonators compared to JavaScript's proxies would have been a cause for concern, since the overall design of Racket generally makes it easier to implement efficiently than JavaScript.

- The design of interposition to Racket imposes almost no cost on the remainder of the system when it is not used, and the exception for unsafe operations is small (see the **unsafe** vs. **unsafe*** lines in Figures 7, 9, and 11).
- The cost of interposition is quite reasonable for our primary application, contract checking. In real programs, the cost of interposition itself is dominated by the cost of actually checking the contracts. (Contrast the "proxy overhead" lines and "contract checking overhead" lines of Figures 8 and 10; see also the "chaperones versus original" line of Figure 12.)
- Although Racket supports a rich hierarchy of interposition to maintain the language invariants, this additional complexity and the required dynamic checks imposes at most 4% overhead in all of our testing (as reflected by the "additional chaperone overhead" results in Figures 8 and 10). As mentioned in Section 7, JavaScript's proxy design limits its expressiveness by reducing the dynamic checks it performs, but our measurements indicate that these checks are inexpensive.
- Adding support for interposition to the Racket runtime realizes significant performance benefits for existing Racket libraries that otherwise implement interposition manually, such as class contracts and lazy structure contracts (see Figure 12 and the last paragraph of Section 6.3).

7. Related Work

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

7.1 Other Interposition Implementations

The most closely related work to ours is the proxy design (Van Cutsem and Miller 2010) proposed for JavaScript, which is currently implemented in both Firefox and Chrome and used in the benchmarks of section 6. Building on the design of mirages in AmbientTalk (Dedecker et al. 2005; Mostinckx et al. 2009), proxies allow interposition of almost any operation performed on JavaScript objects. Like our design, theirs does not support interposition on some operations, including `instanceof` tests, `typeof` tests, and the 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 for separate kinds of Racket values.

The JavaScript proxy API is in flux; in particular, Van Cutsem and Miller have recently proposed a new design called *direct proxies* (Van Cutsem and Miller 2012) for the proxy system which differs significantly from the original design as implemented in current JavaScript engines. We discuss both designs here.

The initial JavaScript proxy design differed most significantly from chaperones by dispensing with the object being wrapped by the proxy. In other words, a proxy was not a proxy *for* any other object. This simplifies the implementation of some uses of proxies, but in practice, most uses of proxies have a "target," as chaperones do.

This difference lead to the second major difference between JavaScript proxies and our design: 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 importance 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 interposition API is constrained to respect them.

The current JavaScript language does, however, provide reflective operations which can prevent future mutations to a single field, or to an entire object. The original proxy design handled this awkwardly, by producing an entire new object which was then restricted from being mutated. This prevented any further interposition on operations on immutable objects, as well as adding implementation complexity.

These problems, along with discussions with the authors of this paper (chaperones and impersonators were originally added to Racket in May 2010), led Van Cutsem and Miller to propose a new proxy API, dubbed *direct proxies*, which is closely related to our design of chaperones and impersonators. In this design, proxies are always proxies *for* a particular object. Further, proxies are required to respect the mutability constraints of the proxied object—if a field is immutable, the result of a proxied access to the field is checked to be identical to the underlying field. Since JavaScript objects and fields can transition from mutable to immutable during execution, the proxy design does not distinguish ahead of time between chaperones and impersonators; instead the new invariants are enforced once a field has become immutable.

While the direct proxy design is quite similar to ours, it is more limited in a fundamental way: proxies for immutable or otherwise restricted fields must produce *identical* results, whereas chaperones may produce results with further chaperone wrapping, as checked by the `chaperone-of?` procedure. This significantly complicates using JavaScript proxies to implementing higher-order contracts on immutable data, including any contract on methods of an immutable object. The measurements of section 6 demonstrate that these checks impose little overhead, and therefore the JavaScript design could be significantly more expressive with relatively little performance cost. These wrappings do affect the identity of objects, as compared with JavaScript's `===` or Racket's `eq?` comparisons. Since JavaScript, much more than Racket, relies on object identity for comparison, this consideration led the designers of proxies to impose this restriction.[5]

Austin et al. (2011) extend the original JavaScript proxy design to primitive values such as integers and use the system to design a contract system (without blame) for a core JavaScript calculus, including mutable data.

Many other tools that allow unrestricted forms of proxying help to implement contracts but sacrifice the kind of control over invariants that contracts are intended to promote. Notable examples include the MOP (Kiczales et al. 1991), aspect-oriented programming (Kiczales et al. 1997), and `java.lang.reflect.Proxy` (Oracle 2000).

In this vein, our goals with chaperones and impersonators are related to the ideas of *observers and assistants* (Clifton and Leavens 2002), *narrowing advice* (Rinard et al. 2004), and *harmless advice* (Dantas and Walker 2006), and other systems like Open Modules (Aldrich 2005) and EffectiveAdvice (Oliveira et al. 2010) enforce harmless advice by constraining side effects.

Chaperones and impersonators, in contrast, represent a new design point in this space with a different trade-off between enforceable invariants and interposition expressiveness. Specifically, Racketeers already program in a world with mutable state and concurrency, so we do not try to regain the kinds of reasoning that such a combination already invalidates. Instead, since most Racket programs operate mostly with immutable values, chaperones and impersonators are limited to ensure that invariants relating to the behavior of immutable structures are preserved.

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.

7.2 Other Contract Implementations

Eiffel (Meyer 1991), the original embodiment of Design by Contract, supports contracts directly in the language runtime 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 preprepreprocessing since the contracts are more limited, the language is typically less extensible, and the contract system is always less extensible.

Disney (2012) uses JavaScript proxies to implement contracts, producing a system that supports many of the contract system features described in section 2, although lacking extensions such as opaque contracts. We take this as validation of the strategy presented here: design a robust system for low-level interposition, and build a contract system (as well as other applications) on top. Our microbenchmark results indicate that the system is not yet performant under heavy use of contracts.

Several libraries for Haskell (Chitil 2012; 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.

8. Conclusion

New language constructs provide a way for programmers to express and then rely on complex invariants about program behaviors. Implementing some constructs—-such as a full-featured contract system—requires the ability to interpose on the programming language's primitive operations. At the same time, 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.

[5] Personal communication, Tom Van Cutsem, August 2012.

Acknowledgements

Thanks to Tom Van Cutsem for valuable discussions about proxies and chaperones over several years. Thanks to Daniel Brown, Christos Dimoulas, and Matthias Felleisen for their feedback on earlier versions of this work.

The technical appendicies are available in the ACM Digital Library (together with this paper) and online at
`http://sstrickl.net/chaperones/`

Bibliography

Jonathan Aldrich. Open Modules: Modular Reasoning About Advice. In *Proc. European Conf. Object-Oriented Programming*, 2005.

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. Practical Typed Lazy Contracts. In *Proc. ACM Intl. Conf. Functional Programming*, 2012.

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*, 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/`

Tim Disney. Contracts.coffee. 2012. `http://disnetdev.com/contracts.coffee/`

James E. Donnelley. A distributed capability computing system. In *Proc. Intl. Conf. on Computer Communication*, 1976.

Pedro Felzenszwalb and David McAllester. A min-cover approach for finding salient curves. In *Proc. IEEE Wksp. Perceptual Organization in Computer Vision*, 2006.

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*, 2007.

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

Brian Hackett and Shu-Yu Guo. Fast and precise type inference for JavaScript. In *Proc. Conf. on Programming Language Design and Implementation*, 2012.

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.

Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD dissertation, John Hopkins University, 2006.

Stijn Mostinckx, Tom Van Cutsem, Elisa Gonzalez Boix, Stijn Timbermont, Éric Tanter, and Wolfgang De Meuter. Mirror-based reflection in AmbientTalk. *Software—Practice and Experience* 39, pp. 661–699, 2009.

Toby Murray. *Analysing the Security Properties of Object-Capability Patterns*. PhD dissertation, Hertford College, Oxford University, 2010.

Bruno C. d. S. Oliveira, Tom Schrijvers, and William R. Cook. EffectiveAdvice: Disciplined Advice with Explicit Effects. In *Proc. Aspect-Oriented Software Development*, 2010.

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

Susan A. Rajunas. The KeyKOS/KeySAFE system design. Key Logic, Inc, SEC009-01, 1989. `http://www.cis.upenn.edu/~KeyKOS`

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.

Tom Van Cutsem and Mark Miller. On the design of the ECMAScript Reflection API. Vrije Universiteit Brussel, VUB-SOFT-TR-12-03, 2012.